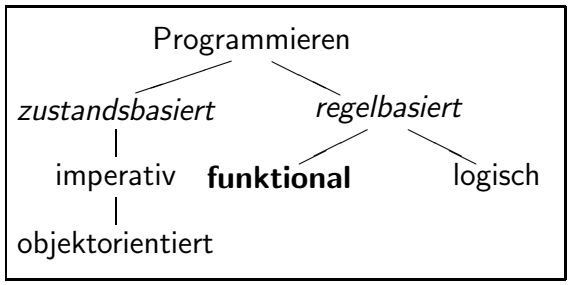
Berthold Hoffmann

hof@informatik.uni-bremen.de

Funktionales Programmieren



Modul 03-05-G–705.03, *Praktische Informatik 3*
Wintersemester 2009/2010

Fachbereich Mathematik und Informatik – Universität Bremen

Vorbemerkungen

Dieser Text beschreibt den Inhalt der Lehrveranstaltung *Praktische Informatik 3* im 3. Semester des Informatikstudiums an der Universität Bremen. Diese Veranstaltung behandelt alles, was jeder Studierende der Informatik über *funktionales Programmieren* wissen sollte.

Unter `www.informatik.uni-bremen.de/agbkb/lehre/pi3/` Weitere Informationen zur Veranstaltung finden sich an zwei Stellen:

- Über `stud.ip` sollen die Studierenden sich Tutorien zuordnen, Übungsaufgaben lesen und abgeben, und sich Termine für Fachgespräche reservieren. Hier können auch im Forum Fragen zur Veranstaltung diskutiert werden.
- Unter der URL `www.informatik.uni-bremen.de/agbkb/lehre/pi3/` stehen das Script, Handbücher zur verwendeten Programmiersprache HASKELL und der benutzen Implementierung `ghci` zur Verfügung.

Dieser Text ist sicher nicht frei von Fehlen Hinweise auf Fehler aller Art und Anregungen für Verbesserungen werden gerne entgegengenommen und – soweit möglich – in zukünftige Fassungen berücksichtigt.

Inhaltsverzeichnis

(Dies ist Fassung 2.0 von 4. Februar 2010.)

1

Einführung

> You can never understand one language
> until you understand at least two.
> *– Ronald Searle, engl. Karrikaturist (1920–)*

In der ersten Veranstaltung werden wir *Funktionales Programmieren* in den Zyklus *Praktische Informatik* einordnen, wesentliche Merkmale des funktionalen Programmierens darstellen und begründen, weshalb jede InformatikerIn dies können sollte. Danach wird auf die Gechichte des funktionalen Programmierens eingegangen und Lehrbücher zu diesem Thema werden vorgestellt.

Außerdem muss einiges zur Organisation der Veranstaltung gesagt werden, was jeweils aktuell im WWW nachzulesen ist, und zwar unter

`www.informatik.uni-bremen.de/agbkb/lehre/pi3/`

1.1 Programmieren – auch eine Frage des *Stils*

In den ersten beiden Veranstaltungen des Zyklus *Praktische Informatik* lernen die Studierenden, *Algorithmen* und *Datenstrukturen* zu entwerfen und in der Programmiersprache Java zu implementieren. Damit haben sie die derzeit meist benutzten Programmierstile kennengelernt: das *imperative Programmieren* und das daraus entwickelte *objektorientierte Programmieren*. Diese Programmierstile sind *zustandsorientiert*. Ihr Rechenmodell bildet die Architektur der heute benutzten Rechner gut ab: In imperativen Programmen verändern Befehle und Prozeduren den Zustand des Speichers, und in objektorientierten Programmen verändern Objekte ihre Zustände durch Nachrichtenaustausch.

Auch wenn diese Stile derzeit in der Praxis dominieren, sind sie doch nicht die einzigen. Das *funktionale Programmieren* beruht allein darauf, Funktionen – im mathematischen Sinne – auf Werte anzuwenden.[1] Dabei werden keine Zustandsvariablen benutzt. Beim *logischen Programmieren* werden Schlussfolgerungen in einer dafür geeigneten Teilmenge der Logik zum Programmieren hergeleitet. Beide Stile werden als *regelbasiert* bezeichnet, weil in ihnen

[1] Manche nennen diesen Stil auch *applikativ*, weil eine seiner wichtigsten Operationen das Anwenden einer Funktion auf ihre Argumente ist.

mit den Mitteln der Mathematik Regeln dafür angegeben werden, *was* ein Programm tun soll, und nicht genau *wie* es ausgeführt werden soll.[2] Beim funktionalen Programmieren sind diese Regeln mathematische Gleichungen, während beim logischen Programmieren logische Schlussregeln definiert werden. Abbildung 1.1 zeigt die Klassifizierung von Programmierstilen.

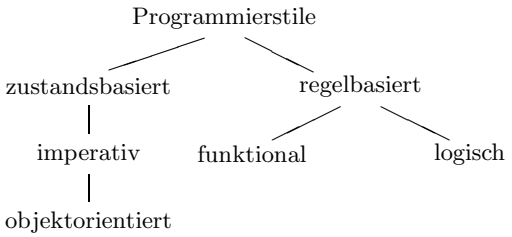


Abb. 1.1. Eine Klassifikation von Programmierstilen

In dieser Lehrveranstaltung werden wir uns mit dem funktionalen Programmieren auseinandersetzen und logische Eigenschaften von Programmen untersuchen.

1.2 Funktionales Programmieren in der Nussschale

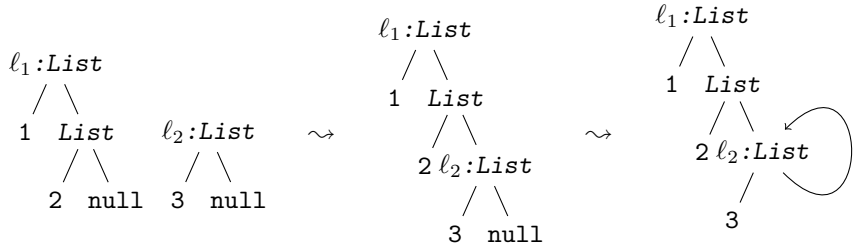
Wir betrachten ein kleines objektorientiertes Programm, an dem wir später die Unterschiede funktionaler Programme erläutern werden.

Beispiel 1.1 (Listen objektorientiert). Die folgende Klasse definiert Listen ganzer Zahlen und eine Methode zum Verketten von Listen in JAVA.

```
class List {
  int entry;
  List next;
  public void cat(List ys){
    List last = next;
    while (last != null)
      last = last.next;
    last = ys;
  }
  ...
}
```

Wenn die Listen ℓ_1 und ℓ_2 die Zahlen 1 und 2 bzw. 3 enthalten, und wir zwei mal die Nachricht ℓ_1`.cat(`ℓ_2`)` senden, bewirkt das folgendes:

[2] In der Literatur findet sich dafür auch die Bezeichnung *deklarativ*.



Wir beobachten:

- Daten sind Objekte, die beliebige Geflechte (Graphen) bilden.
- Objekte haben Zustände.
- Der Aufruf von Methoden verändern den Zustand ihres Empfängerobjekts.
- Methoden können als *Seiteneffekt* auch ihre Argumente oder ein globales Objekte verändern.

Beide Aufrufe von `cat` verändern das Attribut `next` des letzten Objekts in der Liste ℓ_1. Dadurch wird der ursprüngliche Wert von ℓ_1 zerstört. Beim zweiten Aufruf entsteht eine zyklische Liste, weil gilt $\ell_2 = \ell_2$`.next`. Das ist möglicherweise unerwünscht, weil jeder weitere Methodenaufruf ℓ_1`.cat(...)` nicht terminieren würde.

Im Folgenden skizzieren wir, wie das Verketten von Listen in einer funktionalen Sprache definiert werden müsste. Statt Klassen und Methoden werden dazu Datentypen und Funktionen definiert.

1.2.1 Daten sind Wertemengen

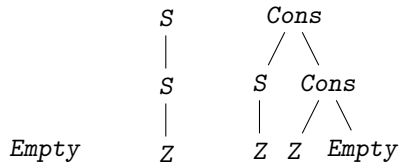
Daten werden in einer funktionalen Sprache oft als *Werte* bezeichnet. Werte sind entweder *atomare Konstruktoren*, z.B. `Z` für die Zahl Null (engl. *zero*) oder `Empty` für die leere Liste, oder aber Anwendungen eines *Konstruktors* auf Werte, wie `S` zur Konstruktion des Nachfolgers einer natürlichen Zahl oder `Cons` zur Konstruktion nicht leerer Listen. Werte werden also als Ausdrücke über Konstruktoren geschrieben.

```
Empty        S (S Z)        Cons (S Z) (Cons Z Empty)
```

Hier verwundert zunächst, wie die Klammern in den Ausdrücken "`S (S Z)`" und "`Cons (S (S Z)) (Cons (S Z) Empty)`" gesetzt werden. Im einfachsten Fall werden die Argumente einer Funktion einfach hinter ihren Namen geschrieben, wie im Teilausdruck "`S Z`". Nur wenn ein Argument zusammengesetzt ist, wie die Argumente "`S Z`" in "`S (S Z)`" oder "`Cons (S Z) Empty`", muss es geklammert werden.[3]

Werte kann man sich als Bäume vorstellen: atomare Konstruktoren sind die Blätter, und die anderen Konstruktoren sind innere Knoten, die auf ihre Argumentwerte verweisen.

[3] Eine bessere Begründung werden wir später noch kennen lernen.



Polymorphe Typen. Werte werden in modernen funktionalen Sprachen anhand von *Typen* klassifiziert. Der Typ `Bool` der *Wahrheitswerte* enthält die atomaren Konstruktoren `False` und `True`.

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

Der Typ `Nat` der *natürlichen Zahlen* enthält die Zahl Null und beliebig viele Nachfolger von Null, die als rekursive Anwendungen des Konstruktors `S` repräsentiert werden.

```
data Nat    = Z | S Nat
```

Listen von natürlichen Zahlen sind ein anderer rekursiver Datentyp: So eine Liste ist entweder leer (dargestellt durch den Konstruktor `Empty`), oder sie wird (mit dem Datenkonstruktor `Cons`) aus einem Element des Typs `Nat` und einer Liste natürlicher Zahlen gebildet.

```
data NatList = Empty | Cons Nat NatList
```

Oft möchte man auch Listen bilden, die andere Werte enthalten, z.B. Wahrheitswerte oder Listen von natürlichen Zahlen. Statt dafür viele weitere Typen `BoolList`, `NatListList` usw. mit ähnlicher Struktur definieren zu müssen, kann man in funktionalen Sprachen einen *polymorphen* Typen `List` definieren (griechisch für "vielgestaltig"):

```
data List t = Empty | Cons t (List t)
```

In dieser Definition ist `t` eine *Typvariable* für die Typen der Elemente. Durch Instanziierung von `t` mit konkreteren Typen wie `Nat`, `List t`, `List Nat` usw. kann man so Listen über beliebigen Typen bilden, z.B. `List Bool`, `List (List t)` oder `List (List Nat)`. Diese Listen sind *homogen*, weil ihre Elemente alle denselben Typ haben müssen.

Jeder Wert hat einen Typ, der aber polymorph sein kann:

```
True :: Bool
S (S (S (S Z))) :: Nat
Empty :: List t
Cons Empty Empty :: List (List t)
Cons (S (S Z)) (Cons Z Empty) :: List Nat
```

Typen stehen schon vor Ausführung des Programms fest; wir sagen deshalb, sie seien *statisch*.

1.2.2 Gleichungen definieren mathematische Funktionen

Funktionen werden mit Gleichungen definiert; ihre Parameter und ihr Ergebnis haben ebenfalls einen statischen Typ. Die Addition von natürlichen Zahlen und die Verkettung von Listen kann so definiert werden:

```
plus :: Nat → Nat → Nat
plus  Z     y = y
plus (S x)  y = S (plus x y)

cat :: List t → List t → List t
cat  Empty       ys = ys
cat (Cons x xs)  ys = Cons x (cat xs ys)
```

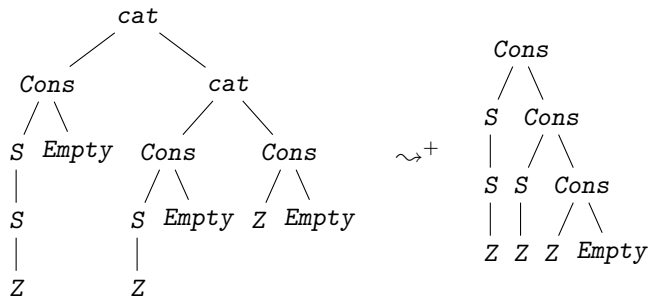
Die Definitionen von `plus` und `cat` zeigen zwei wichtige Elemente von Funktionsdefinitionen: Fallunterscheidung und Rekursion.

- Jeweils zwei Gleichungen behandeln verschiedene Fälle der Funktionen. Durch *Mustervergleich* (*pattern matching*) wird festgestellt, ob der erste Parameter Null bzw. eine leere Liste oder aber eine Zahl größer Null bzw. eine nicht-leere Liste ist und die passenden Gleichungen zur Auswertung ausgewählt.
- Ist die Zahl nicht Null bzw. die Liste nicht leer, werden die Funktionen rekursiv aufgerufen.

Die Gleichungen folgen der Struktur der Datentypen. Wir sagen, die Funktionen seien durch *strukturelle Rekursion* definiert. Weil die rekursiven Aufrufe jeweils kleinere Parameter benutzen, ist diese Art der Rekursion *wohlfundiert* und definert eine terminierende Funktion.

Auswertung. Ein funktionales Programm wird ausgeführt, indem ein Ausdruck gemäß den Funktionsdefinitionen ausgewertet wird. Dazu werden die Definitionen eingesetzt, bis der Ausdruck nur noch Konstruktoren enthält.

```
   cat (Cons (S (S Z)) (cat (Cons (S Z) Empty) (Cons Z Empty))
↝ cat (Cons (S (S Z)) (Cons (S Z) (cat Empty (Cons Z Empty)))
↝ cat (Cons (S (S Z)) (Cons (S Z) (Cons Z Empty))
↝ cat (Cons (S (S Z)) (Cons (S Z) Empty)) (Cons Z Empty)
↝ Cons (cat (S (S Z)) (Cons (S Z) Empty) (Cons Z Empty))
↝ Cons (S (S Z)) (Cons (S Z) (cat (Empty) (Cons Z Empty)))
↝ Cons (S (S Z)) (Cons (S Z) (Cons Z Empty))
```



Auf einen Ausdruck können Gleichungen in der Regel an mehreren Stellen angewendet werden; in unserem Beispiel an zwei Stellen. Hier wurde das innere vor dem äußeren `cat` ausgewertet. Anders herum ginge es auch. Wenn Gleichungen immer auf die innerste aller möglichen Stellen angewendet wird, sprechen wir von *strikter Auswertung* (engl. *strict*, *innermost*, *by value*). Werden die Gleichungen immer auf die äußerste aller möglichen Stellen angewendet, heißt das *verzögerte Auswertung* (engl. *lazy*, *outermost*, *by need*).

Reinheit. In einer funktionalen Sprache werden die Argumente einer Funktion nicht verändert. Deshalb können gleiche Argumentwerte gefahrlos durch ein und denselben Baum repräsentiert werden. Intern werden Werte also als Graphen repräsentiert, ohne dass dies für den Programmierer sichtbar oder relevant wäre.[4] Das Ergebnis einer Funktionen hängt darüber hinaus nur von den Werten ihrer Argumente ab. Jeder Aufruf mit den gleichen Argumente liefert das gleiche Ergebnis. (Das wird *referentielle Transparenz* genannt.)

Aufwand. Die Komplexität einer Funktion kann man so abschätzen: Der Zeitaufwand ist proportional zur Anzahl der Auswertungsschritte (Regelanwendungen). Der Platzbedarf ist proportional zur Anzahl der von einer Funktion aufgerufenen Datenkonstruktoren.[5] Der Zeitaufwand für die Auswertung von `cat` $\ell\ \ell'$ ist also proportional zur Länge von ℓ; das gilt auch für den Platzbedarf, da die Knoten von ℓ kopiert werden. Die Funktion `cat` hat also höheren Platzbedarf als die objektorientierte Methode.

Die funktionale Implementierung der Listenverkettung zeigt fundamentale Unterschiede zum objektorientierten Programm:

- Daten sind Wertemengen.
- Typen sind statisch und polymorph.
- Werte sind unveränderlich, sobald sie einmal konstruiert sind.
- Funktionen haben keinerlei Seiteneffekte.

[4] Im Gegensatz zu JAVA kann man in funktionalen Sprachen nicht unterscheiden, ob zwei Werte durch *denselben* Untergraphen oder durch *zwei gleiche* Untergraphen repräsentiert werden.

[5] Auch der Platz für Zwischenergebnisse in der Rekursion spielt eine Rolle, was wir aber erst später berücksichtigen werden.

Funktionale Programme brauchen etwas mehr Speicher als objektorientierte, während ihr Zeitaufwand meistens vergleichbar bleibt.

1.2.3 Abstraktion und Verifikation

Ein weiteres Konzept und eine weitere Eigenschaft funktionaler Programme sollen hier noch erwähnt werden, weil sie in andere Programmierstilen so nicht anzutreffen sind.

Funktionen höherer Ordnung. Auch Funktionen sind Werte. Eine *Lambda-Abstraktion* `\x → B` konstruiert einen Funktionswert mit Parameter `x` und Rumpf `B`.[6] Die Parameter und das Ergebnis einer Funktion können auch Funktionen sein. Zwei Beispiele für solche Funktionen höherer Ordnung sind:

```
map :: (a → b) → List a → List b
map f Empty       = Empty
map f (Cons x xs) = Cons (f x) (map f xs)

comp :: (b → c) → (a → b) → (a → c)
comp f g x = f (g x)
```

Die Funktion `map` wendet eine Funktion `f` auf alle Elemente einer Liste an. Die Funktion `comp` definiert die Komposition von zwei Funktionen `f` und `g`, wobei zuerst `g` und dann `f`. angewendet wird.

Durch Kombination höherer Funktionen kann man einfach neue Funktionen konstruieren, z.B. eine Funktion, die die Zahlen einer Liste verdoppelt, und eine Funktion, die eine Funktion zweimal anwendet:

```
doublelist :: List Nat → List Nat
doublelist = map (\x → plus x x)

twice :: (a → a) → a → a
twice f = comp f f
```

Eine Auswertung der Funktion `doublelist` liefert:

```
    doublelist (Cons (S (S Z)) (Cons (S Z) Empty))
⇝+ Cons (S (S (S (S Z)))) (Cons (S (S Z)) Empty)
```

Formale Eigenschaften. Funktionales Programmieren erleichtert den Nachweis formaler Eigenschaften eines Programms.

Jede Gleichung einer Funktionsdefinition definiert ein Axiom für das Verhalten der Funktion. Für die Verkettung gilt also:

$$\begin{aligned} \forall \ell \in \mathit{List}\ \tau : &\quad \mathit{cat}\ \mathit{Empty}\ \ell = \ell \\ \forall x \in \tau, \ell, \ell' \in \mathit{List}\ \tau : &\quad \mathit{cat}\ (\mathit{Cons}\ x\ \ell)\ \ell' = \mathit{Cons}\ x\ (\mathit{cat}\ \ell\ \ell') \end{aligned}$$

[6] Das "\" ist ein "verstümmeltes" Lambda (λ), was im Lambdakalkül für Funktionen benutzt wurde.

Dies erleichtert den Nachweis anderer Eigenschaften von Funktionen. Beispielsweise ist die Verkettung assoziativ, und hat `Empty` als neutrales Element.

$$\begin{aligned} \forall \ell \in \mathit{List}\ \tau: &\quad \mathit{cat}\ \mathit{Empty}\ \ell = \ell = \mathit{cat}\ \ell\ \mathit{Empty} \\ \forall \ell, \ell', \ell'' \in \mathit{List}\ \tau: &\ \mathit{cat}\ (\mathit{cat}\ \ell\ \ell')\ \ell'' = \mathit{cat}\ \ell\ (\mathit{cat}\ \ell'\ \ell'') \end{aligned}$$

(Auch die Funktion `plus` ist assoziativ, und hat `Z` als neutrales Element.) Zwischen den höheren Funktionen `map` und `comp` bzw. `twice` gibt es folgende Zusammenhänge:

$$\forall f \in \beta \to \gamma, g \in \alpha \to \beta: \mathit{comp}\ (\mathit{map}\ f)(\mathit{map}\ g) = \mathit{map}\ (\mathit{comp}\ f\ g) \tag{1.1}$$

$$\forall f \in \alpha \to \alpha: \mathit{twice}\ (\mathit{map}\ f) = \mathit{map}\ (\mathit{twice}\ f) \tag{1.2}$$

Solche Eigenschaften braucht man, um die Korrektheit von Funktionen zu zeigen, die mit Funktionen wie `cat`, `map`, `comp` und `twice` definiert sind. Darüber hinaus kann man diese Eigenschaften auch benutzen, um Funktionsdefinitionen zu transformieren, um sie effizienter zu machen. Diese Eigenschaften können mit *struktureller Induktion* gezeigt werden. Hier ein einfaches Beispiel:

Theorem 1.1 (`Empty` ist rechtsneutral für `cat`).

$$\forall \ell \in \mathit{List}\ \tau: \mathit{cat}\ \ell\ \mathit{Empty} = \ell \tag{1.3}$$

Beweis. Durch Induktion über die Struktur von `List` τ.
Induktionsanfang. Sei $\ell =$ `Empty` in (1.3). Dann gilt:

$$\begin{aligned} & \mathit{cat}\ \mathit{Empty}\ \mathit{Empty} & \\ = \quad & \mathit{Empty} & \text{1. Gleichung von } \texttt{cat} \end{aligned}$$

Induktionsannahme. Gelte (1.3) für beliebige Listen ℓ der Länge $n \geqslant 0$.
Induktionsschritt. Wir zeigen, dass dann (1.3) für alle Liste der Länge $n+1$ gilt. So eine Liste hat *oBdA* (ohne Beschränkung der Allgemeinheit) die Form `Cons` $x\ \ell$, wobei ℓ höchstens n Elemente enthält. Dann gilt:

$$\begin{aligned} & \mathit{cat}\ (\mathit{Cons}\ x\ \ell)\ \mathit{Empty} & \\ = \ & \mathit{Cons}\ x\ (\mathit{cat}\ \ell\ \mathit{Empty}) & \text{2. Gleichung von } \texttt{cat} \\ = \quad & \mathit{Cons}\ x\ (\mathit{Empty}) & \text{nach Induktionsannahme} \\ = \quad & \mathit{Cons}\ x\ \ell & \end{aligned}$$

Damit gilt (1.3) für Listen beliebiger Länge. □

1.3 Weshalb funktional programmieren lernen?

Das wird sich mancher Studierende fragen. Schließlich wird in der Praxis doch überwiegend objektorientiert programmiert!

Beim funktionalen Programmieren werden wir abstrakte und mächtige Wege kennenlernen, Datenstrukturen und Algorithmen zu definieren. Weil dieser Stil sich an der Mathematik orientiert, können wir auch formale Eigenschaften von Programmen leichter untersuchen, wie z. B. die Korrektheit. Schon aus diesen Gründen sollte jede Informatikerin und jeder Informatiker die Prinzipien des funktionalen Programmierens kennen. Auch wenn das (noch) nicht in jeder Stellenausschreibung verlangt wird, ist es doch für einige Hauptstudiumsveranstaltungen sehr nützlich (wie *Programmiersprachen*, *Übersetzer*, *Algebraische Spezifikation*, *Techniken zur Entwicklung korrekter Software*). Außerdem werden wir sehen, dass viele Konzepte, die jetzt nach und nach in objektorientierte Sprachen und Skriptsprachen Eingang finden, in modernen funktionalen Sprachen schon selbstverständlich sind. Wir wagen also einen Ausblick in die *Zukunft des Programmierens*!

In Übertragung des Zitats vom Kapitelanfang gilt außerdem: Wir werden das Wesen des objektorientierten Programmierens besser begreifen, wenn wir auch funktionales Programmieren verstehen.

1.3.1 Einwände gegen Funktionales Programmieren

Einige Einwände gegen das funktionale Programmieren werden immer wiedervorgebracht. Darauf wollen wir hier kurz eingehen.

Funktionales Programmieren sei nicht praktisch einsetzbar. Moderne funktionale Sprachen unterstützen Module und getrennte Übersetzung. Sie erlauben damit auch die Entwicklung großer Systeme, gerade weil sie auch Konzepte wie Polymorphie und Funktionen höherer Ordnung in einem Maße unterstützen wie kein anderer Programmierstil. So wurde beispielsweise der Theorembeweiser Isabelle ganz in der funktionalen Sprache Ml entwickelt, und in unserer Forschungsgruppe wird ein *Heterogeneous Tool Set* für formale Spezifikation weitgehend in Haskell entwickelt. Die Sprache Erlang [Arm96] wird bei Sony-Ericsson im großen Stil für die Entwicklung von Telekommunikations-Software benutzt, und Lisp wird vielfach benutzt – allerdings oft wie eine imperative oder objektorientierte Sprache.

Funktionale Programme seien ineffizient. Imperative und objektorientierte Programme können die Eigenschaften heutiger Rechner besser ausnutzen. Zustandsbehaftete Datentypen wie doppelt verkettete Listen können funktional nicht programmiert werden. Funktionale Programme benötigen daher oft mehr Speicherplatz als objektorientierte Programme; sie können aber meistens etwa gleich schnell ausgeführt werden.

Wie überall hat Abstraktion ihren Preis, der jedoch in diesem Fall nicht allzu hoch ist. Dafür ist es in der Regel einfacher, funktionale Programme zu erstellen. Diese Effizientz (des Programmiervorgangs) ist auch sehr wichtig.

Funktionales Programmieren sei schwer erlernbar. Dies wird gegen jeden neuen Programmierstil eingewendet. Kein Programmierstil kann die inhärente

Komplexität des Programmierens übertünchen, aber wer funktional programmiert, lernt, Datenstrukturen und Algorithmen *abstrakt* zu formulieren und kann übersichtliche und klare Programme schreiben – auch in anderen Programmierstilen.

1.4 Rechnen mit Funktionen

Beim funktionalen Programmieren fassen wir ein Programm auf als eine mathematische Funktion

$$p\colon \mathit{Eingabe} \to \mathit{Ausgabe}$$

wobei *Eingabe* und *Ausgabe* Mengen sind. Die Ausgabe hängt dabei ausschließlich von den Werten der Eingabe ab. Bei jeder Anwendung von p auf einen Eingabewert e wird also der gleichen Wert $a = p(e)$ ausgegeben, und die Eingabe e wird nicht verändert. Diese Eigenschaft heißt *referentielle Transparenz*; sie gilt für p und alle weiteren Hilfsfunktionen, die in einem funktionalen Programm definiert werden. Funktionsprozeduren, wie wir sie in imperativen oder objektorientierten Sprachen kennen, etwa die Methoden in Java, sind dagegen *nicht* referentiell transparent, weil ihre Ergebnisse vom Zustand globaler Variablen oder Dateien abhängen können, die *implizit* benutzt und auch verändert werden dürfen. In funktionalen Programmen gibt es generell keine Zustände und Zustandsvariablen, was den Programmierstil grundlegend verändert. Variablen gibt es zwar schon, aber nur im *mathematischen Sinne*, das heißt, als Platzhalter für unveränderliche Werte.

Funktionen werden durch Gleichungen definiert, etwa die – lächerlich einfachen – Funktionen *pred* und *binom*:

$$\begin{aligned} \mathit{pred}&\colon \mathbb{Z} \to \mathbb{Z} \\ \mathit{pred}(z) &= z - 1 \\ \mathit{binom}&\colon \mathbb{R} \times \mathbb{R} \to \mathbb{R} \\ \mathit{binom}\, a\, b &= (a + b) \cdot (a + b) \end{aligned}$$

Die Gleichungen besagen, dass für alle Eingaben $z \in \mathbb{Z}$ der Wert von *pred* gleich $z - 1$ sein soll, und bei *binom* entsprechend für alle Eingaben $a, b \in \mathbb{Z}$ das Ergebnis $e = (a + b) \cdot (a + b)$ liefern soll. Sie stützen sich auf vordefinierte Funktionen wie Addition, Subtraktion und Multiplikation ab.

Diese Art der Definition erleichtert die Untersuchung von Eigenschaften von Funktionen, weil jede Definition eine universelle Gleichheit beschreibt, die bei semantischen Untersuchungen benutzt werden kann. So können wir mithilfe der binomischen Formel leicht nachweisen, dass die Funktion

$$\begin{aligned} \mathit{binom}'&\colon \mathbb{R} \times \mathbb{R} \to \mathbb{R} \\ \mathit{binom}'\, a\, b &= a^2 + 2ab + b^2 \end{aligned}$$

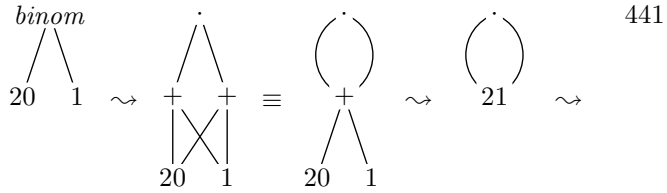
äquivalent zu *binom* ist. Bei ähnlichen Überlegungen zu einer Prozedur oder Methode müsste man alle implizit verwendeten Größen und ihre mögliche Zustände mit betrachten.

Ein funktionales Programm wird ausgeführt, indem ein Aufruf gemäß der definierenden Gleichungen ausgewertet wird. Etwa:

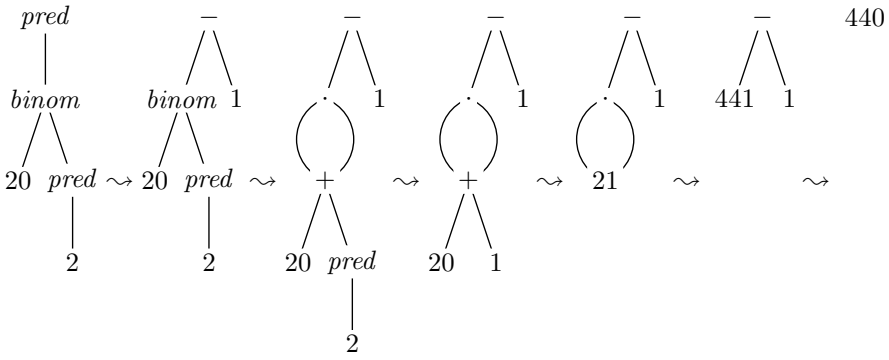
$$\begin{aligned} &\mathit{pred}\,42 \leadsto 42-1 \;\leadsto\; 41\\ &\mathit{binom}\,20\,1 \leadsto (20+1)\cdot(20+1) \;\leadsto\; 21\cdot(20+1) \;\leadsto\; 21\cdot 21 \;\leadsto\; 441 \end{aligned}$$

Diese Art von Ausführung heißt "*Reduktion auf Normalform*", weil Ausdrücke wie $\mathit{pred}\,42$ solange mithilfe der Gleichungen umgeformt werden, bis keine Gleichung mehr anwendbar ist. Die Auswertung von vordefinierten Funktionen kann als Anwendung vordefinierter Gleichheiten wie $42-1=41$ und $21\cdot 21=441$ verstanden werden.

Die Ausdrücke kann man sich dabei als *Bäume* vorstellen, deren Blätter Werte und deren innere Knoten Funktionen sind, unter anderem vordefinierte Operationen wie Addition. Wegen der referentiellen Transparenz kann man gleiche Teilbäume auch zusammenfallen lassen, weil sie ohnehin auf die gleichen Normalformen reduziert werden. Die Ausführung von *binom* kann als Graphreduktion also folgendermaßen dargestellt werden:



Auch geschachtelte Ausdrücke können ausgewertet werden.Dabei werden die Regeln *von außen nach innen* und *von links nach rechts* angewendet, wobei gleiche Teilausdrücke zusammenfallen und also alle auf einmal ausgewertet werden:



Diese Auswertungsreihenfolge heißt *verzögerte Auswertung* (engl.: *evaluation by need* oder *lazy evaluation*).

Funktionales Programmieren ist also im Prinzip sehr einfach, und wir werden sehen, dass moderne funktionale Sprachen – wie die in dieser Veranstaltung benutzte Sprache Haskell – mächtige Konzepte zur Definition

von Wertemengen und Funktionen enthalten, unter anderem Polymorphie, Mustervergleich (*pattern matching*) und Funktionen höherer Ordnung, die es in objektorientierten Sprachen nicht gibt.

1.5 Funktionales Programmieren in der Nussschale

Wie in jedem anderen Programmierstil sollen mit funktionalen Programmen Daten und Algorithmen modelliert werden.

Wir haben in Abschnitt 1.4 schon gesehen, dass Algorithmen als (mathematische) Funktionen definiert werden. In diesem Abschnitt wollen wir noch etwas genauer betrachten, wie die Berechnungsvorschriften und Datentypen in funktionalen Programmen beschrieben werden.

Mathematische Funktionen

In der Mathematik wird eine Funktion f angegeben als

$$f\colon A \to B$$
$$f(x) = E$$

Hierbei ist A die *Definitionsmenge*, B die *Wertemenge* und die Gleichung "$f(x) = E$" die *Funktionsvorschrift*. Eine Funktion definiert eine Relation

$$R_f = \{(x, f(x) \mid x \in A\} \subseteq A \times B$$

die *Funktionsgraph* genannt wird und *rechtseindeutig* sein muss:

$$\text{Für alle } x \in A, y, y' \in B : \text{Wenn } (x, y) \in R_f \text{ und } (x, y) \in R_f, \text{ dann } y = y'$$

Im Allgemeinen nehmen wir *nicht* an, dass jede Funktion auch *linkstotal* sein muss:

$$\text{Für alle } x \in A : \text{Es gibt ein } y \in B \text{ so dass } (x, y) \in R_f$$

Es kann also Elemente $x \in A$ geben, für die f *undefiniert* ist; das schreiben wir manchmal als "$f(x) = \perp$". So eine Funktion heißt *partiell*.

Fallunterscheidung und Rekursion

Die rechte Seite E der Funktionsvorschrift ist ein *Ausdruck*, in dem typischerweise x auftritt, und durch dessen Auswertung für jedes Element x der Definitionsmenge A ein Wert $f(x) \in B$ bestimmt wird.

Beispiel 1.2 (Fakultätsfunktion). Die sattsam bekannte Fakultätsfunktion berechnet das Produkt aller positiven Zahlen bis zu einer Zahl n.

$$\begin{aligned} fac &: \mathbb{N} \to \mathbb{N} \\ fac(n) &= 1 \cdot 2 \cdot \dots \cdot n \end{aligned}$$

Sie kann mit folgender Berechnungsvorschrift definiert werden:

$$\text{Für alle } n \in \mathbb{N} : fac(n) = \begin{cases} 1 & \text{wenn } n = 1 \\ n \cdot fac(n-1) & \text{sonst} \end{cases}$$

Die Berechnungsvorschrift definiert den Funktionsgraphen

$$R_{fac} = \{(1,1), (2,2), (3,6), (4,24), (5,120), \dots\}$$

Fasst man – wie in der Informatik üblich – $\mathbb{N}$ als die Menge $\{0, 1, 2, \dots\}$ auf, ist die Funktion *fac partiell*, weil sie für die Eingabe 0 *undefiniert* ist. Also gilt $fac\ 0 \rightsquigarrow \bot$.

Die Berechnungsvorschrift für *fac* enthält typische Elemente von Ausdrücken:

- Der Ausdruck setzt Operationen wie Gleichheit, Multiplikation und Subtraktion als anderswo definiert voraus; praktisch sind solche Operationen für einfache Wertemengen wie $\mathbb{N}$ *vordefiniert*.
- Eine *Fallunterscheidung* wählt einen Teilausdruck in Abhängigkeit von x aus.
- Die Vorschrift ist *rekursiv*; sie benutzt *fac*; weil das Argument $n - 1$ des rekursiven Aufrufs kleiner ist als der Wert, für den *fac* definiert wird, ist die Rekursion wohlfundiert.

Beispiel 1.3 (Fakultät). Die Definition von Beispiel 1.2 kann ganz einfach in Haskell umgesetzt werden. Da in Haskell aber nur die ganzen Zahlen (unter dem Namen `Integer`) vordefiniert sind, würde die Berechnungsvorschrift oben alle nicht-positiven Werte der Definitionsmenge undefiniert lassen. Deshalb werden wir für $n \leqslant 0$ einen Fehlermeldung als Ergebnis liefern (`error`).

```
fac :: Integer → Integer
fac n = if      n ≤ 0 then error "call␣of␣fac(n≤0)"
        else if n == 1 then 1
                       else n * fac (n-1)
```

In Haskell können Fallunterscheidungen auch als *bedingte Gleichungen* geschrieben werden, was der mathematischen Schreibweise noch näher kommt.

```
fac' :: Integer → Integer
fac' n | n ≤ 0    = error "call␣of␣fac(n≤0)"
       | n == 1   = 1
       | otherwise = n * fac (n-1)
```

Später werden wir noch sehen, dass eine Funktion wie *fac* auch ohne explizite Rekursion fast wie in Definition (1.4) definiert werden kann:

$$fac'' \ : \ \mathbb{N} \to \mathbb{N}$$

Für alle $n \in \mathbb{N} : fac''(n) = product(fromTo\ 1\ n)$

Dabei definiert *fromTo u o* eine Liste von ganzen Zahlen i mit $u \leqslant i \leqslant o$, und *product l* berechnet das Produkt einer Zahlenliste l. Solche Funktionen heißen *Kombinatoren*; oft sind sie vordefiniert, können aber vom Programmierer auch selbst eingeführt werden. In HASKELL könnte man sogar ganz einfach schreiben

```
fac'' :: Integer → Integer
fac'' n = product [1..n]
```

Funktionen können auch *direkt* definiert werden.

$fac''' \ : \ \mathbb{N} \to \mathbb{N}$

$fac''' = \lambda n \to product(fromTo\ 1\ n)$

Ein Ausdruck der Form $(\lambda x \to E)$ heißt *Lambda-Abstraktion*, er liefert als Ergebnis die namenlose Funktion, die, wenn sie (später) auf einen Argumentausdruck A angewendet wird, den Wert des Ausdrucks $E[x/A]$ berechnet. (Der Ausdruck $E[x/A]$ entsteht aus E, wenn jedes Auftreten des formalen Parameters x durch den aktuellen Parameter A ersetzt wird.) In HASKELL wird das so geschrieben

```
fac''' :: Integer → Integer
fac''' =  n → product [1..n]
```

Daten

Wie in allen Programmiersprachen sind Basisdatentypen wie verschiedene Arten von Zahlen, Wahrheitswerte und Zeichen mit den für sie typischen Funktionen und Operationen vordefiniert.

Auch zusammengesetzte Datentypen sind Wertemengen, und zwar Bäume über Datentypen.

Zusammengesetzte Datentypen können aus unterschiedlich aufgebauten Werten bestehen, die dann auch rekursiv Werte des zu definierenden Datentyps enthalten können. Binäre Bäume über ganzen Zahlen können so beschrieben werden:

$$IntTree = Leaf\ \mathbb{Z} \parallel Branch\ IntTree\ IntTree$$

Ein Wert vom Typ *IntTree* ist also entweder ein Blatt vom Typ $\mathbb{Z}$, oder eine Verzweigung mit zwei Kindern, die wieder vom Typ *IntTree* sind. Da die Blätter und Verzweigungen anhand ihrer Wurzel unterschieden werden können, modelliert *IntTree* die *Summe* (oder *disjunkte Vereinigung*) von zwei

Wertemengen, ganzen Zahlen (an den Blättern) und Paaren von Bäumen (an den Verzweigungen): $\mathbb{Z} + \mathit{IntTree} \times \mathit{IntTree}$. Diese Art von Datentypen werden *algebraisch* genannt, denn sie sind Summen von kartesischen Produkten.

Aus funktionaler Sichtweise werden *Leaf* und *Branch* als Funktionen aufgefasst, die aus Werten neue Werte aufbauen; sie werden daher *Konstruktorfunktionen* oder einfach *Konstruktoren* genannt.

Funktionen über zusammengesetzten Werten werden auch mit Fallunterscheidungen und Rekursion definiert. Dabei kann man oft auf die in objektorientierten Sprachen benutzen Selektorfunktionen für die Komponenten zusammengesetzter Werte verzichten. Statt dessen wird der Aufbau eines Wertes durch Mustervergleich (*pattern matching*) abgefragt und dabei gleichzeitig ihre Komponenten selektiert. Dies wollen wir anhand der Funktion *height* illustrieren, die die Höhe eines Baumes bestimmt:

$$\begin{aligned} \mathit{height}\colon\ & \mathit{IntTree} \to \mathbb{N} \\ \mathit{height}(\mathit{Leaf}\ n) &= 1 \\ \mathit{height}(\mathit{Branch}\ l\ r) &= 1 + \mathit{max}(\mathit{height}\ l)(\mathit{height}\ r) \end{aligned}$$

(Die Funktion $\mathit{max}\colon \mathbb{Z} \times \mathbb{Z} \to \mathbb{Z}$ bestimmt die Größere von zwei Zahlen.) Jede der beiden Gleichungen für *height* behandelt eine Art von Bäumen, Blätter bzw. Verzweigungen, indem ein Muster auf der linken Seite angegeben wird. Die in den Mustern auftretenden Namen n, l und r wählen die Komponenten aus, die dann im Ausdruck auf der rechten Seite benutzt werden können.

Bewahrung von Argumenten

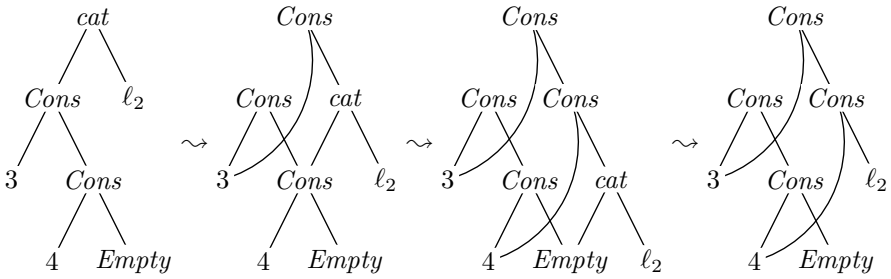
Eine weitere Besonderheit funktionaler Programme wollen wir an *dem* funktionalen Datentyp *an sich* erläutern: Zahlen-Listen sind entweder leer, oder entstehen, wenn eine Zahl einer Liste vorangestellt wird:

$$\mathit{IntList} = \mathit{Empty} \parallel \mathit{Cons}\ \mathbb{Z}\ \mathit{IntList}$$

Empty ist ein Wert ohne Komponenten. Das Verknüpfen zweier Listen kann mit der Funktion *cat* so definiert werden:

$$\begin{aligned} \mathit{cat}\colon\ & \mathit{IntList} \to \mathit{IntList} \to \mathit{IntList} \\ \mathit{cat}\ \mathit{Empty}\ \ell &= \ell \\ \mathit{cat}(\mathit{Cons}\ h\ t)\ \ell &= \mathit{Cons}\ h(\mathit{cat}\ t\ \ell) \end{aligned}$$

Ein Aufruf $\mathit{cat}\ \ell_1\ \ell_2$ konstruiert eine neue Liste, in der die Elemente der Liste ℓ_1 der Liste ℓ_2 vorangestellt werden. Die Argumente ℓ_1 und ℓ_2 beiben unverändert.



Für jeden Knoten von ℓ_1 wird *cat* also einmal aufgerufen und jeweils ein neuer Knoten aufgebaut. In einem imperativen oder objektorientierten Programm würde man eher den Abschluss *Empty* von l_1 mit l_2 überschreiben. Dann wird aber der Wert l_1 zerstört, und alle anderen Werte, die ℓ_1 ebenfalls als Teil enthalten, werden als Seiteneffekt ebenfalls verändert.

In funktionalen Programmen können solche Seiteneffekte nicht auftreten, weil einmal bestimmte Werte (wie ℓ_1) nicht mehr verändert werden dürfen. Das wird mit höherem Speichebedarf bezahlt: *cat* hat linearen Speicherbedarf (statt konstantem Aufwand in Java). Dafür kann ein einmal konstruierter Wert beliebig oft benutzt werden, ohne dass er jemals durch Seiteneffekte verändert werden könnte. Die Liste ℓ_2 muss also nicht kopiert werden, und auch nicht die Elemente der Liste (die ja auch zusammengesetzte Werte sein könnten). Durch den Verzicht auf selektives Verändern wird eine häufige Fehlerquelle in imperativen Programmen vermieden.

Trotzdem wird nur die Liste ℓ_1 kopiert; die Liste ℓ_2 wird im Ergebnis von *cat* unverändert wieder benutzt. Listenwerte sind also eigentlich *Graphen*, nicht Bäume. Dies ist aber für den Programmierer nicht relevant, weil wieder benutzte Werte unverändert bleiben.

Polymorphie

Beim Betrachten der Definitionen von *height* und *cat* fällt auf, dass eigentlich egal ist, welche Werte in Listen oder Bäumen enthalten sind: Die Höhe der Bäume bleibt die gleiche, und sie müssen genauso verkettet werden. Diese Beobachtung kann man sich zunutze machen, indem man Typdefinitionen mit *Typparametern* (oder *Typvatiablen*) parametrisiert. Listen und Binärbäume über beliebigen Typen α könnten allgemein so beschrieben werden, und auch Paare und Tripel können parametrisiert definiert werden:

$$\mathit{List}\,\alpha = \mathit{Empty} \parallel \mathit{Cons}\,\alpha(\mathit{List}\,\alpha)$$
$$\mathit{Tree}\,\alpha = \mathit{Leaf}\,\alpha \parallel \mathit{Branch}(\mathit{Tree}\,\alpha)(\mathit{Tree}\,\alpha)$$

Die Typen *IntTree* und *IntList* sind *Instanzen* dieser Typen:

$$\mathit{IntTree} = \mathit{Tree}\,\alpha$$
$$\mathit{IntList} = \mathit{List}\,\alpha$$

Die Funktionen *height* und *cat* könnten mit folgender Signatur über Bäumen bzw. Listen beliebigen Elementtyps definiert werden, weil ihre Gleichungen die Elemente der Bäume und Daten nicht betrachten:

$$height : (Tree\ \alpha) \to \mathbb{N}$$
$$cat : (List\ \alpha) \to (List\ \alpha) \to (List\ \alpha)$$

Funktionen mit parametrisierten Typen werden *polymorph* genannt.

In funktionalen Sprachen sind auch Typnamen wie *IntTree* und *Tree* nur Funktionen, die *Typkonstruktoren* genannt werden: Der parameterlose Typkonstruktor *IntTree* definiert Bäume über Zahlen, und der parametrisierte Typkonstruktor *Tree* liefert Bäume über beliebigen Typen, die für α eingesetzt werden. Dabei ist es wichtig, das an allen Stellen der gleiche Typ für die Typparameter eingesetzt wird. Bäume und Listen sind also *homogen* bezüglich der Typen ihrer Elemente.

Funktionale

Eine letzte Besonderheit funktionaler Sprachen soll hier nur noch kurz erwähnt werden: Auch Funktionen sind "nur" Werte, die Komponenten von Datenstrukturen sein können und als Parameter oder Ergebnisse von Funktionen auftreten können. Der Typ *Tree* kann also auch so instantiiert werden:

$$FunTree = Tree(\mathbb{R} \to \mathbb{R})$$

So ist $trigFuns = Cons \sin (Cons \cos Empty)$ ein Wert vom Typ *FunTree* und könnte als Parameter für entsprechend getypte Funktionen verwendet werden, aber auch von einer Funktion berechnet werden. Solche Typen und Funktionen *höherer Ordnung* werden wir aber erst später behandeln.

Beispiel 1.4 (Bäume und Listen). Die Definitionen von Typen und Funktionen können einfach in Haskell übertragen werden.

```
data Tree a = Leaf a | Branch (Tree a) (Tree a) deriving Show
height :: (Tree a) → Integer
height (Leaf n)       = 1
height (Branch l  r) = max (height l) (height r)

data LIST a = EMPTY | CONS a (LIST a) deriving Show
cat :: (LIST a) → (LIST a) → (LIST a)
cat EMPTY      l  = l
cat (CONS x l) l' = CONS x (concatenation l l')
```

In Ermangelung griechischer Buchstaben auf Tastaturen kann man in Haskell leider nur lateinische Buchstaben verwenden.

Der Zusatz "`deriving Show`" in den Definitionen von `Tree` und `LIST` besagt, dass für diese Typen automatisch Funktionen

```
show :: (Tree a) → String
show :: (LIST a) → String
```

hergeleitet werden, mit denen Werte dieser Typen als Zeichenkette dargestellt werden können. Diese Funktionen braucht man, wenn man sich im `ghci`-System die Ergebnisse von Ausdrücken anzeigen lassen will, die Bäume oder Listen sind.

Listen und Produkte werden so häufig benutzt, dass sie in HASKELL vordefiniert sind. Für sie gibt es spezielle Schreibweisen, die erst im nächsten Kapitel behandelt werden.

1.6 Geschichte

Die Geschichte des funktionalen Programmierens beginnt mit Schönfinkels und Currys Arbeiten über *kombinatorische Logik* [Sch24, CF58]. Zusammen mit Churchs Arbeiten zum *Lambda-Kalkül* [Chu41] bilden sie die mathematische Grundlage des funktionalen Programmierens. Auf Basis des Lambda-Kalküls entwickelte McCarthy die erste "funktionale" Sprache LISP [McC60], aus der Abelson und Sussman die heute noch benutzte Sprache SCHEME [AS93] entwickelten. Weiter wurde die Sprache als COMMON LISP standardisiert; sie wird in der künstlichen Intelligenz benutzt. LISP und ihre Nachfolgerinnen können nur mit Einschränkungen als funktional bezeichnet werden, weil ihnen viele Konzepte moderner funktionaler Sprachen fehlen, und in der Praxis meistens imperativ oder objektorientiert in ihnen programmiert wird.

Die Geschichte der "modernen" funktionalen Sprachen beginnt mit Landins visionärem Artikel über die nicht implementierte Sprache ISWIM ("*If you see what I mean*", [Lan66]), die viele Sprachen beeinflusst hat, unter anderem Milners STANDARD ML [Pau91] und Turners MIRANDA [Tur86]. In seinem Turing-Vortrag hat Backus die Vorteile eines funktionalen Stils gegenüber dem imperativen Programmieren gepriesen und eine sehr *spezielle* funktionale Sprache vorgestellt: FP [Bac78]. Die Sprache HASKELL [PJ+02] hat sich in der Fassung von 1998 zu einem Standard entwickelt. Sehr ähnlich dazu ist die Sprache CLEAN [BvEvLP87].

1.7 Lehrbücher

Es gibt einige Lehrbücher zum funktionalen Programmieren, auch auf Deutsch. Auch wenn sie den Stoff der Vorlesung nicht genau so darstellen wie wir, eignen sie sich als Ergänzung oder Hintergrundlektüre.

Im Buch *Funktionale Programmierung* von Peter Pepper wird das Thema etwas behäbig behandelt, und die Programme werden zunächst in der wenig verbreiteten Sprache OPAL entwickelt, und dann in die Sprachen ML und HASKELL übertragen. ([Pep03], 300 Seiten, €29,95)

Manuel M. T. Chakravarty und Gabriele Keller haben eine *Einführung in die Programmierung mit* Haskell für absolute Programmieranfänger verfasst, die auch eine Einführung in Unix enthält ([CK04], 199 Seiten, €29,95).

Gert Smolka hat eine sehr gute theoretische Einführung in das Programmieren geschrieben, in der das funktionale Programmieren mit Standard ML im Mittelpunkt steht ([Smo08], 371 Seiten, €34,80).

Simon Thompsons *Craft of Functional Programming* ist das Standardbuch über das Programmieren mit Haskell ([Tho99], 487 Seiten, €39,47 bei `amazon.de`).

Im Buch *Algorithms – A Functional Programming Approach* von Fethi Rabhi und Guy Lapalme stehen Datenstrukturen und Algorithmen im Vordergrund ([RL99], 235 Seiten, €69,90 bei `amazon.de`); deshalb werden einige Konzepte der Sprache Haskell (wie die Monaden) nur am Rande behandelt.

Paul Hudaks anspruchsvolles Buch *The* Haskell *School of Expression* zeigt mit ungewöhnlichen Beispielen aus Graphik, Robotik und Musik, dass die Sprache Haskell universell einsetzbar ist ([Hud00], 416 Seiten, €25,95 bei `amazon.de`).

Graham Huttons Buch *Programming in Haskell* ist eine knappe und übersichtliche Einführung in die Konzepte von Haskell ([Hut07], 171 Seiten, €24,95 bei `amazon.de`).

Das Buch *Real World Haskell* von Bryan O'Sullivan, John Goerzen und Don Stewart erklärt die Benutzung von Haskell an Hand realistischer Problemstellungen; dieser Text ist sehr detailliert und zum Selbststudium gedacht ([OGS08], 671 Seiten, €39,95 bei `amazon.de`).

Darüber hinaus sind viele Einführungen in Haskell im Internet verfügbar; unter anderem auch das vollständige Buch *Real World Haskell* [OGS08].

Der Artikel [Ode05] von Martin Odersky im Informatik-Handbuch fasst das Wesentliche des funktionalen Programmierens zusammen.

1.8 Haskells Eck

Von der Sprache Haskell haben wir noch nicht viel kennen gelernt, weil es hier ja *allgemein* um funktionales Programmieren ging. Trotzdem haben wir für die Definitionen von Typen und Funktionen die Notation von Haskell benutzt, in sehr, sehr einfacher Form. Die in `Schreibmaschinenschrift` angegebenen Definitionen lassen sich mit dem `ghci`-System laden und ausführen.

Diesen Kern von Haskell rekapitulieren wir hier. Außerdem beschreiben wir die Benutzung des Glasgower Haskell-Systems, und Kommentare, weil dies für die Bearbeitung der Übungsaufgaben nützlich ist.

1.8.1 Der Kern

In diesem Kapitel haben wir schon einiges über funktionale Programme im Allgemeinen erfahren. Damit haben wir auch den Kern von Haskell kennengelernt, in dem Datentypen und Funktionen mit Gleichungen definiert werden.

Es treten drei verschiedene Arten von Ausdrücken über fünf Arten von Namen auf (siehe Abbildung 1.2):

- *Typausdrücke* sind geschachtelte Anwendungen von Typkonstruktoren auf Typvariablen, die Typen beschreiben.
- *Muster* sind geschachtelte Anwendungen von Konstruktoren auf Variablen, die in Fallunterscheidungen verwendet werden.
- *Ausdrücke* sind geschachtelte Anwendungen von Funktionen und Konstruktoren auf Variablen.[7]

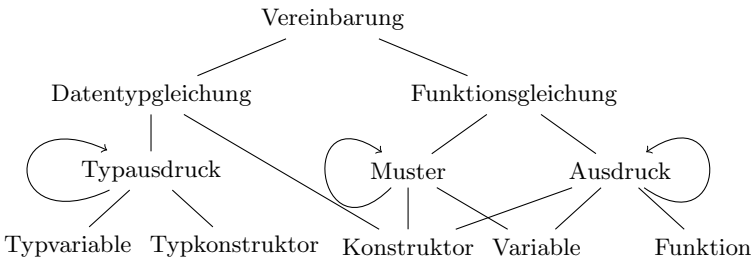


Abb. 1.2. Der Kern von Haskell

In den folgenden Kapiteln werden wir neue Varianten von Ausdrücken und viele vordefinierte Typkonstruktoren, Konstruktoren und Variablen kennen lernen, die es erlauben, Programme noch knapper und verständlicher zu schreiben. Dies wird *syntaktischer Zucker* genannt, der den Kern des funktionalen Programmierens zwar versüßt, ohne ihn aber zu verändern. Auch einige Vereinbarungen kommen noch dazu, mit denen das Typsystem um Typklassen und Typklasseninstanzen verfeinert wird.

1.8.2 Das Glasgower Haskell-System

Die Glasgower Haskell-Implementierung kommt mit einem Übersetzer `ghc`, der eigenständige Programme erzeugt. Wir werden meistens den Interpreter `ghci` ("Glasgow Haskell compiler interective") benutzen.

Aufruf des `ghci`

`ghci` ⟨*Dateiname*⟩ ruft die interaktive Variante des Übersetzers auf und lädt die Datei ⟨*Dateiname*⟩; dabei können die Endungen "`.hs`" oder "`.lhs`" weggelassen werden.

[7] Da auch Funktionen nur Werte sind, wird aber in der Definition von Haskell zwischen Variablen für Werte und Funktionen begrifflich nicht unterschieden.

`ghci --help` erklärt die Optionen des Aufrufs.
`ghci --print-libdir` gibt an, in welchem Verzeichnis das Programm `unlit` z ufinden ist.

Kommandozeilenbefehle im `ghci`

⟨*Ausdruck*⟩ wertet den HASKELL-Ausdruck aus.
`it` ist eine implizit definierte Variable, die den Wert des zuletzt ausgewerteten Ausrucks enthält
`:q[uit]` verlässt `ghci`
`:l[oad]` ⟨*Dateiname*⟩ lädt die angegebene HASKELL-Datei.
`:r[eload]` lädt die zuletzt geladene HASKELL-Datei erneut (z.B. nach Veränderungen im Editor)
`:i[nfo]` ⟨*Name*⟩ zeigt Informationen zu dem Typkonstruktor bzw. der Typklasse ⟨*Name*⟩ an. Der Modul, in dem der Name definiert wird, muss geladen sein. Der Typkonstruktor für Listen ist "`[]`", die für n-Tupel ist "`(`$,^{n-1}$`)`". Der Typkonstruktor für Funktiontypen ist `(→)`.
`:t[ype]` ⟨*Ausdruck*⟩ gibt den Typ des Ausdrucks an. Insbesondere können so die Typen aller sichtbaren "Variablennamen" (von Konstruktoren und Funktionen) angezeigt werden.
`:li[st]` ⟨*Name*⟩ gibt die Definition von ⟨*Name*⟩ in einem der geladenen Module an. Dies funktioniert nicht für Module aus der Bibliothek.

1.8.3 Kommentare

Kommentare sollen das Programm dokumentieren – nicht verzieren oder mit nutzlosem Text überschwemmen.

In Haskell gibt es zwei Arten von Kommentaren:

- *zeilenweise* Kommentare stehen zwischen von `--` und dem nächsten Zeilenende.
- mehrzeilige Kommentare werden zwischen `{-` und `-}` eingeschlossen. Das geht auch geschachtelt.

Ein Beispiel zeigt beide Formen:

```
{- Lehrveranstaltung Funktionales Programmieren (Praktische Informatik 3)
   (c) Berthold Hoffmann, Universität Bremen, hof@tzi.de
   Beispielprogramme für die Vorlesung am 28. Oktober 2009
-}

data List t = Empty | Cons t (List t) -- Listen polymorph

cat :: List t → List t → List t     -- Listenverkettung
cat  Empty      ys = ys
cat (Cons x xs) ys = Cons x (cat xs ys)
```

In Haskell gibt es noch eine alternative Sicht auf Programme: In Literate Haskell (normalerweise in Dateien mit der Endung `.lhs` statt `.hs` versehen) werden nicht die Kommentaranteile markiert, sondern umgekehrt der *Programmtext*. Alles andere ist Literatur. (*Ähm*, Kommentar.)

Hier gibt es wiederum zwei Möglichkeiten. Man kann alle Programmzeilen mit > in der ersten Spalte kennzeichnen:

```
Lehrveranstaltung Funktionales Programmieren (Praktische Informatik 3)
(c) Berthold Hoffmann, Universität Bremen, hof@tzi.de
Beispielprogramme für die Vorlesung am 28. Oktober 2009

> data List t = Empty | Cons t (List t) -- Listen polymorph

> cat :: List t → List t → List t     -- Listenverkettung
> cat  Empty      ys = ys
> cat (Cons x xs) ys = Cons x (cat xs ys)
```

(Die mit "`--`" eingeleiteten Zeilenkommentare gehen trotzdem noch.)

Oder man schreibt Programmstücke zwischen `\begin{code}` und `\end{code}`:

```
Lehrveranstaltung Funktionales Programmieren (Praktische Informatik 3)
(c) Berthold Hoffmann, Universität Bremen, hof@tzi.de
Beispielprogramme für die Vorlesung am 28. Oktober 2009

\begin{xcode}
data List t = Empty | Cons t (List t) -- Listen polymorph

cat :: List t → List t → List t     -- Listenverkettung
cat  Empty      ys = ys
cat (Cons x xs) ys = Cons x (cat xs ys)
```

Das ist besonders praktisch zur Benutzung mit LaTeX, weil man dann ein Programms in einem LaTeX setzen-Dokument beschreiben kann und *die gleiche Datei* auch mit `ghci` übersetzen kann. Dies kann auch für die Übungsaufgaben benutzt werden.

(Dies ist Fassung 2.1.4 von 4. Februar 2010.)

2

Standardtypen

In diesem Kapitel werden wir Datentypen wie Wahrheitswerte, Zahlen, Zeichen, Listen und Tupel mit den dazu gehörenden Funktionen betrachten, die in vielen funktionalen Sprachen als Standards vorgegeben sind. Konkret werden wir uns die Basis-Datentypen der Sprache HASKELL ansehen und lernen, wie wir sie zur Definition einfacherer Typen und Funktionen nutzen können.

2.1 Einfache Standardtypen

Die einfachen Standardtypen sind größtenteils aus der Mathematik oder aus anderen Programmiersprachen hinlänglich bekannt, und anhand der Sprache JAVA haben wir auch schon kennengelernt, wie sie in Programmiersprachen typischerweise realisiert werden – was bei Zahlen oft nur mit Einschränkungen geschieht.

In diesem Abschnitt werden wir uns aus daher darauf konzentrieren, die Unterschiede zwischen den Standardtypen von Haskell und ihren mathematischen "Idealen" bzw. ihrer Realisierung in JAVA zu beschreiben.

Die vollständige Standardbibliothek ist *online* unter

`http://www.haskell.org/onlinelibrary/`

zu finden; wir beschränken uns hier auf die Typen, die wir jetzt schon verstehen können.

2.1.1 Wertemengen der einfachen Standardtypen

Die für uns wichtigen einfachen Standardtypen, ihre Wertemengen, und die Form ihrer Literale sind in Tabelle 2.1 zusammengestellt.[1]

[1] *Literale* nennt man die Teile eines Programmtextes, mit denen Werte von einfachen Standardtypen im Programm bezeichnet werden.

Haskell-Typ	Wertemenge	Literale
`Bool`	Wahrheitswerte	`True` und `False`
`Char`	Zeichen	`'a'`, `'6'`, `';'`, `'\n'`, ...
`Int`	ganze Zahlen $[-2^{291}, .., 2^{29} - 1]$	`0`, `1`, ...,
`Integer`	ganze Zahlen $\mathbb{Z}$	`42`, ...
`Float`	Fließkommazahlen	0.0, 2E10 ...
`Double`	(32 bzw. 64 Bit)	`3.1415`, `1.2E-12` ...
`Rational`	rationale Zahlen $\mathbb{Q}$	`1\%2`, `-7\%5000000`

Tabelle 2.1. Standardtypen von Haskell, ihre Wertmengen und Literale

Im Typ `Char` ist der komplette UNICODE darstellbar; uns genügen die Zeichen, die wir auf der Tastatur finden, und ein paar nicht druckbare Zeichen wie *newline* (`'\n'`). Während `Int` die mit 32 Bits darstellbaren ganzen Zahlen umfasst, die heutzutage von der Hardware unterstützt werden, umfasst `Integer` beliebig große Zahlen. (Sie werden intern als Listen von `Int`-Zahlen dargestellt, also .) *Fließkommazahlen* der Typen `Float` und `Double` werden gemäß den Standards der IEEE realisiert, also die Teilmengen der (rationalen) reellen Zahlen, sie in 32 bzw. 64 Bits darstellbar sind. Der Typ `Rational` ist insofern nicht "Standard", als er explizit aus der Bibliothek `Ratio` importiert werden muss. Seine Werte kann man sich als teilerfremde Brüche von `Integer`-Zahlen mit positivem Nenner vorstellen.

2.1.2 Überladene Operationen auf einfachen Standardtypen

Für jeden einfachen Standardtyp gibt es die üblichen Operationen, die in den meisten Fällen auch mit den in der Mathematik und anderen Programmiersprachen üblichen Operatoren benannt werden.[2] Viele der Standardtypen haben ähnliche Operationen, die üblicherweise mit demselben Operator benannt werden:

- Alle einfachen Standardtypen besitzen einen Test auf *Gleichheit*, der mit dem Symbol "==" bezeichnet wird.
- Alle Zahlentypen besitzen eine Additionsoperation, die mit dem Symbol "+" bezeichnet wird.
- Die Fließkommatypen stellen die Konstante π unter dem Namen "`pi`" zur Verfügung.

Auch wenn diese Operationen gleich benannt sind und – abstrakt gesehen – das Gleiche tun, haben sie doch in den einzelnen Standardtypen verschiedene Implementierungen: Die Addition muss für alle Arten von Zahlen verschieden

[2] Unter den *Operationen* eines Datentypen verstehen wir die für ihn definierten Konstanten und Funktionen. Die Namen von Operationen werden *Operatoren* genannt. Dies sind entweder *Bezeichner*, für Konstanten und Funktionen, oder *Symbole*, für die normalerweise infix verwendeten binären Operatoren.

realisiert werden, und die Konstante `pi` wird in den Typen `Float` und `Double` verschieden genau dargestellt. In Programmiersprachen heißen Operatoren wie =, + und `pi` *überladen*, weil ihr Name verschiedene Definitionen hat, unter denen – abhängig vom gewünschten Typ – die jeweils "richtige" ausgewählt werden muss.

In Haskell werden überladene Operatoren realisiert, indem sie in einer *Typklasse* spezifiziert werden, und für jeden Typ, der diese Operatoren überlädt, eine konkrete Instanz des Operators definiert wird.

Die Standardtypen sind als Instanzen von meist mehreren Typklassen definiert, das heißt, sie definieren die in diesen Typklassen spezifizierten Operationen für den entsprechenden Standardtyp.

2.1.3 Vordefinierte Typklassen

Wir beschreiben kurz das allgemeine Konzept von Typklassen, gehen konkret aber nur auf diejenigen ein, die für die Standardtypen relevant sind. Allgemein werden wir Typklassen erst in Abschnitt 7.3.2 behandeln.

Eine *Typklasse* definiert abstrakte Eigenschaften, die für Typen gelten können, wenn sie diese Klasse *instantiieren*. Folgende abstrakten Eigenschaften kann eine Typklasse C haben:

- Typklassen, von denen C abhängt.
- Namen und Typen von Operatoren, die C enthält.
- Gleichungen, die einige der Operatoren mithilfe der anderen Operatoren von C definieren.

Ein Typ T ist *Instanz* einer Typklasse C, wenn er auch eine Instanz aller Klassen ist, von denen C abhängt, und die Operationen von C definiert. Dabei dürfen die Gleichungen von C als *default*-Definition benutzt werden.

Die Typklasse `Eq` definiert beispielsweise *Vergleichbarkeit* von Typen. Sie hängt von keiner anderen Klasse ab und spezifiziert die Signaturen und Gleichungen

```
class  Eq a  where
   (==), (/=)  :: (==), (/=) :: a → a → Bool
   x /= y      =  not (x == y)                  -- Defaults
   x == y      =  not (x /= y)
```

Die Typklasse `Num` definiert *numerische* Typen.

```
class  (Eq a, Show a) ⇒ Num a  where
   (+), (-), (*)  :: a → a → a
   negate         :: a → a
   abs, signum    :: a → a
   ...
```

Sie fordert Operationen, die für alle Zahlentypen definiert sind, und hängt von `Eq` ab, sowie von der Klasse `Show`, die Typen beschreibt, deren Werte sich mit einer Funktion `show`:: `a` → `String` als Zeichenketten darstellen lassen.

Jeder Typ, der eine Instanz von `Num` ist, ist also auch Instanz von `Eq` und `Show` und implementiert die Operationen aller drei Klassen.

Typklassen entsprechen also ziemlich genau *abstrakten Klassen* beim objektorientierten Programmieren. In der Terminologie des objektorientierten Programmierens würde man sagen: Die Standardtypen sind Klassen, die alle Operationen der Oberklassen implementieren, von denen sie erben.

Abbildung 2.1 zeigt die wichtigsten Typklassen, die im Zusammenhang mit den einfachen Standardtypen gebraucht werden, und die Standardtypen, die sie instantiieren.

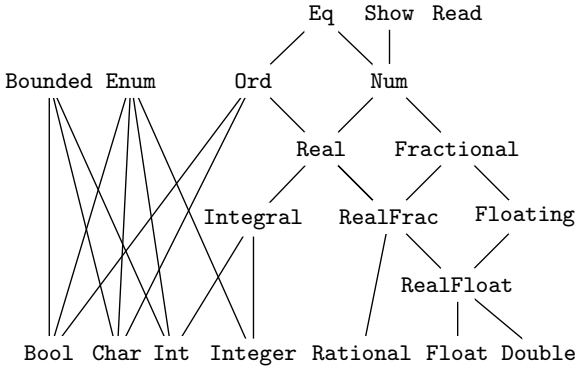


Abb. 2.1. Vordefinierte einfache Typen und Typklassen in Haskell

- `Show` und `Read` beschreiben Typen, deren Werte *druckbar* und *lesbar* sind, d.h., deren Werte sich mit den Funktionen `show` bzw. `read` als Zeichenketten darstellen bzw. aus Zeichenketten herauslesen lassen. Alle Standardtypen in Haskell sind druckbar und lesbar.
- Die Klasse `Eq` beschreibt *vergleichbare Typen*, die eine Gleichheit == und Ungleichheit /= implementieren. Nur eine dieser Operationen muss in jeder Instanz definiert werden, weil die andere mit der logischen Negation aus der einen abgeleitet werden kann.
- Die Klasse `Ord` beschreibt die *total geordneten Datentypen*, die über Vergleichsoperationen verfügen.
- Die Klasse `Num` ist die Mutter aller Zahlentypen. In ihr sind die für alle Zahlen gültigen Operationen *Addition*, *Subtraktion* und *Multiplikation* sowie Zahlenliterale wie `42` spezifiziert.

- `Fractional` spezifiziert die *teilbaren* Typen mit Division (/). Zu dieser Klasse gehören auch gebrochene Zahlen wie `47.11` und `13e-3`.[3]
- `Integral` spezifiziert die *ganzzahligen* Typen mit ganzzahliger Division und Rest (`div` und `mod`).
- Die Klasse `Floating` definiert *transzendente* Funktionen wie `exp`, `log`, `sin` und irrationale Konstanten wie `pi`.
- Die Klassen `Real` und `RealFrac` sind aus technischen Gründen in die Hierarchie der Typklassen eingebaut worden; in ihnen sind Typkonversions-Funktionen definiert.
- `RealFloat` umschreibt die typischen Eigenschaften von *Fließkommazahlen*, nämlich Operationen zum Zerlegen einer Zahl in Mantisse, Exponent usw.
- `Enum` ist die Klasse der *aufzählbaren* Typen, die eine Nachfolger- und Vorgänger-Funktion haben (`succ` und `pred`).[4]
- Die Klasse *Bounded* beschreibt die *beschränkten Typen*, die einen kleinsten und größten Wert besitzen.

Tabelle 2.2. Eigenschaften und wichtige Operationen vordefinierter Typklassen

Klasse	*Eigenschaft*	*wichtige Operationen*
`Show`	druckbar	`show`
`Read`	lesbar	`read`
`Eq`	vergleichbar	`== /=`
`Ord`	vollständig geordnet	`< <= >= >`
`Num`	numerisch	Zahlen `+ - *`
`Real`	–	`toRational`
`Fractional`	teilbar	gebrochene Zahlen `/`
`Integral`	ganzzahlig	`div mod`
`RealFrac`	–	`truncate round`
`Floating`	transzendent	`pi, exp log ...`
`RealFloat`	Fließkomma	`exponent`
`Enum`	aufzählbar	`succ pred toInt toEnum`
`Bounded`	beschränkt	`minBound maxBound`

[3] Allerdings wird auch die durch und durch ganze Zahl `2e+10` als gebrochen klassifiziert, das heißt, alles was Dezimalpunkt oder Exponent hat.

[4] Die Typen `Rational`, `Float` und `Double` sind nur insofern eine Instanz von `Enum`, als sie auch diese Funktionen implementieren: Sie addieren bzw. subtrahieren `1%1` bzw. `1.0`; damit wird aber in diesen Klassen nicht der *direkte* Nachfolger- oder Vorgänger einer Zahl bestimmt.

2.1.4 Eingeschränkte Polymorphie

Wir haben im Kapitel 1 bereits polymorphe Funktionen kennengelernt. So hat die Funktion `cat` den Typ `List a →List a →List a`. Das ist eine knappe Schreibweise für die Aussage

> "Für beliebige Typen `a` ist `cat` eine Funktion vom Typ
> `List a →List a →List a`."

Diese Art von Polymorphie wird *universell* genannt, weil die Elementtypen der Liste überhaupt nicht eingeschränkt sind.

Typklassen definieren eine etwas verfeinerte Art von Polymorphie. Das `ghci`-System gibt die Typen von `show` und + beispielsweise so an:

```
show :: (Show a) ⇒ a → String
(+) :: (Num a) ⇒ a → a → a
```

Das ist so zu lesen: "Für alle als Zeichenketten darstellbaren Typen `a` (die Instanzen der Klasse `Show` sind) ist `show` eine Funktion vom Typ `a → String`." Die Polymorphie ist nicht universell, sondern *eingeschränkt* auf die Typen, die Instanz einer Klasse sind. Hier ist also `show` nur für alle druckbaren Typen definiert. Dies nennt man eingeschränkte Polymorphie (*bounded polymorphism*). Analog ist die Addition + nur für alle numerischen Typen definiert. Auch Zahlenliterale sind eingeschränkt polymorph; `ghci` bestimmt ihre Typen so:

```
42 :: (Num a) ⇒ a
0.341 :: (Fractional a) ⇒ a
```

Erst die Anwendung spezifischerer Operationen macht den Typ von Zahlenausdrücken eindeutig oder mindestens eindeutiger:

```
4 + 2 :: (Num a) ⇒ a
4 + 2.3 :: (Fractional a) ⇒ a
4 / 2 :: (Fractional a) ⇒ a
4 `div` 2 :: (Integral a) ⇒ a
sin 2 :: (Floating a) ⇒ a
```

2.1.5 Wahrheitswerte

Der Typ der *Wahrheitswerte* mit den Operationen *Negation*, *Konjunktion*, *Disjunktion* usw. könnte in Haskell selbst so definiert werden, wäre er nicht schon vordefiniert:

```
data Bool = False | True
          deriving (Eq, Ord, Enum, Bounded, Read, Show)

not :: Bool→ Bool                -- Negation
not True  = False
not False = True
```

```
(&&), (||) :: Bool→ Bool→ Bool
True  && b =  b                    -- Konjunktion
False && _ =  False
True  || _ =  True                 -- Disjunktion
False || b =  b
```

Der Typ `Bool` hat also zwei Werte, die mit `False` und `True` bezeichnet werden, und ist eine Instanz der Typklassen `Eq`, `Ord`, `Enum`, `Bounded`, `Show` und `Read`. Seine Werte sind also vergleichbar, total geordnet, aufzählbar, beschränkt, druck- und lesbar. Da der Datentyp algebraisch definiert ist, müssen die Vergleichsoperationen, Ordnungsprädikate usw. nicht explizit definiert werden, sondern können kanonisch abgeleitet werden.[5] Mit diesen abgeleiteten Operationen stehen weitere logische Operationen (mit mehr oder weniger sinnigen Bezeichnungen zur Verfügung:

- Die Gleichheit == implementiert die logische Äquivalenz "⇔".
- Die Ungleichheit /= implementiert das exklusive Oder "$\dot{\vee}$".
- Das Prädikat $\leq$ implementiert die logische Implikation "⇒".

Die Negation `not` ist eine Funktion, während Konjunktion und Disjunktion als zweistellige *Operationen* definiert sind, die den mit Sonderzeichenfolgen `&&` und `||` bezeichnet sind. Operationen schreibt man in Ausdrücken zwischen ihre Operanden; in der Signaturdefinition muss man sie in Klammern einschließen.[6] Der Unterstrich `\_` ist eine "anonyme Variable", die auf der linken Seite von Funktionsdefinitionen immer dann verwendet wird, wenn der Wert der Variable auf der rechten Seite (oder für die vorangestellte Bedingung) nicht benötigt wird. (Tatsächlich sollte man alle Variablen, die nur auf der linken Seite benötigt werden, anonymisieren, damit sie auf der rechten Seite nicht "versehentlich" benutzt werden.) Man beachte, dass Konjunktion und Disjunktion ihren ersten Parameter immer auswerten, den zweiten aber nicht. (Wie in Java.) Die Ausdrücke `n==0 || z/n` und `n/=0 && z/n` führen also nie zu einem Auswertungs-Fehler, weil die Division nur ausgewertet wird, wenn der Nenner ungleich 0 ist.

`Bool` ist *semantisch eingebaut*, denn dieser Datentyp gebraucht wird, um die Bedeutung bestimmter Konzepte in Haskell zu definieren.

Beispiel 2.1 (Das Nim-Spiel). Das *Nim-Spiel* ist sehr einfach: Zwei Spieler nehmen abwechselnd 1–3 Hölzchen. *Verloren* hat derjenige, der das letzte Hölzchen nimmt.

Wir wollen ein Programm schreiben, das entscheidet, ob ein Zug *gewinnt*. Die Eingabe ist die Gesamtanzahl der Hölzchen und ein Zug (eine Anzahl von

[5] Mehr dazu beim allgemeinen Thema *algebraische Datentypen* in Kapitel 5.

[6] Umgekehrt kann man auch jeden Bezeichner einer zweistelligen Funktion in Infixschreibweise verwenden, wenn man ihn in "Rückwärts-Anführungsstriche ‘ einschließt. So ist `(CONS x l) ‘cat‘ l’` ein Ausdruck mit der Verkettungsfunktion aus Kapitel 1.

1–3 genommenen Hölzchen). Die Ausgabe soll die Frage "Gewonnen?" mit *ja* oder *nein* beantworten. Daraus ergibt sich sie Signatur

```
type Move= Int
winningMove :: Int→ Move→ Bool
```

(Dabei ist "`type Move =Int`" eine *Typsynonym-Definition*, die `Move` als einen neuen Namen für den Standardtyp `Int` einführt.) Ein Zug führt zum Gewinn, wenn er legal ist und der Gegner nicht mehr gewinnen kann:

```
winningMove total move =
  legalMove total move && mustLose (total-move)
```

So können wir überprüfen, ob ein Zug legal ist:[7]

```
legalMove :: Int→ Int→ Bool
legalMove total m = (m≤ total) && (1≤ m) && (m≤ 3)
```

Der Gegner kann nicht gewinnen, wenn *(i)* nur noch ein Hölzchen übrig ist, oder*(ii)* wir bei jedem möglichen Zug von ihm gewinnen können.

```
mustLose :: Int→ Bool
mustLose n
  | n == 1    = True
  | otherwise = canWin n 1 &&
                canWin n 2 &&
                canWin n 3
```

Wir gewinnen, wenn es einen legalen Zug gibt, der gewinnt:

```
canWin :: Int→ Int→ Bool
canWin total move =
  winningMove (total- move) 1 ||
  winningMove (total- move) 2 ||
  winningMove (total- move) 3
```

Genaueres Hinschauen macht klar, dass die Aufrufe von `canWin` in `mustLose` unnützerweise überprüfen, ob die Züge legal sind.

Gibt es eine einfache Antwort auf die Frage: *Wann gewinnen wir?* Vermutlich immer, wenn die Anzahl der Hölzchen ungleich $4n + 1$ ist.

Eine interaktive Fassung dieses Spiels werden wir in Abschnitt 9.4 entwickeln.

2.1.6 Zahlen

In Programmiersprachen unterliegen Zahlendarstellungen meist gewissen Einschränkungen:

[7] Aufmerksame LeserInnen werden merken, dass die Funktion für negative Eingaben divergieren kann. Es sollte also einmal zu Beginn sichergestellt werden, dass `total` und `move` nicht negativ sind.

- Die ganzen Zahlen $\mathbb{Z}$ werden nur in einem eingeschränkten Bereich wie $-2^{31}, \ldots, 2^{31} - 1$ dargestellt.
- die reellen Zahlen $\mathbb{R}$ werden durch Fließpunktzahlen dargestellt, die eigentlich nur eine Teilmenge der rationalen Zahlen darstellen können.

In Haskell sind fünf Arten von Zahlen vordefiniert, vgl. Tabelle 2.1:

- `Int` ist der Typ der *beschränkten ganzen Zahlen*, der dem gleichnamigen JAVA-Typ entspricht.
- Die *unbeschränkten ganzen Zahlen* `Integer` werden als Listen von `Int`-Zahlen dargestellt, praktisch also als "Zahlen" über Ziffern zur Basis 2^{32}.
- `Real` und `Double` sind 32-Bit- bzw. 64-Bit-*Fließkommazahlen* nach IEEE-Standards 754 und 854 mit den bekannten Größenbeschränkungen und Rundungsfehlern.
- `Rational` ist der Typ der *unbeschränkten rationalen Zahlen* $\mathbb{Q}$, die man sich als Paare teilerfremder `Integer`-Zahlen mit positivem Nenner vorstellen kann. Zur Konstruktion rationaler Zahlen gibt es eine Umwandlungsfunktion und einen Infixoperator `(%)`, der aus zwei granzen Zahlen z und n eine rationale Zahl mit Zähler z und Nenner n konstruiert:[8]

  ```
  toRational :: (Real a) ⇒ a → Rational
  (%) :: Integer → Integer → Rational
  ```

In Haskell hat man beim Rechnen mit Zahlen also die Wahl: Entweder gewohnte Effizienz und (gewohnte) Beschränkung des Wertebereiches bei den Typen `Int`, `Float` und `Double`, oder (langsam) wachsender Aufwand für beliebig genaue ganze oder gebrochene Zahlen der Typen `Integer` und `Rational`.

Bei der Benutzung der Operationen auf Zahlen ist Einiges zu beachten:

- Das einstellige Minus (Vorzeichenwechsel) kann wegen der Klammerkonventionen leicht mit dem zweistelligen Minus (Subtraktion) verwechselt werden und sollte deshalb in Zweifelsfällen geklammert werden, z. B. `abs (-34)`.
- Für die ganzzahligen Typen gilt `(x 'div` y`)*y +x 'mod` y `==x`.
- Polymorphe Typen von Zahlenliteralen können durch die Konversionsfunktion angepasst werden:

  ```
  fromIntegral :: (Integral a, Num b) ⇒ a → b
  ```

- Gebrochene Zahlen können abgeschnitten oder gerundet in ganze Zahlen umgewandelt werden:

  ```
  truncate, round :: (RealFrac a, Integral b) ⇒ a → b
  ```

[8] Trotzdem ist `(%)` kein Wertkonstruktor.

2.2 Zusammengesetzte Standardtypen

In diesem Abschnitt werden wir "typisch funktionale" zusammengesetzte Datentypen kennen lernen: Tupel und Listen. Zeichenketten sind als Listen von Einzelzeichen definiert.

2.2.1 Tupel

Tupel komponieren eine feste Anzahl von $n \geqslant 2$ Werten unterschiedlichen Typs. Tupel sind in Haskell vordefiniert und mit einer Schreibweise versehen, wie sie in der Mathematik benutzt wird.

```
data (a,b)   = (a,b)   deriving (Eq, Ord, Bounded, Show, Read)
data (a,b,c) = (a,b,c) deriving (Eq, Ord, Bounded, Show, Read)
...

pair :: (Char,Int)
pair =  ('F',6)
...
```

Dies ist kein legales Haskell, weil die Typkonstruktoren (auf der linken Seite) und die Wertkonstruktoren (auf der echten Seite) "um die Komponenten herum" statt vor sie geschrieben werden. Man beachte, dass die Schreibweise "$(\ldots,\ldots)$" auf der *rechten* Seite der Definition einen *Wertkonstruktor* definiert, der in der Definition von `pair` benutzt wird, um aus Werten der Typen `Char` und `Int` einen Wert der Paarmenge `(Char, Int)` zu konstruieren, während die gleiche Schreibweise auf der *linken* Seite der Definition den *Typkonstruktor* bezeichnet, der aus den Typen `a` und `b` die Paarmenge (`a`, `b`) selbst definiert, wie er in der Signatur `pair` :: `(Char,Int)` benutzt wird.

2.2.2 Listen

Listen komponieren eine variable Anzahl von Werten des gleichen Typs. Sie entsprechen dem parametrisierten Typ *List* aus Kapitel 1:

```
List a = Empty | Cons a (List a)
```

Listen haben in Haskell ebenfalls eine besondere Schreibweise.

```
data [a] = [] | a : [a] deriving (Eq, Ord, Show, Read)

fibs :: [Integer]
fibs = 0: 1: 1: 2: 3: 5: 8: 13: 21: 34: []
```

Auch dies ist wegen der komischen Schreibweise für den Typkonstruktor kein legales Haskell. Die leere Liste `Empty` wird als `[]` geschrieben, und dem Konstruktor *Cons* entspricht der Infix-Konstruktor `(:)`. Außerdem gibt es folgende abkürzende Schreibweise für Listenwerte:

```
fibs = [0, 1, 1, 2, 3, 5, 8, 13, 21, 34]
```

Die Schreibweise von Listen suggeriert eine Ähnlichkeit zu *Mengen*, die auch tatsächlich gegeben ist. Mengen enthalten auch eine variable Anzahl von Elementen des gleichen Typs. Trotzdem muss man die Unterschiede zwischen beiden beachten:

- Die Reihenfolge der Elemente ist in Listen relevant, aber in Mengen nicht.
- Alle Elemente einer Menge sind verschieden, die einer Listen nicht unbedingt.

Beispiel 2.2 (Einkaufskorb, Polygon). Viele Daten lassen sich als Kombination von Tupeln und Listen darstellen.

Ein *Einkaufskorb* enthält eine Menge von Gegenständen, die jeweils einen Namen und einen Preis haben.

```
type Item   = (String, Int)
type Basket = [Item]
```

Punkte sind Koordinatenpaare, Geraden sind durch zwei Punkte festgelegt, und Polygone durch eine Liste von Punkten.

```
type Point   = (Int, Int)
type Line    = (Point, Point)
type Polygon = [Point]
```

Funktionen über Tupeln und Listen werden durch Gleichungen mit Mustervergleich (*pattern matching*) beschrieben. Für Tupelwerte gibt es jeweils nur ein passendes Muster:[9]

```
add :: Point→ Point→ Point
add (x, y) (dx, dy) = (x+ dx, y+ dy)
```

Bei Funktionen auf Listen müssen zwei Fälle unterschieden werden:

- Eine Liste ist entweder *leer*
- oder sie besteht aus einem *Kopf* und einem *Rest*

Zum Beispiel beim Aufsummieren der Elemente einer Liste:

```
sum :: (Num a) ⇒ [a]→ a
sum []     = 0                -- leere Liste
sum (x:xs) = x + sum xs  -- nicht-leere Liste
```

Listenmuster dürfen auch komplizierter sein, wie hier beim Prüfen der Ordnung einer Liste:

```
ordered :: (Ord a) ⇒ [a]→ Bool
ordered []         = True                        -- leer
ordered [_]        = True                        -- einelementig
ordered (x: y: xs) = x ≤ y && ordered (y: xs) -- mehrelementig
```

[9] Solche Muster werden unwiderlegbar *irrefutible* genannt.

Je komplizierter die Muster in einer Funktionsdefinition sind, um so sorgfältiger muss man prüfen, ob auch wirklich alle möglichen Formen von Listen behandelt werden. Sonst wird die Funktion für die nicht berücksichtigten Fälle *undefiniert*.

Manche Funktionen sollen aber partiell sein, wie die vordefinierten Selektorfunktionen `head` und `tail` auf (nicht leeren) Listen:

```
head        :: [a]→ a
head (h: _) =  h
tail        :: [a] → [a]
tail (_: t) =  t
```

Wird eine dieser Funktionen auf eine leere Liste angewendet, tritt ein Laufzeitfehler auf, und das geht auch nicht anders. Man könnte nur selber eine hoffentlich aufschlussreichere Fehlermeldung ausgeben lassen, wenn man eine Gleichung hinzufügt wie

```
...
head []     =  error "head␣of␣[]"
...
tail []     =  error "tail␣of␣[]"
```

Zurück zu den anfangs eingeführten Beispielen für Typen. Der Gesamtpreis für einen Einkaufskorb kann so berechnet werden:

```
total :: Basket → Int
total [] = 0
total ((name, price):rest) = price + total rest
```

Die Verschiebung eines Polygons kann so definiert werden:

```
move :: Polygon→ Point→ Polygon
move [] p                  = []
move ((x, y):ps) (dx, dy) = (x+ dx, y+ dy):
                              (move ps (dx, dy))
```

Als Variante von Tupeln sind in Haskell auch *Verbunde* (*records*) vordefiniert, deren Komponenten mit Namen selektiert werden können. (Genauso wie Attribute einer Klasse in Java.) Beim Umgang mit diesen Typen sind in Haskell einige Regeln zu beachten, mit denen wir uns – wenn überhaupt – erst später belasten wollen. Ähnlich ist es mit *Feldern* (*array*): auch sie sind in Haskell vordefiniert, aber nicht ganz einfach zu benutzen, und im übrigen auch nicht so effizient, weil sie nicht selektiv überschrieben werden dürfen.

2.2.3 Zeichen und Zeichenketten

Der Datentyp `Char` definiert *Einzelzeichen*, die laut Sprachstandard als *Uni-Code* definiert sind. Der Typ `Char` gehört zu den Typklassen `Eq`, `Ord`, `Enum` und `Bounded`. (Weshalb wohl nicht zu `Show` und `Read`?)

Nützliche Funktionen auf Zeichen sind z. B.:

```
ord :: Char → Int
chr :: Int → Char
toLower :: Char→ Char
toUpper :: Char→ Char
isDigit :: Char→ Bool
isAlpha :: Char→ Bool
```

Zeichenketten sind als Listen von Zeichen vordefiniert:

```
type String = [Char]
```

Für die Angabe von konstanten Zeichenketten gibt es folgenden *syntaktischen Zucker*:

```
['y','o','h','o'] == "yoho"
```

Damit kann die Häufigkeit eines Zeichens in einer Zeichenkette zum Beispiel so bestimmt werden:

```
frequency :: Char→ String→ Int
frequency c []     = 0
frequency c (x:xs) = if (c== x) then 1+ frequency c xs
                     else frequency c xs
```

Funktionen über Zeichenketten werden also genau so geschrieben wie solche über Listen. Und alle für Listen vordefinierten Funktionen lassen sich natürlich ohnehin für `String` verwenden.

Beispiel 2.3 (Palindrome). Palindrome sind Wörter, die vorwärts und rückwärts gelesen gleich sind, wie *Otto* oder *Reliefpfeiler*. Die Palindromeigenschaft kann als eine Funktion `palindrom` ::`String`→`Bool` definiert werden.

Die Idee ist einfach rekursiv zu formulieren: Der erste Buchstabe eines Palindrom gleicht dem letzten, und der Rest ist auch ein Palindrom; Wörter der Länge 0 oder 1 sind immer Palindrome. Dazu können wir die vordefinierten Hilfsfunktionen `last: String`→`Char`, `init: [a]`→`[a]` benutzen, die (dual zu `head` und `tail`, eine Liste in ihr letzte Element und den den Teil davor aufteilen:

```
palindrom :: String→ Bool
palindrom []     = True
palindrom [_]    = True
palindrom (x:xs) = x == last xs && palindrom (init xs)
```

Diese Definition unterscheidet Groß- und Kleinbuchstaben. Besser ist:

```
palindrom :: String→ Bool
palindrom []     = True
palindrom [_]    = True
palindrom (x:xs) = toLower x == toLower (last xs)
                   && palindrom (init xs)
```

Die Funktion `toLower ::Char →Char` wandelt Großbuchstaben in kleine um und lässt alle anderen Zeichen gleich.

Außerdem sollten andere Zeichen als Buchstaben nicht berücksichtigt werden. Dann könnte man auch ganze *Sätze* wie "*Ein Neger mit Gazelle zagt im Regen nie!*" als Palindrome erkennen. Dies bleibt der LeserIn überlassen.

2.3 Fragen

1. Die Operationen Addition und Multiplikation sind auf den ganzen und reellen Zahlen assoziativ und haben 0 bzw. 1 als neutrale Elemente. Überlegen Sie Beispiele dafür, dass diese Eigenschaften für die Datentypen `Int` oder `Float` nicht immer gelten.
2. Definieren Sie die Funktionen `init ::[a] →[a]` und `last ::[a] →a` aus Beispiel 2.3.
   Wie groß ist der Zeitaufwand und Platzbedarf dieser Funktionen?
3. Geben Sie eine Gleichung an, die die Funktionen `head` und `tail` auf Seite 34 beschreibt. Finden Sie eine analoge Gleichung für `init` und `last`.
4. Die in Übungsblatt 1 zu definierende Funktion *`rev`* `::List a →List a` kehrt eine Liste um.
   Geben Sie Zusammenhänge zwischen *`rev`*, `head` und `tail` auf der einen Seite, und `init` und `last` auf der anderen Seite an.
5. Geben Sie einen Zusammenhang zwischen *`palindrom`* und *`rev`* an.

2.4 Haskells Eck

In diesem Abschnitt wird Einiges beschrieben, was hoffentlich das allgemeinere Verständnis der Sprache fördert.

2.4.1 Lexikalisches

Jedes Haskell-Programm besteht aus einer Folge von *Symbolen* oder *Lexemen*, die sich verschieden aus Zeichen zusammensetzen. Die wichtigsten Lexeme sind:

- *Bezeichner* beginnen mit einem Buchstaben und können weitere Buchstaben, Ziffern, Unterstriche "`_`" und *Striche* "`'`" enthalten. (Allgemeiner als in der Mathematik üblich dürfen die Striche auch in der Mitte eines Bezeichners auftreten, etwa "`a''b'c`".)
- *Konstruktorbezeichner* beginnen mit einem Großbuchstaben. Sie werden als Namen für *Wertkonstruktoren*, *Typkonstruktoren*, *Typklassen* und *Modulnamen* verwendet.

- *Variablenbezeichner* beginnen mit einem Kleinbuchstaben. Sie werden als Namen für Konstanten und Funktionen sowie als *Variablen* (Platzhalter) für Typen in Typdefinitionen und für Werte und Funktionen in Funktionsdefinitionen oder Ausdrücken verwendet.
- Folgende Variablenbezeichner sind als Schlüsselwörter reserviert:
  ```
  case   class    data   default deriving do
  else   if       import in      infix    infixl
  infixr instance let    module  newtype  of
  then   type     where  _
  ```
  Sie können nicht als normale Variablenbezeichner verwendet werden.
- *Operatorsymbole* bestehen aus Sonderzeichen `!#$%&*+./<=>?@\^|-~`. Die folgenden Operatorsymbole sind reserviert:
  `..` `:` `::` `=` `\` `|` $\leftarrow$ $\rightarrow$ `@` `\~` $\Rightarrow$
- *Trennzeichen* wie `(`, `)`, `{`, `!` und `;` dienen als Satzzeichen im Programm.
- *Numerische Literale* bestehen a us Ziffern und können einen Dezimalpunkt und einen mit "`e`" eingeleiteten Exponentteil haben.
- Zeichen- und Zeichenketten-Literale werden in einzelne Hochkommata "`'`" bzw. doppelte Anführungszeichen "`"`" eingeschlossen. Dazu gibt es noch Möglichkeiten, Zeichen als oktale oder hexadezimale Zahlen anzugeben.
- *Layoutzeichen* wie das Leerzeichen, das Tabulatorzeichen, das Zeilenende- oder Wagenrücklaufzeichen dienen dazu, das Ende eines Lexems anzuzeigen. So ist *`meineFunktion`* *ein* Bezeichner, während *`meine Funktion`* zwei nebeneinander stehende Bezeichner sind. Außerdem dient Layout dazu, das Programm verständlich und übersichtlich hinzuschreiben.
- *Kommentare* kennzeichnen Programmtext, der überlesen werden soll. Zeilenkommentare können zwischen '`--`' und das Zeilenende geschrieben werden; längere Kommentare sind von der Form "*`{- ... -}`*"; sie dürfen auch geschachtelt werden.

2.4.2 Layout

in einem Haskell-Programm können an vielen Stellen Layoutzeichen eingefügt werden, aber im Gegensatz zu anderen Sprachen (auch JAVA) ist das Layout nicht immer egal.

In Haskell gilt die Abseitsregel (*offside rule*). Für eine Funktionsdefinition

$$\texttt{f}\,\texttt{x}_1\,\texttt{x}_2 \ldots \texttt{x}_n = \texttt{E}$$

gilt: *Alles, was gegenüber* `f` *in den folgenden Zeilen eingerück ist, gehört noch zur Definition von* `f`. Erst mit der nächsten Zeile, die in der gleichen Spalte wie `f` beginnt, fängt eine neue Definition an.

```
f x = hier faengts an
  und hier gehts weiter
      immer weiter
g y z = und hier faengt was neues an
```

Das gilt auch bei verschachtelten Definitionen.

Diese Regel ist vielleicht zunächst gewöhnungsbedürftig. Sie ist trotzdem sinnvoll, weil man so durch Einrückungen gruppieren und sequenzialisieren kann, wozu in anderen Sprachen wie JAVA Klammerungen mit `{` und `}` und Trennzeichen `;` benötigt werden. (In Haskell können diese Zeichen in genau der gleichen Funktion benutzt werden, wenn man einmal die Abseitsregel verletzen muss oder möchte; wir werden das aber kaum brauchen.)

2.4.3 Operator oder Funktion?

Operatoren sind Namen aus Sonderzeichen. Operatoren werden *infix* geschrieben, wie beispielsweise `x && y`. Ansonsten sind es normale (zweistellige) Funktionen.

Auch andere Funktionen, deren Namen Bezeichner sind, können infix benutzt werden. Dazu werden ihre Namen in Apostrophen (*back quotes*) eingeschlossen.

```
x `mod` y
```

Umgekehrt können auch Operatoren in Präfixschreibweise benutzt werden. Dazu müssen sie geklammert werden:

```
condition = (&&) True ((||) False True)
```

Präfix-Applikation von Funktionen bindet stärker als Infix: Ein Ausdruck "`(f x) +(f y)`" kann ohne Klammern als "`f x +f y`" geschrieben werden.

3

Listen

In diesem Kapitel geht es um *Listen*, den zusammengesetzten Datentyp funktionaler Sprachen schlechthin. Schon in Kapitel 1 hatten wir gelernt, dass Listen ein vordefinierter Typ sind. In diesem Kapitel werden wir die wichtigsten vordefinierten Funktionen auf Listen und weitere spezielle Schreibweisen für Listen kennenlernen, an denen wir typische Arten von Rekursion beleuchten werden, die bei Funktionen auf Listen gebraucht werden.

3.1 Vordefinierte Listenfunktionen

Listen sind der wichtigste vordefinierte Datentyp in funktionalen Sprachen. Deshalb gibt es für diesen Typ in HASKELL ein besonders reichhaltiges Angebot an vordefinierten Funktionen (die sonst aber auch sehr einfach selber definiert werden könnten.) Wir werden einige Eigenschaften dieser Funktionen und Zusammenhänge zwischen verschiedenen Funktionen mit Gleichungen beschreiben. Solche Gleichungen sind nützlich für Korrektheitsbeweise über Funktionen, die diese vordefinierten Funktionen benutzen.

In Kapitel 2 hatten wir gesehen, dass Listen in HASKELL mit folgender Notation vordefiniert sind:

$$\textbf{data}\ [\alpha] = [\,] \parallel \alpha : [\alpha]\ \textbf{deriving}\ (Eq, Ord, Show, Read, \dots)$$

Die *Konstruktoren* sind die leere Liste `[]` und die Konstruktor-Operation `(:)`:

$$[] :: [\alpha] \qquad (:) :: \alpha \to [\alpha] \to [\alpha]$$

Das Prädikat `null` überprüft, ob eine Liste leer ist.

```
null    :: [a] → Bool
null [] =  True
null _  =  False
```

Die *Selektoren head* und *tail* greifen auf die Komponenten einer nicht-leeren Liste zu.

```
head, last  :: [a] → a
head (x:_)  =  x
head []     =  error "Prelude.head:␣empty␣list"

tail, init  :: [a] → [a]
tail (_:xs) =  xs
tail []     =  error "Prelude.tail:␣empty␣list"
```

Diese Selektoren erfüllen die Gleichung

$$\forall \ell \in [\alpha] : \mathit{null}\ \ell \vee \mathit{head}\ \ell : \mathit{tail}\ \ell = \ell$$

Dual zu den Standard-Selektoren *head* und *tail* selektiert *last* das letzte Element und *init* alle übrigen Elemente einer Liste:

```
last [x]    =  x
last (_:xs) =  last xs
last []     =  error "Prelude.last:␣empty␣list"

init [x]    =  []
init (x:xs) =  x : init xs
init []     =  error "Prelude.init:␣empty␣list"
```

Die Selektoren erfüllen die Gleichung

$$\forall \ell \in [\alpha] : \mathit{null}\ \ell \vee \mathit{init}\ \ell +\!\!+ [\mathit{last}\ \ell] = \ell$$

Die *Verkettungsoperation* für Listen kennen wir schon:

```
(++) :: [a]→ [a]→ [a]
[] ++ ys      = ys
(x:xs) ++ ys = x:(xs ++ ys)
```

Sie erfüllt die Gleichungen:

$$\forall \ell_1, \ell_2, \ell_3 \in [\alpha] : \ell_1 +\!\!+ (\ell_2 +\!\!+ \ell_3) = (\ell_1 +\!\!+ \ell_2) +\!\!+ \ell_3$$
$$\forall \ell \in [\alpha] : \ell +\!\!+ [] = [] +\!\!+ \ell_1 = \ell$$

Dass heißt, `(++)` ist assoziativ und hat die leere Liste als neutrales Element.

Die Funktion `length` bestimmt die Länge einer Liste:

```
length :: [a] → Int
length []     = 0
length (x:xs) = 1+ length xs
```

Mit der Funktion `(++)` hängt sie so zusammen:

$$\forall \ell_1, \ell_2 \in [\alpha] : \mathit{length}(\ell_1 +\!\!+ \ell_2) = \mathit{length}\,\ell_1 + \mathit{length}\,\ell_2$$

Die Funktion `reverse` kehrt eine Liste um:

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

Diese Gleichungen sollen nur erklären, *was* `reverse` tut. Tatsächlich ist diese Funktion effizienter implementiert. Diese Funktion erfüllt die Gleichungen

$$\begin{aligned}\forall \ell \in [\alpha] : \mathit{reverse}(\mathit{reverse}\,\ell) &= \ell \\ \mathit{head}(\mathit{reverse}\,\ell) &= \mathit{last}\,\ell \\ \mathit{last}(\mathit{reverse}\,\ell) &= \mathit{head}\,\ell\end{aligned}$$

Es gibt auch eine Operation zum "direkten Indizieren" von Listenelementen:

```
(!!)                 :: [a] → Int → a
xs     !! n | n < 0 =  error "Prelude.!!: negative index"
[]     !! _         =  error "Prelude.!!: index too large"
(x:_)  !! 0         =  x
(_:xs) !! n         =  xs !! (n-1)
```

Aus der Definition sieht man allerdings, dass dies – anders bei den Feldern in Java – nicht mit konstantem Zeitaufwand geht.

Mit *length* kann man sicher Listenelemente indizieren, weil $\ell \,!!\, i$ nur für Indizes $0 \leqslant i < \mathit{length}\,\ell - 1$ definiert ist. Im allgemeinen ist Listenindizierung eine unsichere Operation – genau wie die Feldindizierung. Deshalb wird sie selten verwendet.

Für Listen mit vergleichbaren Elementen gibt es Abfragen auf (nicht-) Enthaltensein.

```
elem, notElem :: (Eq a) ⇒ a → [a] → Bool
elem x []     =  False
elem x (y:ys) =  x==y || elem x ys
notElem x xs  =  not (elem x xs)
```

Weitere oft benutzte Funktionen auf Listen werden wir in Abschnitt 3.3 kennenlernen, wo sie als Beispiele für verschiedene Arten von Rekursion herhalten müssen.

3.2 Listenumschreibungen

Aus der Mathematik kennen wir die *prädikative Definition* von Mengen:

$$M' = \{x \mid x \in M, P(x)\}$$

Hiermit wird die Menge M' als die Teilmenge all derjenigen Elemente x aus einer Basismenge M definiert, die das *Prädikat* P erfüllen (siehe [EMC+01, S. 9ff]). Solche Definitionen können erweitert werden zu

$$M'' = \{F(x) \mid x \in M, P(x)\}$$

Hierbei ist F eine Funktion, mit Hilfe derer die Elemente von M' in die von M'' *transformiert* werden.

Diese Art der Mengenbeschreibungen ist zuerst in MIRANDA [Tur86] auf Listen übertragen und später in HASKELL übernommen worden. Diese so genannten *Listenumschreibungen* (engl. *list comprehensions*) haben die Form

$$[T \mid x \leftarrow L, P]$$

Hierbei sind L, P und T Ausdrücke folgender Art:

- Der Ausdruck L *generiert* eine Basisliste ℓ.
- Der Boole'sche Ausdruck P *testet* jedes Element x von ℓ.
- Der Ausdruck T *transformiert* jedes Element x von ℓ, das den Ausdruck P wahr macht.

An dieser Stelle sei gerade wegen der Ähnlichkeit zur Mengenschreibweise noch einmal darauf hingewiesen, dass Listen wie `str` und `str'` Elemente mehrmals enthalten können, und dass ihre Reihenfolge relevant ist.

Beispiel 3.1 (Nochmal Palindrome). In Beispiel 2.3 hatten wir angemerkt, dass aus der zu untersuchenden Zeichenkette `str` die Buchstaben herausgesucht werden sollten, und dass die wiederum in Kleinbuchstaben umgewandelt werden sollten. Dies kann mit Hilfe der vordefinierten Funktionen `isAlpha ::Char →Bool` und `toLower ::Char →Char` und einer Listenumschreibung elegant erledigt werden:

```
palindrom'' :: String → Bool
palindrom'' w = w' == reverse w'
                where w' = [ toLower c | c← w, isAlpha c]
```

Die allgemeine Form der Listenumschreibung sieht so aus:

$$[T \mid m_1 \leftarrow L_1, \ldots, m_k \leftarrow L_k, P_1, \ldots, P_n]$$

Die Ausdrücke L_1 bis L_k generieren mehrere Listen ℓ_1 bis ℓ_k. Zur Auswahl aus diesen Listen können außer Variablen (wie im vorigen Beispiel) Muster m_1 bis m_k benutzt werden, die auf die Elemente von ℓ_1 bis ℓ_k passen. Schließlich können mehrere Ausdrücke P_1 bis P_n benutzt werden, um Elemente aus den Listen ℓ_1 bis ℓ_k auszuwählen.

Ein Beispiel für ein Muster (von Tupeln) wird in der folgenden Funktion benutzt:

```
pairwiseSum    ::    Num a ⇒ [(a, a)]→ [a]
pairwiseSum pairs =  [ x+y | (x, y) ← pairs ]
```

Mit dem Generator `(x, y) ←pairs` wird ein Tupel aus der Liste `pairs` ausgewählt und durch das Tupel-Muster gleich in seine beiden Komponenten `x` und `y` zerlegt.

Listenumschreibungen mit zwei generierenden Ausdrücken finden wir in dieser Definition:

```
chessBoard :: [(Int,Int)]
chessBoard = [(x,y) | x ← [1..8], y ← [1..8] ]
```

Der Ausdruck `[1..8]` zählt die Liste `[1,2,3,4,5,6,7,8]` auf; diese Schreibweise ist eine Abkürzung für den Aufruf `enumFromTo 1 8` einer Funktion, die für alle aufzählbaren Typen definiert ist, d.h. für alle Instanzen der Typklasse `Enum`. Eine Variation dieser Aufzählung, `[2,4..8]`, zählt die Liste `[2,4,6,8]` auf.

Beispiel 3.2 (Leihkartei einer Bücherei). Zur Modellierung einer Leihkartei werden folgende *Typen* gebraucht:

- Ausleihende Personen
- Bücher
- Die Leihkartei ist eine Datenbank, bestehend aus ausgeliehenen Bücher und Ausleihen

```
type Person = String
type Book   = String
type DBase  = [(Person, Book)]
```

Wir wollen drei Operationen auf der Datenbank realisieren:

- Buch *ausleihen*.
- Buch *zurückgeben*.
- Liste aller Bücher *aufstellen*, die eine Person ausgeliehen hat.

```
makeLoan :: DBase → Person → Book → DBase
makeLoan dBase pers bk = (pers,bk) : dBase

returnLoan :: DBase → Person → Book → DBase
returnLoan dBase pers bk
    = [ loan | loan ← dBase, loan /= (pers,bk) ]

books :: DBase → Person → [Book]
books db who = [ book | (pers,book)← db, pers== who ]
```

In diesem Beispiel ist `makeLoan` der Konstruktor, `returnLoan` ein Selektor und `books` eine Inspektionsfunktion.

3.3 Rekursion

Rekursion ist neben der Fallunterscheidung das wichtigste (eigentlich: das *einzige*) weitere Konzept zur Definition von Funktionen. In diesem Abschnitt betrachten wir zunächst primitiv rekursive Funktionen und dann allgemeinere Arten der Rekursion. Als Beispiele dienen weitere (in HASKELL vordefinierte) Funktionen auf Listen.

3.3.1 Primitive Rekursion

Primitive Rekursion auf Listen ist gekennzeichnet durch eine Definition über die verschiedenen Formen von Listen. Eine primitiv rekursive Funktion über Listen wird definiert durch

- eine Gleichung für die leere Liste (nicht rekursiv) und
- eine rekursive Gleichung für die nicht-leere Liste, bei der der rekursive Aufruf auf die unveränderte Restliste angewendet wird. [1]

Als Beispiel betrachten wir die Konkatenation einer Liste von Listen zu einer einzigen "flach geklopften" Liste:

```
concat :: [[a]]→ [a]
concat []     = []
concat (x:xs) = x ++ concat xs
```

Weitere Beispiele unter den vordefinierten Listenfunktionen sind `sum`, `product`, `and` und `or`:

```
sum, product :: Num a ⇒ [a]→ a
sum []       =  0
sum (x:xs)   =  x + sum xs

product []     = 1
product (x:xs) = x ∗ product xs

and, or    :: [Bool]→ Bool
and []     =  True
and (x:xs) =  x && and xs

or  []     =  False
or  (x:xs) =  x || or xs
```

Die Anwendung dieser Gleichungen ergibt folgende Zwischenergebnisse:

[1] Dann lässt sich die Termination von Funktionen leicht nachweisen.

```
concat [A, B, C]       ⇝* A ++B ++C++[]
sum [4,7,3]            ⇝* 4 +7 +3 +0
product [4,7,3]        ⇝* 4 * 7 * 3 * 1
and [True,True,False]  ⇝* True && True && False && True
or [True,True,False]   ⇝* True || True || False || False
```

3.3.2 Nicht primitive Rekursion

Neben der primitiven Rekursion gibt es auch allgemeinere Formen der Rekursion:

- Rekursion über mehrere Listenargumente,
- Rekursion über eine andere Datenstruktur und
- Rekursion mit einer anderen Zerlegung der Listen als in Kopf und Rest.

Die vordefinierte Funktion `zip` ist ein Beispiel für Rekursion über mehrere Argumente. Die duale Funktion `unzip` ist primitiv rekursiv.

```
zip :: [a]→ [b]→ [(a, b)]
zip [] _ = []
zip _ [] = []
zip (x:xs) (y:ys) = (x,y):(zip xs ys)

unzip           :: [(a,b)] → ([a],[b])
unzip []        =  []
unzip ((x,y):t) =  (x:xs,y:ys) where (xs,ys) = unzip t
```

Mit **where** wird auf der rechten Seiten der letzten Gleichung eine Hilfsdefinition eingeführt, die wir allgemein im nächsten Abschnitt erklären.

Die Funktionen `zip` und `unzip` sind "annähernd invers" zueinander; für gleich lange Listen ℓ und ℓ' gilt

$$\forall \ell, \ell' \in [\alpha] : \mathit{length}\ \ell == \mathit{length}\ \ell \Rightarrow \mathit{unzip}(\mathit{zip}\,\ell\,\ell') = (\ell, \ell')$$

Schon die Fakultätsfunktion war ein Beispiel für Rekursion über natürlichen Zahlen. Vier weitere sind die zueinander dualen Listenfunktionen `take` und `drop`, die Teilstücke am Anfang und Ende einer Liste auswählen, und die Funktion `replicate`, mit der eine Liste mit n Kopien eines Elementes erzeugt werden kann. Die Funktion `splitAt` ist dagegen als einfache Kombination von `take` und `drop` definiert.

```
take, drop    :: Int→ [a]→ [a]
take 0 _      =  []
take _ []     =  []
take n (x:xs) | n > 0     =  x: take (n-1) xs
              | otherwise =  error "take:␣negative␣Argument"
```

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

splitAt                 :: Int → [a] → ([a],[a])
splitAt n xs            =  (take n xs, drop n xs)

replicate                :: Int → a → [a]
replicate n x | n ≤ 0    =  []
              | otherwise =  x: replicate (n-1) x
```

Weil es in HASKELL nur ganze Zahlen gibt, müssen negative Zahlen als mögliche Fehlerquelle mit behandelt werden.

Beispiel 3.3 (Ein letztes Mal Palindrome). In Beispiel 3.1 hatten die Funktion `palindrom` elegant mit Listenumschreibungen definiert. Mit `splitAt` können wir unnütze Vergleiche sparen:

```
palindrom' w = front == reverse (if even l then rear else tail rear)
               where w' = [ toLower c | c← w, isAlpha c]
                     l  = length w'
                     (front,rear) = splitAt (length w' `div` 2) w'
```

Bei Wörtern mit ungerader Buchstabenanzahl muss der mittlere Buchstabe aus dem hinteren Listenteil entfernt werden – sonst sind die beiden Teillisten nicht gleich lang. Der mittlere Buchstabe wird also nicht verglichen – und das ist in Ordnung!

3.3.3 Sortieren von Listen

Sortieralgorithmen arbeiten nach dem folgendem Schema (*divide and conquer*):

- Die unsortierte Liste ℓ wird *aufgespalten*.
- Die Teillisten werden (rekursiv) *sortiert*.
- Die sortierten Teillisten werden *zusammengefügt*.

Beim Aufspalten wird von einfacheren Algorithmen der Kopf der Liste abgespalten, während effizientere Verfahren die Liste in möglichst gleich große Teillisten aufspalten. Je nach Algorithmus ist entweder das Aufteilen einfach und das Zusammenfügen aufwändig oder umgekehrt. Das führt folgender Klassifizierung der vier geläufigsten Sortierverfahren in Tabelle 3.1, die wir im Folgenfden vorstellen.

*Beispiel 3.4 (*Selectionsort*).* Der Sortieralgorithmus *Selectionsort* entfernt jeweils das kleinste Element der Liste und fügt es in die Ergebnisliste ein. [2]

[2] Die Funktionen `minimum` und `delete` sind vordefiniert.

Aufspalten / Zusammenfügen	Kopf und Rest	gleich große Teillisten
aufwändig/einfach	*selection sort*	*quicksort*
einfach/aufwändig	*insertion sort*	*mergesort*

Tabelle 3.1. Klassifizierung von Sortierverfahren

```
ssort :: Ord a ⇒ [a]→ [a]
ssort []  = []
ssort xs = minimum xs : ssort (delete m xs)
           where minumum :: Ord a ⇒ [a] → a
                 minimum []       = error "minimum:␣empty␣list!"
                 minimum [x]      = x
                 minimum (x:y:xs) | x < y     = minimum (x:xs)
                                  | otherwise = minimum (y:xs)
                 delete :: Eq a ⇒ a → [a] → [a]
                 delete x []                  = []
                 delete x (y: ys) | x == y    = ys
                                  | otherwise = y: delete x ys
```

Diese Fassung von *Selectionsort* ist insofern verschwenderisch, als die Listen jeweils zweimal durchsucht werden, um zuerst das kleinste Element zu finden und es danach zu entfernen. In der folgenden Fassung wird das in der Hilfsfunktion `split` in einem Schritt erledigt.

```
ssort' :: Ord a ⇒ [a]→ [a]
ssort' [] = []
ssort' (x:xs) = m : ssort' r
          where (m,r) = split x xs
                split :: (Ord a) ⇒ a → [a] → (a,[a])
                split x [] = (x,[])
                split x xs = if x < m then (x, m:r) else (m, x:r)
                             where (m,r) = split x r
```

*Beispiel 3.5 (*Insertionsort*).* Der Sortieralgorithmus *Insertionsort* fügt das erste Element in die rekursiv sortierte Ergebnisliste ein. Die Hilfsfunktion `insert` "blättert" in der sortierten Liste bis zur richtigen Stelle. Dies wird mit der vordefinierten Funktion `span` erledigt. [3]

```
isort        :: (Ord a) ⇒ [a] → [a]
isort []     = []
isort (x:xs) = insert x (isort xs)
               where insert :: (Ord a) ⇒ a → [a] → [a]
                     insert x xs = ys ++ [x] ++ zs
                                   where (ys,zs) = span (\ y → y≤x) xs
                     span :: (a → Bool) → [a] → ([a],[a])
```

[3] Die Funktion `insert` könnte man aus `Data.list` importieren.

```
span p []           = ([],[])
span p xs@(x:xs')
   | p x       = (x:ys,zs)
   | otherwise = ([],xs)
          where (ys,zs) = span p xs'
```

Beispiel 3.6 (Quicksort). Der Sortieralgorithmus *Quicksort* zerlegt eine Liste in zwei Teillisten, deren Elemente kleiner bzw. größer sind als das erste Element, sortiert diese Teillisten rekursiv und verkettet die Ergebnisse. Das lässt sich mit Listenumschreibungen elegant definieren.

```
qsort :: Ord a ⇒ [a]→ [a]
qsort [] = []
qsort (p:xs) = qsort [ y | y ← xs, y ≤ p ]
             ++ [p] ++
             qsort [ y | y ← xs, y > p ]
```

Hierin ist ++ die vordefinierte Operation für *Listenverkettung*. Diese Fassung ist insofern verschwenderisch, als die Listen jeweils zweimal durchsucht werden, anstatt sie in einem Schritt in kleinere und größere Elemente aufzuteilen. In der zweiten Fassung wird das von der Hilfsfunktion `splitBy` in einem Durchgang erledigt.

```
qsort' :: Ord a ⇒ [a]→ [a]
qsort' [] = []
qsort' (p:xs) = (qsort' ls) ++ [p]++ (qsort' gs)
              where
              (ls,gs)= splitBy p xs
              splitBy :: Ord a ⇒ a → [a]→ ([a],[a])
              splitBy p []     = ([],[])
              splitBy p (x:xs) =
                    if p ≤ x then (ls,x:gs) else (x:ls,gs)
                    where (ls,gs) = splitBy p xs
```

Beispiel 3.7 (Mergesort). Der Sortieralgorithmus *Mergesort* zerlegt eine Liste in zwei (annähernd) gleich lange Teillisten, sortiert diese Teillisten rekursiv und fügt die Ergebnisse orgnungserhaltend zusammen (*merge*).

```
msort :: Ord a ⇒ [a]→ [a]
msort xs
  | length xs ≤ 1 = xs
  | otherwise = merge (msort ys) (msort zs)
              where
              (ys, zs) = splitAt ((length xs) `div` 2) xs
              merge :: Ord a ⇒ [a]→ [a]→ [a]
              merge [] x = x
              merge y [] = y
```

```
merge (x:xs) (y:ys)
   | x≤ y     = x: merge xs (y:ys)
   | otherwise = y: merge (x:xs) ys
```

Beispiel 3.8 (Das n-Damen-Problem). Das Problem lautet: Wie können n Damen auf einem Schachbrett mit $n \times n$ Feldern platziert werden, ohne dass sie einander bedrohen?

Als Eingabe erwartet das Programm die Anzahl n der zu verteilenden Damen. Das Ergebnis ist eine Liste von Lösungen; jede Lösung ist eine Liste von n Positionen (i, j) der Damen auf dem Schachbrett.

```
type Pos = (Int, Int)
queens :: Int→ [[Pos]]
```

Die Formulierung ist rekursiv über n:

- Für $n = 0$ gibt es keine Dame, also auch kein Problem und keine Lösung.
- Die Lösungen für $n > 0$ Damen erhält man, indem man alle Lösungen für $n - 1$ Damen bestimmt, und die n-te Dame so stellt, dass keine andere sie bedroht. Dabei muss die n-te Dame in der n-ten Spalte plaziert werden.

Die Hauptfunktion lautet:

```
queens num = qu num
  where
  qu :: Int→ [[Pos]]
  qu n | n == 0  = [[]]
       | otherwise = [ p++ [(n, m)] | p ← qu (n-1),
                                      m ← [1.. num],
                                      safe p (n, m)]
```

In der Listenumschreibung brauchen wir wieder mehrere Generatoren. Die **where**-Definition erlaubt es uns, in der Definition der Hilfsfunktion `qu` nicht allein deren Parameter `n` zu benutzen, sondern auch den Parameter `num` der global definierten Funktion `queens`.

Die Position für eine neue Dame ist *sicher*, wenn sie durch keine anderen bedroht wird:

```
safe :: [Pos]→ Pos→ Bool
safe others new =
   and [ not (threatens other new) | other ← others ]
```

Zwei Positionen (i, j) und $m, n)$ *bedrohen* einander, wenn in der gleichen Zeile oder auf einer Diagonalen liegen. (Da wir die neue Dame in eine neue Spalte stellen, kann noch keine alte Dame in dieser Spalte stehen!)

```
threatens :: Pos→ Pos→ Bool
threatens (i, j) (m, n) =
   (j== n) || (i+j == m+n) || (i-j == m-n)
```

3.4 Haskells Eck

An dieser Stelle wollen wir einen Überblick über die Konzepte von HASKELL geben, die wir bis jetzt kennengelernt haben.

3.4.1 Module

Ein Modul besteht aus einem Kopf mit Importen und Vereinbarungen (*declarations*). Der Modulkopf hat die Form

module m **where**
import m_1
$\vdots$
import m_n

Dabei sollte der Modul in einer Datei m`.hs` (bzw., als *literate script* in einer Datei m`.lhs`) abgespeichert werden.

Bisher kennen wir vier Arten von Vereinbarungen:

- Datentypvereinbarungen,
- Typsynonym-Vereinbarungen,
- Funktionssignaturen und
- Funktionsgleichungen

Die Reihenfolge dieser Vereinbarungen ist beliebig.

3.4.2 Typsynonym

Eine Typsynonym wird vereinbart mit

type $t\,\alpha_1 \cdots \alpha_k = T$

wobei T ein Typausdruck ist, wie oben beschrieben.

3.4.3 Funktionssignatur

Die Signatur einer Funktion f mit $n \geqslant 0$ Parametern wird vereinbart mit

$f :: [Con \Rightarrow] T_1 \to \cdots \to T_k \to T_0$

(Wenn f ein Operator ist, d.h., aus Sonderzeichen besteht, muss es geklammert werden Dann ist die Anzahl k der Argumente gleich 2.) Auch hier sind die T_i Typausdrücke. (Funktionen mit $n = 0$ Parametern sind *Konstanten*.) Der optionale *Kontext Con* hat die Form

$c\,\alpha$ oder $(c_1\,\alpha_1, \ldots, c_k\,\alpha_k), k \geqslant 2$

Der Kontext gibt an, zu welchen Klassen C_i die Typvariablen α_i gehören müssen, damit die Funktion definiert ist.

3.4.4 Funktionsgleichung

Eine Gleichung definiert eine Funktion f für die Werte, auf die die angegebenen Muster m_i für die Parameter passen; die rechte Seite ist ein Ausdruck E, oder eine Liste von *bewachten Ausdrücken* (*guarded expressions*):

$$\begin{array}{ll} f\, m_1 \cdots m_k = & E \quad \text{oder} \\ f\, m_1 \cdots m_k \mid & G_1 = E_1 \\ & \vdots \\ \mid & G_n = E_n \end{array}$$

Ausdrücke können lokale Funktionsdefinitionen mit **where** oder **let** einführen; sie können mit **if** anhand eines Prädikats oder mit **case** anhand von Mustern Ausdrücke auswählen, oder einfach geschachtelte Funktionsanwendungen enthalten.

$$\begin{array}{lll} E & \textbf{let}\ D_1 & \textbf{if}\ \ E \\ \textbf{where}\ D_1 & \cdots & \textbf{then}\ E_1 \\ \cdots & D_n & \textbf{else}\ E_2 \\ D_n & \textbf{in}\ E & \end{array} \qquad \begin{array}{ll} \textbf{case}\ E & \\ \textbf{of} & m_1 \to E_1 \\ & \cdots \\ & m_n \to E_n \end{array}$$

Dieses Schema berücksichtigt noch nicht alle Varianten, die in HASKELL möglich sind, sollte aber für einen ersten Überblick reichen.

3.4.5 Muster

Muster (*pattern*) sind eingeschränkte Ausdrücke, die nur mit (Wert-) Konstruktoren und Variablen gebildet werden. Mit Mustern werden in Funktionsdefinitionen Werte in ihre Komponenten zerlegt und Fallunterscheidungen anhand von Werten getroffen.

Als Beispiel betrachten wir zwei Schreibweisen für die Definition der Funktion `ordered`, die Muster auf der linken Seite von Funktionsgleichungen bzw. in einem **case**-Ausdruck benutzen:

```
ordered :: List t → Bool
ordered  Empty              = True
ordered (Cons x Empty)      = True
ordered (Cons x (Cons y xs)) = x ≤ y && ordered (Cons y xs)

ordered = \ xs →
          case xs of
            Empty             → True
            Cons x Empty      → True
            Cons x (Cons y xs) → x ≤ y && ordered (Cons y xs)
```

Hier treten die Muster `Empty`, `Cons x Empty` und `Cons x (Cons y xs)` auf; sie enthalten die Konstruktoren `Empty` und `Cons` sowie die Variablen `x`, `y` und `xs`.

1. Jede Variable darf in einem Muster höchstens *einmal* auftreten. Also ist `Cons x (Cons x xs)` kein erlaubtes Muster, wohl aber ein erlaubter Ausdruck.
2. Von den drei Mustern in diesem Beispiel passt auf jeden Listenwert genau eines. Muster werden immer in der Reihenfolge verglichen, in der sie im Programm stehen; dabei ist es auch erlaubt, verschiedene Muster zu benutzen, die auf ein und denselben Wert passen könnten. Beispielsweise könnte man `ordered` kürzer so definieren:

   ```
   ordered (Cons x (Cons y xs)) = x ≤ y && ordered (y:xs)
   ordered x                    = True
   ```

   Alle Listen mit weniger als zwei Elementen sind geordnet. Wenn man solche *überlappende Muster* benutzt, ist die Reihenfolge der Gleichungen relevant, und nicht jede Gleichung definiert ein Lemma für das Verhalten der Funktion: Die zweite Gleichung gilt nur für solche `x`, die *nicht* auf das Muster der ersten Gleichung passen.
3. In der letzten Definition von `ordered` wird die Variable `x` auf der rechten Seite der Gleichung nicht benötigt. In solchen Fällen sollte man statt dessen die "anonyme Variable" `_` (engl. *wild card*) in das Muster schreiben, also "`ordered _`". Dies verhindert, dass die Variable versehentlich auf der rechten Seite doch benutzt wird.
4. Die Funktion `head` gibt den Kopf einer Liste zurück.

   ```
   head :: List t → t
   head (Cons h _) = h
   ```

   Für leere Listen ist ihr Ergebnis *undefiniert*; man bekommt eine Fehlermeldung ("`*** Exception: Prelude.head: empty list`"). Die Muster von `head` schöpfen also – im Gegensatz zu denen von `ordered` – nicht alle möglichen Fälle aus.
5. Tupeltypen haben nur einen einzigen Wertkonstruktor. Deshalb wird in der Definition

   ```
   fst :: (a,b) → a
   fst (x,_) = x
   ```

   der Vergleich des Musters `(x,_)` immer gelingen. Der Vergleich des Musters `(x: xs,_)` kann dagegen scheitern.
6. Das Muster "x `@` M" passt auf dieselben Werte wie das Muster M, bindet dessen Wert aber zusätzlich an die Variable x. Dies ist nützlich, wenn der Wert des gesamten Muster M auf der rechten Seite benutzt werden soll:

   ```
   ordered (Cons x (r@Cons y xs)) = x ≤ y && ordered r
   ordered x _                    = True
   ```

   Muster werden nicht unbedingt gebraucht. Mithilfe der Inspektionsfunktion `null` und den Selektorfunktionen `head` und `tail` kann man eine die Funktion `ordered` auch definieren:

```
ordered = \ xs → null  xs || null (tail xs)
              || head xs ≤ head (tail xs) && ordered (tail xs)
```

Aber da sieht man nicht so schnell, was gemeint ist, oder?

(Dies ist Fassung 2.1.05 von 4. Februar 2010.)

4

Funktionen höherer Ordnung

In den vorangegangenen Kapiteln haben wir gesehen, dass in funktionalen Sprachen alle zusammengesetzte Werte Ausdrücke über Konstruktoren sind, die intern als Bäume bzw. als Graphen dargestellt werden.

In diesem Kapitel werden wir sehen, dass Funktionen umgekehrt auch (fast) ganz normale Werte sind: Funktionen können Argumente oder Ergebnisse von Funktionen und Bestandteile von anderen Werten sein.

Funktionen, die selber Funktionen als Argumente haben, werden *Funktionen höherer Ordnung* genannt; mit ihnen können aus Funktionen neue Funktionen abgeleitet werden. Deshalb heißen sie auch *Kombinatoren*. Kombinatoren erlauben einen sehr abstrakten Programmierstil, der allerdings etwas gewöhnungsbedürftig ist.

Funktionen können auch Funktionen als Ergebnisse zurückgeben. Tatsächlich werden alle n-stelligen Funktionen standardmäßig als einstellige Funktion aufgefasst, die eine $n-1$-stellige Funktion zurückliefert. Mit Funktionen kann also gerechnet werden wie mit allen anderen Daten auch.

Schließlich können Funktionsräume auch als Komponenten von anderen Datentypen benutzt werden. Trotzdem sind sie nicht total erstklassige Werte, weil einige Operationen, die für andere Datentypen immer existieren, auf Funktionen nicht definiert werden können, wie das Gleichheitsprädikat und die Umwandlung in Zeichenketten.

4.1 Listenkombinatoren

In den vorangegangenen Kapiteln haben wir viele Funktionen auf Listen definiert. Rückblickend können wir bei manchen von ihnen gemeinsame *Berechnungsmuster* entdecken.

- Einige Funktionen wenden eine Funktion f elementweise auf eine Liste an:
  - In der Funktion `palindrom` in Abschnitt 3.2 wird die Funktion `toLower` auf alle Buchstaben einer Zeichenkette angewendet.

Abb. 4.1. Elementweises Anwenden einer Funktion (*map*)

Abb. 4.2. Filtern mit einem Prädikat (*filter*)

Abb. 4.3. Verknüpfen der Listenelemente mit einer Operation (*foldr*)

  - In der Hilfsfunktion `moveSeq` in Abschnitt 5.3 wird die Funktion `add` auf alle Eckpunkte eines Polygons angewendet.
- Manche Funktionen *filtern* Elemente aus einer Liste heraus:
  - Die Funktionen `books` und `returnLoan` in Abschnitt 3.2 filtern aus eine Entleihkartei alle Bücher und Leihkarten heraus, die eine bestimmtes Prädikat P erfüllen.
- Andere Funktionen verknüpfen alle Elemente einer Liste mit einer binären Operation:
  - Die Funktionen `sum`, `product`, `or` und `and` in Abschnitt 3.3.1 verknüpfen die Elemente von Listen mit den Operationen $+$, $*$, `||` bzw. `&&`.

  Genauer gesagt wird der Listenkonstruktor (:) durch die Anwendung einer binären Operation $\otimes$ ersetzt und die leere Liste durch einen Wert $\mathbf{1}_\otimes$, typischerweise das neutrale Element von $\otimes$ ist, also folgende Gleichung erfüllt:

  $$x \otimes \mathbf{1}_\otimes = x$$

Diese Berechnungsmuster können allgemein als Funktionen definiert werden, die andere Funktionen als Parameter haben. Solche *Kombinatoren* können an vielen Stellen benutzt werden und erlauben es, Funktionen sehr abstrakt als Kombinationen von Berechnungsmustern zu definieren.

Das elementweise Anwenden einer Funktion f auf alle Elemente einer Liste ist vordefiniert:

```
map :: (a→ b)→ [a]→ [b]
map f []     = []
map f (x:xs) = (f x):(map f xs)
```

Damit lässt sich die Funktion `moveSeq` so definieren:

```
moveSeq :: Point → [Point] → [Point]
moveSeq p xs = map addPoint xs where addPoint :: Point → Point
                                     addPoint x = add p x
                                     add :: Point → Point → Point
                                     add (x,y) (dx,dy) = (x+dx,y+dy)
```

Das Filtern von Elementen einer Liste mit einem Prädikat ist ebenfalls vordefiniert.

```
filter :: (a→ Bool)→ [a]→ [a]
filter p []                 = []
filter p (x:xs) | p x       = x:(filter p xs)
                | otherwise = filter p xs
```

Damit ist die Listenumschreibung als eine spezielle Schreibweise für die Kombination von `map` und `filter` entlarvt:

$$[T\,x|x \leftarrow l, P\,x] \equiv \texttt{map t (filter p } l\texttt{)}$$
$$\texttt{where t } x = T\,x$$
$$\texttt{p } x = P\,x$$

Beispielsweise können die Kehrwerte aller ungeraden Zahlen von 1 bis 100 sowohl mit der Listenumschreibung

```
[ 1 / x | x ← [1..100], (mod x 2) == 1 ]
```

als auch mit Kombinatoren definiert werden:

```
map t (filter p [1..100])
       where t x = 1 / x
             p x = (mod x 2) == 1
```

(Genau so werden Listenumschreibungen auch implementiert.)

Das Verknüpfen von Listenelementen mit einer binären Operation wird aus historischen Gründen `foldr` (für *fold to the right*) genannt. Für die leere Liste wird ein neutraler Wert `e` zurückgegeben, und im rekursiven Fall wird der Listenkopf und der Rekursionswert mit der binären Operation `op` verknüpft.

```
foldr :: (a→ b→ b)→ b→ [a]→ b
foldr op e []     = e
foldr op e (x:xs) =  x `op` (foldr op e xs)
```

Nun können einige vordefinierte Funktionen für Listen abstrakter definiert werden:

```
sum, product :: [Int]→ Int
sum xs       = foldr (+) 0 xs
product xs   = foldr (*) 1 xs

or, and :: [Bool]→ Bool
or xs    = foldr (||) False xs
and xs   = foldr (&&) True xs
```

Ein weiteres Beispiel ist das "Flachklopfen" von Listen.

```
concat :: [[a]]→ [a]
concat xs = foldr (++) [] xs
```

Auch `map` und `filter` können mit `foldr` definiert werden, siehe Aufgabe 9 auf Übungsblatt 4. Eine Variante von `filter` teilt eine Liste in zwei: eine erfüllt das Prädikat, die andere nicht:

```
partition :: (a → Bool) → [a] → ([a], [a])
partition p xs = (filter p xs, filter (not ∘ p) xs)
```

Wie viele andere Definitionen vordefinierter Funktionen ist dies eher eine Gleichung, die die Eigenschaften von `partition` beschreibt; implementiert ist es (hoffentlich) so, dass die Liste nur einmal durchlaufen wird.

Zum *Zerlegen* von Listen hatten wir in Abschnitt 3.3.2 schon die Funktionen `take, drop` kennengelernt, die n vordere Elemente einer Liste nehmen bzw. weglassen.

Dies können wir zu Kombinatoren erweitern, die den längsten Präfix einer Liste nehmen bzw. weglassen, für den ein Prädikat P gilt:

```
takeWhile :: (a → Bool) → [a] → [a]
takeWhile p []          =  []
takeWhile p (x:xs)
            | p x       =  x : takeWhile p xs
            | otherwise =  []

dropWhile :: (a → Bool) → [a] → [a]
dropWhile p []          =  []
dropWhile p (x:xs)
            | p x       =  dropWhile p xs
            | otherwise =  x : xs
```

Es gilt: `takeWhile p xs ++dropWhile p xs ==xs`. Die Kombination von `takeWhile` und `dropWhile` ist auch vordefiniert:

```
span      :: (a → Bool) → [a] → ([a],[a])
span p xs = (takeWhile p xs, dropWhile p xs)
```

Damit kann ordnungserhaltendes Einfügen so definiert werden:

```
ins :: Ord a ⇒ a→ [a]→ [a]
ins x xs = lessx ++ [x] ++ grteqx
           where (lessx, grteqx) = span less xs
                 less z = z < x
```

Dann kann Sortieren durch Einfügen (*insertion sort*) leicht mit `foldr` definiert werden:

```
isort :: Ord a ⇒ [a]→ [a]
isort xs = foldr ins [] xs
```

Bisher haben alle Sortieralgorithmen aufsteigend nach der Ordnungsrelation "$<$" sortiert. Dies muss nicht immer gewünscht sein. Allgemeiner könnten wir sortieren, wenn wir die Ordnungsrelation `ord` als Parameter übergeben. Allerdings sind nicht alle zweistelligen Prädikate sinnvolle Kandidaten für `ord`. Vielmehr muss das Prädikat eine *totale Ordnung* definieren, d. h., transitiv, *antisymmetrisch*, reflexiv und total sein.

- transitiv: x `ord` y, y `ord` $z \Rightarrow x$ `ord` z
- antisymmetrisch: x `ord` $y \wedge y$ `ord` $x \Rightarrow x = y$
- reflexiv: x `ord` x
- total: x `ord` $y \vee y$ `ord` x

Dann kann beispielsweise *Quicksort* so definiert werden

```
qsortBy :: (a→ a→ Bool)→ [a]→ [a]
qsortBy ord [] = []
qsortBy ord (x:xs) = qsortBy ord l ++ [x] ++ qsortBy ord r
                     where l = [y| y←xs, ord y x]
                           r = [y| y←xs, not (ord y x)]
```

4.2 Rechnen mit Funktionen

Nachdem wir im letzten Abschnitt schon Funktionen als *Argumente* von Kombinatoren wie `map`, `filter`, `foldr` usw. kennengelernt haben, wollen wir jetzt untersuchen, wie neue Funktionen aus alten berechnet werden können, und dabei einige nützliche Konzepte wie anonyme Funktionen, partielle Anwendung von Funktionen und die η-Kontraktion kennenlernen.

4.2.1 Funktionen als Ergebnisse

Funktionen können Funktionen aus alten zusammensetzen. `twice` wendet eine Funktion zweimal hintereinander an:

```
twice :: (a→ a)→ (a→ a)
twice f x = f (f x)
```

`iter` wendet eine Funktion n-mal hintereinander an:

```
iter :: Int→ (a→ a)→ a→ a
iter n f x |  n >  0     = f (iter (n-1) f x)
           |  otherwise = x
```

Diese Funktionen kennen wir schon von Aufgabenblatt 1.

Die einfachste Funktion ist die Identität:

```
id :: a→ a
id x = x
```

Sie ist nützlicher als man vermuten könnte, weil sie die *neutrale* Funktion ist.

Das Hintereinanderschalten von Funktionen, die *Funktionskomposition*, ist eine binäre Operation:

```
(∘) :: (b→ c) → (a→ b)→ a→ c
(f ∘ g) x = f (g x)
```

Die Funktionskomposition ist assoziativ und hat die Identität als neutrales Element:

$$f \circ id = f = id \circ f$$
$$f \circ (g \circ h) = (f \circ g) \circ h$$

Bei der Komposition `(f ∘ g)` wird die Funktion `f` *nach* `g` angewendet. Aber auch die "Vorwärts-Funktionskomposition" kann definiert werden:

```
(>.>) :: (a→ b)→ (b→ c)→ a→ c
(f >.> g) x = g (f x)
```

Dies ist aber im Gegensatz zu `(∘)` *nicht* vordefiniert!

Mit Komposition und Identität können Funktionen prägnanter beschrieben werden:

```
twice':: (a→ a)→ a→ a
twice' f = f ∘ f

iter' :: Int→ (a→ a)→ a→ a
iter' n f | n >  0    = f ∘ iter' (n-1) f
          | otherwise = id
```

Auch für die Angabe der funktionalen Parameter von Kombinatoren können diese Funktionen benutzt werden. Noch zwei weitere Funktionen können dafür nützlich sein: Die Funktion `const` liefert ihr erstes Argument und ignoriert ihr zweites; der Kombinator `flip` vertauscht die beiden Argumente der ihm übergebenen Funktion `f`.

```
const            :: a → b → a
const x _        =  x

flip :: (a → b → c) → b → a → c
flip f x y = f y x
```

4.2.2 Anonyme Funktionen

Nicht *jede* Funktion muß unbedingt einen Namen haben. Manche Hilfsfunktionen werden definiert und nur ein einziges Mal benutzt. Zum Beispiel die Funktion `less` in der Einfüge-Funktion:

```
ins x xs = lessx ++ [x] ++ grteqx where
  (lessx, grteqx) = span less xs
  less z = z < x
```

Dann könnte man besser gleich den Rumpf von `less` einsetzen.

```
ins' x xs = lessx ++ [x] ++ grteqx where
  (lessx, grteqx) = span (\z→ z < x) xs
```

Hier ist `\z→ z <x` die HASKELL-Schreibweise eines *Lambda-Ausdrucks* $\lambda x.E$, der die namenlose Funktion beschreibt, die für jedes Argument x den Ausdruck E auswertet. Es gilt:

$$\lambda x.E \equiv f \textbf{ where } fx = E$$

Auch *pattern matching* ist möglich, aber meist nur für unwiderlegbare Pattern wie "$\lambda(x, y).E$" sinnvoll.

Beispiel 4.1 (Primzahlen). Das *Sieb des Erathostenes* berechnet Primzahlen, indem für jede gefundene Primzahl `p` alle Vielfachen aus der Kandidatenmenge herausgesiebt werden. Dazu filtern wir die Liste der Kandidaten mit `(\n→ n ‘mod‘ p /=0)`.

```
sieve :: [Integer]→ [Integer]
sieve [] = []
sieve (p:ps) = p:(sieve (filter (\n→ n ‘mod‘ p /= 0) ps))
```

Die Primzahlen im Intervall `[1..n]` können dann mit

```
primes :: Integer→ [Integer]
primes n = sieve [2..n]
```

berechnet werden.

4.2.3 Die η-Kontraktion

Bisher haben wir Funktionen immer *elementweise* definiert, d. h., in der Form

$$f\ x = E \text{ oder } \lambda x.E$$

Manchmal hat der Ausdruck E die Form $F\ x$, wobei x in F nicht vorkommt. Dann gilt das Gesetz der η-Kontraktion (sprich: "Eta-Kontraktion"):

$$\lambda x.Fx \Longleftrightarrow F \text{ und } f\ x = F\ x \Longleftrightarrow f = F$$

Ein Beispiel: die vordefinierten Funktionen `any` und `all` berechnen eine Disjunktion bzw. Konjunktion von Prädikaten über Listen:

```
all, any :: (a→ Bool)→ [a]→ Bool
any p      = or  ∘ map p
all p      = and ∘ map p
```

Diese Definitionen sind äquivalent zu

```
any p x =or (map p x)  bzw.  all p x =and (map p x)
```

definieren die Funktion aber *als Ganzes*, nicht was ihre Anwendung auf Argumente berechnet.

4.2.4 Partielle Funktionsanwendung

In Ausdrücken brauchen nicht alle n Argumente einer n-stelligen Funktion angegeben werden.

```
double :: String→ String
double = concat ∘ map (replicate 2)
```

(*Zur Erinnerung*: `replicate ::Int→ a→ [a]` erstellt eine Liste mit k Kopien eines Wertes.) In der Definition von `double` sind nur jeweils eines der beiden Argumente von `map` und `replicate` genannt. Wir nennen das eine *partielle Anwendung* einer Funktion.

Welchen Typ hat der Teilausdruck `replicate 2`? Da `replicate` selbst den Typ `Int → a → [a]` und im Ausdruck ein Parameter `2` vom Typ `Int` angegeben ist, hat `replicate 2` den Typ `a → [a]`. Analog hat `map` den Typ `(a → b) → [a] → [a]` und ein Argument `replicate 2` vom Typ `Int → [Int]`, so dass `map (replicate 2` den Typ `[Int] → [[Int]]`.

Allgemein gilt die folgende Kürzungsregel bei partiellen Funktionsapplikationen.

Wird die Funktion

$$f :: t_1 \to t_2 \to \ldots \to t_n \to t$$

angewendet auf k Argumente mit $k \leq n$

$$e_1 :: t_1, e_2 :: t_2, \ldots, e_k :: t_k$$

werden die Typen der Argumente *gekürzt*:

$$\begin{array}{c} f :: \not{t_1} \to \not{t_2} \to \ldots \to \not{t_k} \to t_{k+1} \to \ldots \to t_n \to t \\ f\, e_1 \ldots e_k :: t_{k+1} \to \ldots \to t_n \to t \end{array}$$

Auch Operationen können partiell angewendet werden. Dann muss der Operator mit seinem vorhandenen (vorangestellten oder nachgestellten) Argument geklammert werden.

```
elem :: (Eq a) ⇒ a→ [a]→ Bool
elem x = any (== x)
```

Ein Ausdruck wie `(==x)` heißt in Haskell *Abschnitt* (engl. *section*) des Operators `(==)`; dies entspricht der anonymen Funktion `\e→ e ==x`.

Schließlich können wir uns fragen, welcher Unterschied zwischen einer Funktion mit zwei Parametern $f :: a \to b \to c$ und einer Funktion mit einem Tupel-Parameter $f :: (a, b) \to c$ besteht. In der ersten Form ist partielle Anwendung möglich, in der zweiten nicht. Ansonsten sind beide Formen äquivalent. Deshalb gibt es auch vordefinierte Funktionen, die eine Form in die andere überführen. Die erste Form heißt "ge*curry*t", nach dem Logiker Haskell Curry, der von Moses Schönfinkel diese Form von Funktionen übernahm.

```
curry   :: ((a, b)→ c)→ a→ b→ c
curry f a b = f (a, b)

uncurry :: (a→ b→ c)→ (a, b)→ c
uncurry f (a, b) = f a b
```

Diese Funktionen erfüllen die Gleichungen

$$uncurry \circ curry = id \qquad curry \circ uncurry = id$$

4.3 Funktionsräume als Datentypen

In den vorangegangenen Abschnitten haben wir gesehen, dass Funktionen *fast* ganz normale Werte sind, wie Zahlen oder Listen. Funktionen können nicht nur Argumente und Ergebnisse von Funktionen sein, sie können auch Komponenten von Datentypen sein. Dazu wollen wir noch einmal auf den abstrakten Datentyp *Speicher* zurückkommen.

Beispiel 4.2 (Speicher als Funktion). Ein Speicher mit Funktionen `initial`, `value` und `update` kann als eine Funktion realisiert werden.

Der Speicher mit Zellen vom Indextyp `a` und Einträgen voma yp `b` wird als Funktion von `a` nach `mayba b` realisiert.

```
data Store a b = St (a→ Maybe b)
```

Der Speicher wird als die total undefinierte Funktion initialisiert.

```
initial :: Store a b
initial  = St (const Nothing)
```

Das Nachschlagen eines Werts im Speicher ist schlicht die Anwendung der Funktion:

```
value :: Store a b → a → Maybe b
value (St f) a  = f a
```

Das Aktualisieren des Inhalts einer Zelle kann als punktweise Funktionsdefinition realisiert werden.

```
update :: Eq a ⇒ Store a b → a → b → Store a b
update (St f) a b
  = St (\x→ if x== a then Just b else f x)
```

Konzept	Zahlen	Funktionsräume
Typ	`Double`	`a → b`
Literale	`042`	`\ x →E`
Konstanten	`pi`	`map`
Operationen	`*`	`F A >.>`
Ausdrücke	`3 * pi`	`(\ x→ x+1) ∘ abs`
Was fehlt?		
Vergleich	`3 ==pi`	*nicht entscheidbar*
als Zeichenkette	Show 3.14	*nicht darstellbar*

Tabelle 4.1. Funktionsräume im Vergleich zu anderen Datentypen

Das Beispiel zeigt, das Funktionsräume normale Datentypen sind – jedenfalls beinahe. Wir können uns fragen, was im Vergleich zu einem Datentypen wie `Double` die wichtigsten Konzepte von Funktionsräumen sind:

- Lambda-Abstraktionen $\lambda x \to E$ sind *Funktionsliterale*; damit werden Werte direkt hingeschrieben (konstruiert).
- Die benannten Funktionen wie `id` und `map` definieren Konstanten eines funktionalen Typs.
- Die wichtigste Operation auf Funktionen ist die *Anwendung* einer Funktion auf ein Argument. Sie hat den Typ `(a → b) → a → b` und ist so wichtig und häufig, dass sie "unsichtbar" bleibt, und einfach durch *Hintereinanderschreiben* (*Juxtaposition*) ausgedrückt wird. So beschreibt `map (+1)` die Anwendung der Funktion `map` auf ein Argument (die Funktion `\x→ x+1`!); der Ausdruck `(map (+1)) [1,2,3]` beschreibt die Anwendung der Funktion `map (+1)` auf die Liste `[1,2,3]`. Da Funktionsanwendung *linksassoziativ* ist, können wir die Klammern weglassen: `map (+1) [1,2,3]`. [1]
- Die Funktionskompositionen `(∘)` und `(>.>)` sind Operationen auf Funktionsräumen.

Ein paar Unterschiede bleiben aber doch; sie verhindern, dass Funktionen wirklich *erstklassige Werte* sein können.

1. Funktionen können nicht *verglichen* werden. Die Gleichheit von Funktionen ist im Allgemeinen unentscheidbar, weil die von Funktionen definierten *Funktionsgraphen* unendliche (rechtseindeutige) Relationen sind.
2. Aus dem gleichen Grund können Funktionen nicht als Zeichenketten dargestellt werden.

Sicher könnte man Funktionsdefinitionen vergleichen oder ausdrucken; das sind aber *syntaktische Objekte*, die nichts aussagen über die *semantische Gleichheit* oder Darstellung des damit definierten Funktionsgraphen.

Die Analogie zu anderen Datentypen, aber auch die Unterschiede sind in Tabelle 4.1 zusammengefasst.

[1] Dagegen ist der Typkonstruktor $(\to)$ *rechtsassoziativ*: $a \to b \to c$ entspricht also $a \to (b \to c)$.

4.4 Programmentwicklung

Abschließend wollen wir ein etwas größeres Beispiel mit den neuen Programmierkonzepten entwickeln. Für einen Text soll ein *Index* erstellt werden, also etwa aus dem Text

```
laber fasel\nlaber laber\nfasel laber blubb
```

eine Liste, in der für jedes Wort die Liste der Zeilen angegeben ist, in denen es auftritt:

```
blubb [3]       fasel [1, 3]       laber [1, 2, 3]
```

Für die Spezifikation der Lösung vereinbaren wir folgende Typen und die Signatur der zu implementierenden Funktion:

```
type Doc = String
type Word= String
makeIndex :: Doc→ [([Int], Word)]
```

Wir zerlegen das Problem in einzelne Schritte (und notieren in Klammern den Typ des Zwischenergebnisses.

1. Zunächst wird der Text in Zeilen aufgespalten (`[Line]` mit `type Line=String`).
2. Jede Zeile wird mit ihrer Nummer versehen (`[(Int, Line)]`).
3. Die Zeilen werden in Wörter aufgespalten und die Zeilennummer verteilt (`[(Int, Word)]`).
4. Erst wird die Liste alphabetisch nach Worten sortiert (`[(Int, Word)]`).
5. Dann werden gleiche Worte in unterschiedlichen Zeilen zusammengefasst (`[([Int], Word)]`).
6. Schließlich werden alle Worte mit weniger als vier Buchstaben gestrichen (`[([Int], Word)]`).

Unsere erste Implementierung benutzt die in Abschnitt 4.2.1 definierte Vorwärtskomposition von Funktionen:

```
type Line = String
makeIndex =
  lines       >.> -- Doc → [Line]
  numLines    >.> -- → [(Int,Line)]
  allNumWords >.> -- → [(Int,Word)]
  sortLs      >.> -- → [(Int,Word)]
  makeLists   >.> -- → [([Int],Word)]
  amalgamate  >.> -- → [([Int],Word)]
  shorten         -- → [([Int],Word)]
```

Nun zur Implementierung der einzelnen Komponenten:

- Zum Zerlegen in Zeilen benutzen wir die vordefinierte Funktion `lines ::String→ [String]`.
- Zur Numerierung der Zeilen benutzen wir `zip`:

```
numLines :: [Line]→ [(Int, Line)]
numLines lines = zip [1.. length lines] lines
```

- Um die Zeilen in Worte zu zerlegen, können wir für jede Zeile die vordefinierte Funktion `words:: String→ [String]` benutzen. Da `words` aber nur Leerzeichen als Wortzwischenräume berücksichtigt, müssen vorher alle Satzzeichen in Leerzeichen umgewandelt werden.

```
splitWords :: Line→ [Word]
splitWords = words ∘ map (\c→ if isPunct c then ' ' else c)
             where isPunct :: Char→ Bool
                   isPunct c = c `elem` ";:.,\'\"!?(){}-\\[]"
```

  `splitWords` muss auf alle Zeilen angewendet werden, und dann muss die Ergebnisliste flachgeklopft werden.

```
allNumWords :: [(Int, Line)]→ [(Int, Word)]
allNumWords = concat ∘ map oneLine
   where
   oneLine :: (Int, Line)→ [(Int, Word)]
   oneLine (num, line) = map (\w→ (num, w)) (splitWords line)
```

- Um die Liste alphabetisch nach Wörtern zu sortieren, muss eine passende Ordnungsrelation definiert werden, die die Zahlen außer Acht lässt.

```
ordWord :: (Int, Word)→ (Int, Word)→ Bool
ordWord (n1, w1) (n2, w2) =
  w1 < w2 || (w1 == w2 && n1 ≤ n2)
```

  Dann kann mit der generische Sortierfunktion `qsortBy` sortiert werden.

```
sortLs :: [(Int, Word)]→ [(Int, Word)]
sortLs = qsortBy ordWord
```

- Gleiche Worte in unterschiedlichen Zeilen werden zusammengefasst. Als Vorbereitung wird jede Zeile in eine (einelementige) Liste von Zeilen umgewandelt.

```
makeLists :: [(Int, Word)]→ [([Int], Word)]
makeLists = map (\ (l, w)→ ([l], w))
```

  Im zweiten Schritt werden gleiche Worte zusammengefasst. (Nach der Sortierung stehen gleiche Worte schon hintereinander!)

```
amalgamate :: [([Int], Word)]→ [([Int],Word)]
amalgamate []  = []
amalgamate [p] = [p]
amalgamate ((l1, w1):(l2, w2):rest)
  | w1 == w2  = amalgamate ((l1++ l2, w1):rest)
  | otherwise = (l1, w1):amalgamate ((l2, w2):rest)
```

- Schließlich werden alle Wörter mit weniger als vier Buchstaben entfernt:

  ```
  shorten :: [([Int],Word)] → [([Int],Word)]
  shorten = filter (\ (_, wd)→ length wd ≥ 4)
  ```

  (Alternative hätte dies als `shorten =filter ((≥4) ∘ length ∘ snd)`! definiert werden können.)

(Dies ist Fassung 2.0.01 von 4. Februar 2010.)

5

Algebraische Datentypen

Wir haben in früheren Kapiteln schon gesehen, dass in einer funktionalen Sprache alle zusammengesetzten Datentypen wie Tupel, Listen und Bäume algebraische Datentypen sind.

In diesem Kapitel werden wir die verschiedenen Arten von algebraischen Datentypen – Aufzählungstypen, Produkttypen, Summentypen mit und ohne Rekursion – genauer ansehen.

5.1 Aufzählungstypen

In vielen Programmiersprachen können Aufzählungen von Werten als Datentypen definiert werden, z.B. die Wochentage als die Menge

$$\mathit{Weekday} = \{\mathit{Mo}, \mathit{Tu}, \mathit{We}, \mathit{Th}, \mathit{Fr}, \mathit{Sa}, \mathit{Su}\}$$

Die Werte von Aufzählungen werden häufig als implizit definierte ganzzahlige Konstanten betrachtet, z.B.:

$$\mathit{Mo} = 0, \mathit{Tu} = 1, \mathit{We} = 2, \mathit{Th} = 3, \mathit{Fr} = 4, \mathit{Sa} = 5, \mathit{Su} = 6$$

Machen wir uns schnell klar, dass die Typesynonyme von Haskell nicht geeignet sind, um Aufzählungstypen zu realisieren. Zwar könnte man definieren

type *Weekday* = *Int*

und auch die oben genannten Konstanten einführen, doch böte dies keinerlei *Typsicherheit*: Typsynonyme sind nur Abkürzungen, die wie Makros expandiert und dann "vergessen" werden. Die Wochentage sind also ganz gewöhnliche ganzzahlige Konstanten, die wie jeder andere Zahlenwert behandelt werden können. Die Ausdrücke `Fr +15` und `Fr * Mo` wären also zulässig, obwohl die Addition keinen Wochentag liefert und Multiplikation auf Wochentagen keinen Sinn macht.

Wir brauchen aber einen Datentyp, der nur Wochentage *Mo*, *Tu*, ..., *Su* als Werte enthält *und sonst nichts*, und wir wollen für diesen Typ nur Funktionen zulassen, die Sinn machen, *und sonst keine*. Wichtig ist, dass alle Wochentage unterschiedlich sind und nicht mit Werten anderer Datentypen verwechselt werden können. Das heißt, wir wollen *Datenabstraktion* betreiben.

Dies können wir mit der einfachsten Art von algebraischem Datentyp erreichen, dem *Aufzählungstyp*:

```
data Weekday = Mo | Tu | We | Th | Fr | Sa | Su
```

Mit dem Typ `Weekday` werden zugleich automatisch die Konstruktoren definiert:

```
Mo, Tu, We, Th, Fr, Sa, Su :: Weekday,
```

So, wie er hier definiert wurde, sind für den Typ `Weekday` keine weiteren Operationen vordefiniert. Funktionen können durch *pattern matching* definiert werden:

```
isWeekend :: Weekday → Bool
isWeekend Sa = True
isWeekend Su = True
isWeekend _  = False
```

Sicher gäbe es einige Operationen, die Sinn machten: Wochentage könnten verglichen, geordnet, als Zeichenkette gedruckt oder aus einer Zeichenkette gelesen werden, sie sind aufzählbar und beschränkt (in ihrer Wertemenge). Man könnte `Weekday` damit zu einer *Instanz* der Klassen `Eq`, `Ord`, `Show`, `Read` und `Bounded` machen und die entsprechenden Operationen für Wochentage definieren.

- Gleichheit und Ungleichheit lässt sich durch Mustervergleich realisieren.
- Als Ordnung nimmt man die Reihenfolge der Konstruktoren in der Typdefinition, in diesem Fall also `Mo` < `Tu` < `We` < `Th` < `Fr` < `Sa` < `Su`.
- Die Namen der Konstruktoren bilden ihre Darstellung als Zeichenkette, also "`Mo`" ... "`Su`", die von `show` ausgegeben bzw. von `read` gelesen wird.
- Die Funktionen `succ` und `pred` der Klasse `Enum` können aus der Ordnung der Konstruktoren abgeleitet werden, so dass dann Listen-Generatoren wie `[Mo..Fr]` oder `[Mo,We..Su]` gebildet werden können. Die Konstanten der Klasse `Bounded` können definiert werden als *minBound* = *Mo* und *maxBound* = *Su*.

Für den algebraischen Typ `Weekday` kann man eine *kanonische Instanz* dieser Klassen auch automatisch aus der Typdefinition *herleiten*. Dies kann man in Haskell so spezifizieren:

```
data Weekday = Mo | Tu | We | Th | Fr | Sa | Su
               deriving (Eq, Ord, Show, Read, Enum, Bounded)
```

Dann könnte die Funktion *isWorkday* auch implementiert werden mithilfe der Funktion `elem:: (Eq a) ⇒a → [a] →Bool` und des Listengenerators "`..`", der nur für Instanzen von `Enum` definiert ist:

```
isWorkday    :: Weekday → Bool
isWorkday d =  d `elem` [Mo .. Fr]
```

5.2 Produkttypen

Das *kartesische Produkt* fasst mehrere Werte zu einem einzigen zusammen:

$$P = T_1 \times \cdots \times T_k \qquad (k \geqslant 2)$$

In Haskell sind Produkttypen als *Tupeltypen* vordefiniert. Man könnte also schreiben:

$$\textbf{type}\ P = (T_1, \cdots, T_k)$$

Beispiel 5.1 (Koordinaten und komplexe Zahlen). Koordinaten in der Ebene und komplexe Zahlen können definiert werden als

```
type Point   = (Double,Double)
type Complex = (Double,Double)
```

Typdefinitionen führen jedoch nur Abkürzungen für Typausdrücke ein, die Typen von Werten `p:: Point`, `c:: Complex` und `x::(Double,Double)` können nicht unterschieden werden; es darf also jeder für den anderen eingesetzt werden.

Will man unterschiedliche Typen mit disjunkten Wertemenge definieren, muss man schreiben

```
data Point   = Point   Double Double
data Complex = Complex Double Double
```

Werte dieser Typen werden als `p= Point 0.0 0.0` bzw. `c= Complex 0.0 -1.0` geschrieben und können wegen der vorangestellten Konstruktoren

```
Point   :: Double → Double → Point
Complex :: Double → Double → Complex
```

nicht mit einander verwechselt werden. In Haskell besteht die Konvention, für den einzigen (Wert-) Konstruktor eines Produkttyps denselben Namen zu verwenden wie für den Typ (-Konstruktor) selber. Das ist nur auf den ersten Blick verwirrend.

Beispiel 5.2 (Datum). Ein *Datum* besteht aus Tag, Monat, und Jahr. Wir könnten es als Tupeltyp definieren:

```
type Date' = (Day, Month, Year)
type Day   = Int
data Month = Jan | Feb | Mar | Apr | May | Jun
           | Jul | Aug | Sep | Oct | Nov | Dec
             deriving (Eq, Ord, Show, Read, Enum, Bounded)
type Year  = Int
```

Damit hätten wir die ähnliche Probleme wie mit der Darstellung von Koordinaten und komplexen Zahlen: Werte jedes anderen Tupeltyps, der aus denselben Komponenten besteht, könnten mit den Werten von `Date'` verwechselt werden. Deshalb ist es besser, ein Datum als algebraischen Datentyp mit einem Konstruktor zu definieren:

```
data Date  = Date Year Month Day
             deriving (Eq, Ord, Show, Read, Bounded)
```

Für den Typ `Date` wird wieder automatisch der *Konstruktor* definiert, nur ist es in diesem Fall eine dreistellige Funktion:

```
Date :: Year→ Month→ Day→ Date
```

Wie bei den Wochentagen können wir `Date` als kanonische Instanz der Klassen `Eq`, `Ord`, `Show`, `Read` und `Bounded` ableiten lassen, da der in `Date` benutzte Typ (`Int`) ebenfalls Instanz dieser Klassen ist. Da bei der Ableitung der Vergleichsoperationen die lexikographische Ordnung verwendet wird, in der zunächst die ersten Komponenten verglichen werden und, wenn die gleich sind, die weiteren, haben wir die Reihenfolge der Komponenten umgedreht, so dass die Ordnung Sinn macht.

Beispielwerte vom Typ `Date` können jetzt anhand des Konstruktors von allen anderen Tupeltypen und Produkttypen mit anderen Konstruktornamen unterschieden werden:

```
today, bloomsday, fstday :: Date
today     = Date 2009 Nov 25
bloomsday = Date 1904 Jun 16
fstday    = Date 1 Jan 1
```

Funktionen für Datum können mit Mustervergleich definiert werden. Damit kann auf die Komponenten des Datums zugegriffen werden.

```
day  :: Date→ Day
day  (Date y m d) = d
year :: Date→ Year
year (Date y m d) = y
```

Der Mustervergleich kann für einen Produkttyp wie `Date` nie scheitern. Man braucht also jeweils nur eine Gleichung für die Definition.

Die oben definierten Funktionen `day`, `year` sind *Selektoren*:

```
day  today     = 26
year bloomsday = 1904
```

Ein Spezialfall von Produkten sind die *homogenen Produkte*, die aus k Komponenten des gleichen Typs bestehen. Mathematisch werden sie oft so geschrieben:

$$P = T^k \qquad (k \geqslant 2)$$

Hierbei kann man sich fragen, weshalb k größer als 1 sein muss. "Eintupel" definieren sicher keinen sinnvollen neuen Typ; deshalb werden Klammern ohne Kommata dazwischen auch einfach nur zur syntaktischen Gruppierung in Ausdrücken verwendet.

Aber was ist mit dem Fall $k = 0$? Das *Nulltupel* () definiert den *Einheitstyp*, der nur einen einzigen Wert enthält, der – genau wie der Typ selbst – mit () bezeichnet wird. Dieser Typ kann – genau wie der Typ **void** in JAVA – überall da eingesetzt werden, wo ein Typ hingeschrieben werden muss, aber kein "interessanter" Wert erwartet wird. Wir werden später sehen, dass es auch in Haskell solche Situationen gibt.

5.3 Summentypen

Summentypen setzen Wertemengen als *disjunkte Vereinigung* andere Wertemengen zusammen:

$$S = T_1 + \cdots + T_k \qquad (k \geqslant 2)$$

Hierbei können die T_i wieder zusammengesetzte Typen, insbesondere Produkttypen sein. Summentypen können als algebraischen Datentypen mit mehreren Konstruktoren definiert werden, wobei die Konstruktoren mehrstellig sein können.

Beispiel 5.3 (Geometrische Figuren). Eine *geometrische Figur* soll sein:

- entweder ein *Kreis*, gegeben durch Mittelpunkt und Durchmesser,
- oder ein *Rechteck*, gegeben durch zwei Eckpunkte,
- oder ein *Polygon*, gegeben durch die Liste seiner Eckpunkte.

Geometrische Figuren können so definiert werden:

```
type Point = (Double, Double)
data Shape = Circ Point Double
           | Rect Point Point
           | Poly [Point]
           deriving (Eq,Show,Read)
```

Für jede Art von Figur gibt es einen eigenen Konstruktor:

```
Circ :: Point→ Double→ Shape
Rect :: Point→ Point→ Shape
Poly :: [Point]→ Shape
```

Funktionen werden wieder durch *pattern matching* definiert. Die Funktion `corners` berechnet beispielsweise die Anzahl der Eckpunkte.

```
corners :: Shape→ Int
corners (Circ _ _)       = 0
corners (Rect _ _)       = 4
corners (Poly ps)        = length ps
```

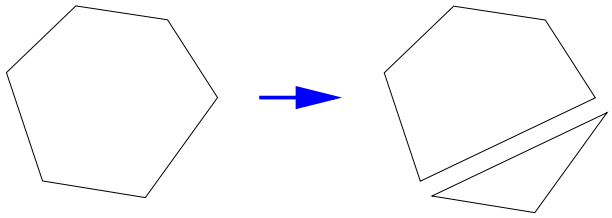
Die Translation eines Punktes wird durch Addition berechnet.

```
translation :: Point→ Point→ Point
translation (x, y) (dx, dy) = (x+ dx, y+ dy)
```

Damit kann die Verschiebung einer Figur so definiert werden:

```
move :: Shape→ Point→ Shape
move (Circ c r)    dp = Circ (translation c dp) r
move (Rect c1 c2)  dp = Rect (translation c1 dp) (translation c2 dp)
move (Poly ps)     dp = Poly (map (translation dp ps))
```

Die Berechnung der Fläche ist für Kreise und Rechtecke einfach. Für Polygone beschränken wir uns aus Bequemlichkeit halber auf *konvexe* Polygone, und reduzieren die Berechnung durch Abspalten eines Teildreiecks rekursiv auf die Flächenberechnung eines einfacheren Polygons:



```
area :: Shape→ Double
area (Circ _ d) = pi∗ d
area (Rect (x1, y1) (x2, y2)) =
      abs ((x2- x1)∗ (y2- y1))
area (Poly ps) | length ps < 3  = 0
area (Poly (p1:p2:p3:ps)) =
            triArea p1 p2 p3 +
            area (Poly (p1:p3:ps))
```

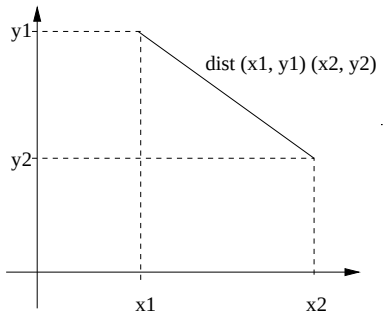
Dabei ergibt sich die Fläche eines Dreieck mit den Eckpunkten (x_1, y_1), (x_2, y_1), (x_3, y_3) als

$$A = \frac{|(x_2 - x_1)(y_3 - y_1) - (x_3 - x_1)(y_2 - y_1)|}{2}$$

Kodiert in Haskell sieht das so aus:

```
triArea :: Point→ Point→ Point→ Double
triArea (x1,y1) (x2,y2) (x3, y3) =
    abs ((x2-x1)∗(y3-y1) - (x3-x1)∗(y2-y1)) / 2
```

Die Distanz zwischen zwei Punkten ist nach Pythagoras so definiert:




```
dist :: Point→ Point→ Double
dist (x1, y1) (x2, y2) =
  sqrt((x1-x2)^2+ (y2- y1)^2)
```

Algebraische Datentypen können auch mit einer Typvariablen *parametrisiert* sein. Weiter unten werden wir Bäume als ein Beispiel für solche Typen betrachten. Hier wollen wir nur kurz einen vordefinierten Typ erwähnen, mit dem Fehler behandelt werden können.

Beispiel 5.4 (`Maybe`). Beim Programmieren kommt man oft in die Verlegenheit, dass eine Funktion in bestimmten Fehlersituationen keine gültige Ergebnisse ihres Ergebnistyps α liefern kann. Betrachten wir eine Funktion `lookup'`, die in einer Paarliste den Eintrag für einen Schlüssel sucht.

```
lookup' :: Eq a ⇒ a → [(a,b)] → b
lookup' k []      = ...         -- ??
lookup' k ((k',e):xs)
   | k == k'   = e
   | otherwise = lookup' k xs
```

Welchen Wert soll sie liefern, wenn der Schlüssel in der Liste gar nicht auftaucht? Man könnte irgendeinen Wert von `b` als den *Fehlerwert* vereinbaren; das wäre aber nicht sehr sauber programmiert. Besser wäre es, den Ergebnistyp `b` um einen Fehlerwert zu erweitern. Dies kann man für beliebige Typen mithilfe des parametrisierten Summentyps `Maybe` tun, der in Haskell folgendermaßen vordefiniert ist

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

`Nothing` repräsentiert den Fehlerwert, und der Konstruktor `Just` kennzeichnet die "richtigen" Werte. Dann können wir die Funktion so definieren:

```
lookup :: Eq a ⇒ a → [(a,b)] → Maybe b
lookup k []      = Nothing
lookup k ((k',e):xs)
   | k == k'    = Just e
   | otherwise = lookup k xs
```

Brauchen wir aber nicht, denn sie ist vordefiniert.

Der parametrisierte Summentyp schlechthin ist der vordefinierte Typ `Either a b`, der entweder Werte des Typs `a` oder des Typs `b` enthält.

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

5.4 Rekursive algebraische Datentypen

In Summentypen kann der definierte Typ auf der rechten Seite benutzt werden; dann ist der Datentyp *rekursiv*. Funktionen auf solchen Typen sind meist auch rekursiv. (Rekursive Produkttypen machen dagegen keinen Sinn, weil es dann keinen Konstruktor gäbe, der einen Anfangswert erzeugen könnte.)

Beispiel 5.5 (Arithmetische Ausdrücke). Einfache arithmetische Ausdrücke bestehen entweder aus Zahlen (Literalen), oder aus der Addition bwz. der Subtraktion zweier Ausdrücke.

```
data Expr = Lit Int
          | Add Expr Expr
          | Sub Expr Expr
          deriving (Eq,Read,Show)
```

Betrachten wir Funktionen zum Auswerten und Drucken eines Ausdrucks:

```
eval :: Expr→ Int
eval (Lit n)= n
eval (Add e1 e2) = eval e1+ eval e2
eval (Sub e1 e2) = eval e1- eval e2

print :: Expr→ String
print (Lit n) = show n
print (Add e1 e2) = "("++ print e1++ "+"++ print e2++ ")"
print (Sub e1 e2) = "("++ print e1++ "-"++ print e2++ ")"
```

An diesen Beispielen können wir ein Prinzip der *primitive Rekursion* auf Ausdrücken erkennen, das der primitiven Rekursion aus Listen entspricht, die wir in Abschnitt 3.3.1 entdeckt haben. Primitiv rekursive Funktionen über Ausdrücken definieren eine Gleichung für jeden Konstruktor:

- Für Literale, die Rekursionsverankerung, wird ein Wert eingesetzt.

- Für die rekursiven Fälle, Addition, bzw. Subtraktion, werden jeweils binäre Funktionen eingesetzt.

Kombinatoren wie `map` und *`fold`* machen nicht nur Sinn für Listen. Sie können für viele algebraische Datentypen definiert werden, insbesondere wenn sie rekursiv sind.

Beispiel 5.6 (Strukturelle Rekursion über Ausdrücken). In Beispiel 5.5 haben wir ein Prinzip der primitiven Rekursion über Ausdrücken entdeckt, dass wir jetzt in einem Kombinator kapseln können.

Für jeden Wertkonstruktor des Typs *`Expr`* müssen wir beim Falten eine Funktion entsprechenden Typs angeben:

```
data Expr = Lit Int
          | Add Expr Expr
          | Sub Expr Expr
            deriving (Eq,Read,Show)

foldE :: (Int→ a)→ (a→ a→ a)→ (a→ a→ a) → Expr→ a
foldE b a s (Lit n)     = b n
foldE b a s (Add e1 e2) = a (foldE b a s e1) (foldE b a s e2)
foldE b a s (Sub e1 e2) = s (foldE b a s e1) (foldE b a s e2)
```

Damit kann die Auswertung und die Ausgabe so definiert werden:

```
eval' :: Expr→ Int
print':: Expr→ String

eval'  = foldE id (+) (-)
print' = foldE show (\s1 s2 → "("++ s1++ "+"++ s2++ ")")
                    (\s1 s2 → "("++ s1++ "-"++ s2++ ")")
```

5.5 Bäume

Bäume sind der rekursive algebraische Datentyp schlechthin. Ein binärer Baum ist hier

- entweder leer,
- oder ein Knoten mit genau *zwei* Unterbäumen, wobei die Knoten Markierungen tragen.

```
data Tree a = Null
            | Node (Tree a) a (Tree a)
              deriving (Eq, Read, Show)
```

Einen Test auf Enthaltensein kann man wie gewohnt mit Mustervergleich definieren:

```
member :: Eq a⇒ Tree a→ a→ Bool
member Null _ = False
member (Node l a r) b =
  a == b || (member l b) || (member r b)
```

Diese Funktion benutzt ein Schema für *primitive Rekursion* auf Bäumen:

- Der leere Baum bildet den Rekursionsanfang; dann wird ein Wert zurückgeliefert.
- Im Rekursionsschritt wird für jeden Knoten aus seiner Markierung und den rekursiv bestimmten Werten für die Unterbäume der Rückgabewert berechnet.

Auf die gleiche Weise können wir Funktionen zum *Traversieren* von Bäumen definieren, die alle Markierungen in einer Liste zusammenfassen:

```
preorder, inorder, postorder :: Tree a→ [a]
preorder Null          = []
preorder (Node l a r)  = [a] ++preorder l ++preorder r

inorder  Null          = []
inorder  (Node l a r)  = inorder l ++[a] ++inorder r

postorder Null          = []
postorder (Node l a r)  = postorder l ++postorder r++ [a]
```

Auch für Bäume können Kombinatoren definiert und benutzt werden.

Beispiel 5.7 (Strukturelle Rekursion auf Bäumen). Der Kombinator `foldT` kapselt Rekursion auf Bäumen.

```
foldT :: (a→ b→ b→ b)→ b→ Tree a→ b
foldT f e Null = e
foldT f e (Node l a r) = f a (foldT f e l) (foldT f e r)
```

Damit kann der Elementtest und eine `map`-Funktion für Bäume realisiert werden.

```
member' :: Eq a⇒ Tree a→ a→ Bool
member' t x =
   foldT (\e b1 b2→ e == x || b1 || b2) False t

mapT :: (a→ b)→ Tree a→ Tree b
mapT f = foldT (flip Node ∘ f) Null
```

Dabei ist `flip` die vordefinierte Funktion, die zwei Argumente eine Funktion vertauscht. Traversierung kann dann so beschrieben werden:

```
preorder', inorder', postorder' :: Tree a→ [a]
preorder'   = foldT (\x t1 t2→ [x] ++ t1  ++ t2 ) []
inorder'    = foldT (\x t1 t2→  t1 ++ [x] ++ t2 ) []
postorder'  = foldT (\x t1 t2→  t1 ++ t2  ++ [x]) []
```

Beispiel 5.8 (geordnete Bäume). Bäume können dazu benutzt werden, Mengen effizient darzustellen. Eine Voraussetzung dafür ist, dass die Elemente der Menge geordnet sind, d.h. in der Terminologie von Haskell, dass ihr Elementtyp `a` eine Instanz der Klasse `Ord` sein muss.

Für alle Knoten `Node a l r` eines *geordneten Baum* soll gelten, dass der linke Unterbaum nur kleinere und der rechte Unterbaum nur größere Elemente als `a` enthält:

$$\forall t = \texttt{Node l a r} \texttt{member x l} \Rightarrow \texttt{x} < \texttt{a} \wedge \texttt{member x r} \Rightarrow \texttt{a} < \texttt{x}$$

Eine Konsequenz dieser Bedingung ist, dass jeder Eintrag höchstens einmal im Baum vorkommt. Der Test auf Enthaltensein lässt sich dann so vereinfachen:

```
member :: Ord a⇒ Tree a→ a→ Bool
member Null _ = False
member (Node l a r) b
  | b < a  = member l b
  | a == b = True
  | b > a  = member r b
```

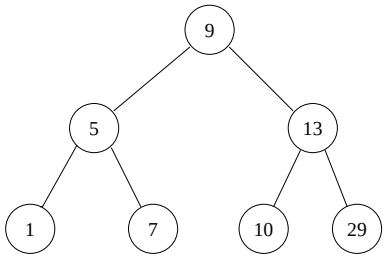
Die Funktion für ordnungserhaltendes Einfügen kann dann so definiert werden:

```
insert :: Ord a⇒ Tree a→ a→ Tree a
insert Null a = Node Null a Null
insert (Node l a r) b
  | b < a  = Node (insert l b) a r
  | b == a = Node l a r
  | b > a  = Node l a (insert r b)
```

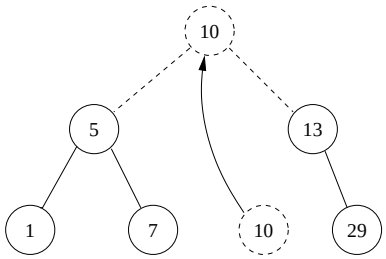
Etwas komplizierter ist das Löschen eines Elements:

```
delete :: Ord a⇒ a→ Tree a→ Tree a
delete x Null = Null
delete x (Node l y r)
  | x < y  = Node (delete x l) y r
  | x == y = join l r
  | x > y  = Node l y (delete x r)
```

Sei beispielsweise folgender Baum gegeben:



Wenn wir danach `delete t 9` ausführen, wird die Wurzel von `t` gelöscht, und die beiden Unterbäume müssen von `join` ordnungserhaltend zusammengefügt werden:



Dabei wird der Knoten *links unten* im *rechten* Teilbaum (oder der rechts unten im linken Teilbaum) die Wurzel des neuen Baums.

Wir implementieren dazu eine Funktion *splitTree*, die den Baum in den Knoten links unten und den Rest aufspaltet.

```
join :: Tree a→ Tree a→ Tree a
join xt Null = xt
join xt yt   = Node xt u nu
   where
   (u, nu) = splitTree yt
   splitTree :: Tree a→ (a, Tree a)
   splitTree (Node Null a t) = (a, t)
   splitTree (Node lt a rt) = (u, Node nu a rt)
                              where (u, nu) = splitTree lt
```

Dabei nehmen wir an, dass *xt* nicht leer ist, wenn *yt* nicht leer ist.

5.6 Zusammenfassung

Die Definition eines algebraischen Datetypen $T\,\alpha_1 \cdots \alpha_k$ hat im Allgemeinen folgende Form:

$$\begin{aligned} \textbf{data}\ T\,\alpha_1 \cdots \alpha_k = {} & K_1\ \mathit{Typ}_{1,1} \ldots \mathit{Typ}_{1,k_1} \\ & \vdots \\ \mid {} & K_n\ \mathit{Typ}_{n,1} \ldots \mathit{Typ}_{n,k_n} \\ & \textbf{deriving}\ (C_1, \ldots, C_n) \end{aligned}$$

Hierin sind die $\mathit{Typ}_{i,j}$ Typausdrücke, in denen die Typvariablen $\alpha_1 \cdots \alpha_k$ und auch T vorkommen dürfen. Die Konstruktorfunktionen

$$K_i\ ::\ \mathit{Typ}_{i,1} \to \ldots \to \mathit{Typ}_{i,k_i} \to T\,\alpha_1 \cdots \alpha_k$$

sind verschieden von einander (und verschieden von allen anderen Konstruktoren algebraischer Datentypen).

Algebraische Datentypen haben drei wichtige Eigenschaften:

1. Die Konstruktoren sind *total*: Jede Anwendung eines Konstruktors auf Argumente geeigneten Typs liefert einen Wert.
2. Sie sind *Konstruktor-erzeugt*: Ihre Werte können ausschließlich durch die (verschachtelte) Anwendung von Konstruktoren aufgebaut werden.
3. Die Werte sind *frei erzeugt*: Für verschiedene Argumente liefert die Anwendung eines Konstruktors immer auch verschiedene Werte.

Beispiel 5.9 (Nochmal Bäume). Für den Datentyp `Tree a` folgert aus den oben definierten Eigenschaften:

1. Für alle Werte v vom Typ `a` und alle Bäume vom l, r vom Typ `Tree a` ist $t =$ `Node` $v\, l\, r$ ein Baum (von Typ `Tree a`).
2. Jeder Baum $t ::$ `Tree` a hat entweder die Form $t =$ `Null` oder die Form $t =$ `Node` $v\, l\, r$, wobei v ein Wert vom Typ `a` und l, r Bäume vom Typ `Tree a` sind.
3. Für Werte $t =$ `Node` $v\, l\, r$ und $t' =$ `Node` $v'\, l'\, r'$ mit $(v, l, r) \neq (v', l', r')$ gilt auch $t \neq t'$. (Umgekehrt folgt aus $t = t'$, dass $(v, l, r) = (v', l', r')$.)

Die erste Eigenschaft macht uns Probleme, wenn wir geordnete Bäume implementieren wollen. Mit den Konstruktoren können jederzeit beliebige, also auch ungeordnete Bäume erzeugt werden. Um die Ordnung zu erhalten, müssten die Konstruktoren `Null` und `Node` *verborgen* werden, und alleine `insert` als *Pseudo-Konstruktor* benutzbar sein. Diese Operation ist total, erzeugt die geordneten Bäume aber nicht frei, denn

$$\texttt{insert}\ (\texttt{insert}\ t\ x)\ x = \texttt{insert}\ t\ x$$

Im nächsten Kapitel werden wir sehen, wie geordnete Bäume als *abstrakte Datentypen* genau so realisiert werden können.

6

Abstrakte Datentypen I

Ein Datentyp heißt *abstrakt*, wenn seine Eigenschaften allein durch die für ihn definierten Operationen festgelegt sind, während die Repräsentation seiner Werte dem Benutzer verborgen bleibt. Zur Realisierung abstrakter Datentypen – aber nicht nur dafür – braucht man Konzepte zur *Kapselung*, die Vereinbarungen zusammenfassen, und einige davon verbergen können, um so das Prinzip des *information hiding* zu realisieren.

In diesem Kapitel werden zunächst anhand des Beispiels der *geordneten Bäume* entwickeln, wie ein Datentyp in einem Modul gekapselt werden kann, wie durch Exportschnittstellen seine Hilfsdefinitionen nach außen verborgen bleiben, und wie endlich ein abstrakter Datentyp für Mengen entsteht, wenn auch die Realisierung des Datentypen verborgen wird. Dann betrachten wir die Realisierungen einiger klassische abstrakter Datentypen in Haskell: Stapel (*Stack*) und Warteschlangen (*Queue*).

6.1 Konkrete und abstrakte Datentypen

Ein *konkreter Datentyp* besteht aus der Definition eines Datentypen D und einer Menge von *Operationen* auf D. D heißt konkret, weil die Darstellung seiner Werte vorgegeben ist – in einer funktionalen Sprache geschieht dass das durch die Definition der Wertkonstruktoren für D.

Beispiel 6.1 (Rationale Zahlen). Die rationalen Zahlen $\mathbb{Q}$ sind Brüche $\frac{n}{d}$ von ganzahligen Zählern n und granzzahligen Nennern d (engl. *nominator* und *denominator*). Die arithmetischen Operationen auf rationalen Zahlen sind wie folgt definiert:

$$\frac{n}{d} + \frac{n'}{d'} = \frac{nd' + n'd}{dd'} \quad \frac{n}{d} - \frac{n'}{d'} = \frac{nd' - n'd}{dd'} \quad \frac{n}{d}\,\frac{n'}{d'} = \frac{nn'}{dd'} \quad \frac{n}{d} \Big/ \frac{n'}{d'} = \frac{nd'}{dn'}$$

Die Vergleichsoperationen sind definiert als

$$\frac{n}{d} \sim \frac{n'}{d'} = nd' \sim dn' \text{ für } \sim \in \{<, \leqslant, \neq, =, \geqslant, >\}$$

In einer funktionalen Sprache könnte man dies leicht hinschreiben:

```
type Rat = Rat Integer Integer

plus, minus, times, by :: Rat → Rat → Rat

(Rat n d) `plus`  (Rat n' d') = Rat (n * d' + n' * d) (d * d')
(Rat n d) `minus` (Rat n' d') = Rat (n * d' - n' * d) (d * d')
(Rat n d) `times` (Rat n' d') = Rat (n * n'         ) (d * d')
(Rat n d) `by`    (Rat n' d') = Rat (n * d'         ) (d * n')

lt, le, ne, eq, ge, gt :: Rat → Rat → Rat

(Rat n d) `lt` (Rat n' d') = n * d' <  n' * d
(Rat n d) `le` (Rat n' d') = n * d' ≤  n' * d
(Rat n d) `ne` (Rat n' d') = n * d' /= n' * d
(Rat n d) `eq` (Rat n' d') = n * d' == n' * d
(Rat n d) `ge` (Rat n' d') = n * d' ≥  n' * d
(Rat n d) `gt` (Rat n' d') = n * d' >  n' * d
```

Diese Implementierung macht jedoch einige Probleme, denn `Rat` hat eine andere Wertemenge als $\mathbb{Q}$:

1. Einige Werte von `Rat` repräsentieren überhaupt keine rationale Zahl: Der Nenner einer rationalen Zahl muss immer ungleich Null sein.
2. `Rat` enthält (unendlich viele Werte, die jeweils die gleiche rationale Zahl repräsentieren: $\frac{n}{d} = \frac{m \cdot n}{m \cdot d}$ für beliebige ganze Zahlen m.

Bei äquivalenten Werte nach 2. haben Nenner und Zähler einen gemeinsamen Teiler m. Für die Darstellung rationaler Zahlen reichen die Brüche $\frac{n}{d}$ aus, wo n und d *teilerfremd* sind, also nicht mehr gekürzt werden können, und wo der Nenner nicht negativ ist. Da der Nenner auch nicht Null sein darf, reichen für eine nicht redundante Darstellung von rationalen Zahlen also folgende Brüche aus:

$$\mathbb{Q} = \left\{ \frac{n}{d} \mid n, d \in \mathbb{Z}, d > 0, \gcd\ n\ d = 1 \right\}$$

Diese Wertemenge lässt sich aber mit den gängigen Tykkonstruktoren nicht definieren. wegen der Bedingung der Teilerfremdheit.

Wenn wir mit allgemeinen Paaren von ganzen Zahlen rechnen, müssen wir also vor jeder Operation überprüfen, dass keiner der Nenner Null ist, und nach jeder Operation sicherstellen, dass Zähler und Nenner teilerfremd sind. [1] Das kann man mit der Funktion `reduce` erreichen:

[1] Eigentlich ist das für die Endergebnisse nötig, denn die Definitionen der Operationen funktionieren auch so – höchstens etwas langsamer.

```
reduce :: Integer → Integer → Rat
reduce _ 0 = error "reduce: zero denominator"
reduce n d = Rat (n `div` g) (d `div` g) where g = gcd n d
```

Trotzdem bleibt das Arbeiten mit rationalen Zahlen unbequem. Eigentlich sollten sich die Benutzer von rationalen Zahlen nicht mit solchen Details iher konkreten Darstellung herumschlagen müssen. Wünschenswert wäre eine Implementierung, die die Invarianten für die Elemente von $\mathbb{Q}$ schon bei der Konstruktion "automatisch" sicherstellt, so dass sie danach bei den Operationen nicht immer überprüft werden müssen.

So eine Funktion könnte leicht implementiert werden; hier definieren wir die Selektoren (von Zähler und Nenner einer Zahl) gleich mit:

```
infixl 7  %
(%)     :: Integer → Integer → Rat

x % y  =  reduce (x * signum y) (abs y)

numerator, denominator  :: (Integral a) ⇒ Ratio a → a
numerator (Rat x  _)      =  x

denominator (Rat _ y)     =  y
```

Das hilft wenig, weil mit dem Wertkonstruktor `Rat` immer noch Zahlen wie `Rat 12 (-4)` und `Rat 1 0` konstruiert werden können.

Wir müssen die Definition des Datentyps *kapseln*, d.h. in einen Modul stecken und erreichen, dass alle Benuzter des Moduls rationale Zahlen nur noch mit der Operation `(%)`, nicht aber mehr mit `Rat` konstruieren können.

In Haskell können die Definitionen so in ein Modul gepackt werden:

```
module  Rat
  (Rat, (%), nominator, denominator, plus, minus, times, by,
        lt, le, ne, eq, ge, gt) where

... -- hier kommen die Definitionen
```

Die Namen in der Klammer werden vom Modul *exportiert*. Alle anderen bleiben nach außen *verborgen* – z.B. `reduce`. Vom Datentyp `Rat` wird nur desen Name, nicht aber dessen Wertkonstruktor gleichen Namens exportiert. In einem Modul, der `Rat` importiert, können nur diese Operationen benutzt werden. Inbesondere kann der Wertkonstruktor *nicht* benuztzt werden, und also bei der Definition von neuen Funktionen auch keine Muster des Typ `Rat` gebildet werden.

So ähnlich wie hier vorgestellt sind die rationalen Zahlen in Haskell auch im Modul `Ratio` vordefiniert. Folgende Unterschiede sind zu beachten:

- Der Typ heißt `Rational` und ist ein Synonym für `Ratio Integer`. Es können also auch rationale Zahlen über `Int` (den ganzen Zahlen, die die Hardware unterstützt) gebildet werden – auch wenn das wenig Sinn macht.
- `Ratio` und damit `Rational` instanziieren die Klassen `Eq`, `Ord`, `Num`, `Real`, `Fractional`, `RealFrac`, `Enum`, `Read` und `Show` so dass die arithmetischen Operationen und Vergleichsprädikate mit ihren vertrauten Symbolen `(+)`, `(-)`, `(*)`, `(*)`, `(<)`, `(≤)`, `(==)`, `(/=)`, `(≥)` und `(>)` bneutzt werden können.
- Zusätzlich sind noch einige Konversionen mit ganzen und Fließpunktzahlen definiert.

Details finden sich in der Bibliothel `Ratio`.

6.2 Kapselung

Im vorigen Kapitel haben wir geordnete Bäume mit den passenden Operationen darauf implementiert (vgl. Beispiel 5.8). Es könnte sich lohnen, dies als ein wiederverwendbares Stück Software in einem Modul zu kapseln. In HASKELL dienen Module nicht nur zur Kapselung, sondern definieren auch Übersetzungs-Einheiten. Ein Modul wird also getrennt übersetzt.

Beispiel 6.2 (Gekapselte geordnete Bäume). Die Operationen `insert` und `delete` können zusammen mit dem Datentyp `Tree` in einen Modul geschrieben und in einer Datei `OrderedTree.hs` abgespeichert werden.

```
module OrderedTree where

data Ord a ⇒ Tree a = Null | Node (Tree a) a (Tree a)
                      deriving (Eq, Read, Show)

insert :: Ord a⇒ Tree a→ a→ Tree a
insert Null a = Node Null a Null
insert (Node l a r) b | b < a  = Node (insert l b) a r
                      | b == a = Node l a r
                      | b > a  = Node l a (insert r b)

delete :: Ord a⇒Tree a →  a→ Tree a
delete Null x = Null
delete (Node l y r) x | x < y  = Node (delete l x) y r
                      | x == y = join l r
                      | x > y  = Node l y (delete r x)

join ::  Ord a ⇒ Tree a→ Tree a→ Tree a
join xt Null = xt
join xt yt   = Node xt u nu
```

```
  where
  (u, nu) = splitTree yt
  splitTree :: Ord a ⇒ Tree a→ (a, Tree a)
  splitTree (Node Null a t) = (a, t)
  splitTree (Node lt a rt) = (u, Node nu a rt)
                               where (u, nu) = splitTree lt
```

Diesen Modul können wir in einem anderen Modul importieren:

```
module Use where
import OrderedTree
...
```

Danach können wir die Definitionen des Moduls benutzen, als stünden sie in `Use` selber.

Oft werden wir nicht alle Definitionen eines Moduls exportieren wollen: Die Operation `join` ist z. B. eine Hilfsfunktion, die nur lokal benutzt werden sollte. Wollen wir sie im Modul `OrderedTree` *verbergen*, müssen wir in seinem Kopf angeben, welche Definitionen exportiert werden sollen. Ändern wir also den Modulkopf wie folgt:

```
module Orderedtree (Tree(..), -- a type with constructors
                    -- Null :: Tree a
                    -- Node :: Tree a -> a -> Tree a -> Tree a
                   member,   -- Ord a=> Tree a -> a-> Bool
                   insert,   -- Ord a=> Tree a -> a-> Tree a
                    delete   -- Ord a=> Tree a -> a-> Tree a
                   ) where
```

Dann werden alle Definitionen bis auf das ausgelassene `join` exportiert. (In der Exportliste dürfen nur die Namen der exportierten Definitionen stehen; deshalb schreiben wir ihre Typen dahinter, damit im Kopf des Moduls alle wesentlichen Informationen für seine Benutzung beisammen stehen. Die Zeichen "`(..)`" hinter dem Typnamen `Tree` bedeuten, dass seine Wertkonstruktoren `Null` und `Node` zusammen mit dem Typnamen exportiert werden sollen. Das kann dazu führen, das im Modul `Use` mithilfe der Konstruktoren "ungeordnete" Bäume wie

```
t = Node (Node Null 4 Null) 3 (Node Null 2 Null)
```

konstruiert werden. Dann funktionieren Aufrufe wie `delete t 4` oder `insert t 2` nicht richtig, weil die Invariante für geordnete Bäume verletzt ist.

Dies kann man verhindern, indem man die Konstruktoren `Null` und `Node` verbirgt und `Tree` zu einem *abstrakten Datentyp* macht, der nur durch die expliziten Operationen definiert ist. Dies kann man durch Weglassen von "`(..)`" nach `Tree` in der Export-Schnittstelle erreichen. Leider stellt sich dann das nächste (kleine) Problem: Wie können die Werte des Typs ohne die Wertkonstruktoren aufgebaut werden? Die Funktion `insert` fügt zwar ein Element in einen Baum ein, aber es gibt keine Möglichkeit, einen leeren Baum (`Null`)

zu erzeugen. Wir definieren dafür eine Konstante *empty* und exportieren sie. Gleichzeitig führen wir auch noch eine weitere Operation *enumeration* ein, die alle Einträge eines geordneten Baumes aufzählt, sowie ein Prädikat *contains*, das feststellt, ob ein Element in einem Baum bereits vorhanden ist.

```
module OrderedTree (Tree,        -- an abstract type
                    empty,       -- Tree a
                    isEmpty,     -- Tree a -> Bool
                    insert,      -- Ord a => Tree a -> a-> Tree a
                    delete,      -- Ord a => Tree a -> a-> Tree a
                    contains,    -- Ord a => Tree a -> a-> Bool
                    enumeration -- Ord a => Tree a -> [a]
                   ) where

  -- Definition von insert und delete

empty :: Tree a
empty =  Null

isEmpty :: Tree a → Bool
isEmpty Null = True
isEmpty _    = False

contains :: Ord a⇒ Tree a→ a→ Bool
contains Null _ = False
contains (Node l a r) b =  (b < a && contains l b)
                        ||  b == a
                        || ( b > a  && contains r b)

enumeration :: Ord a ⇒ Tree a→ [a]
enumeration  Null        = []
enumeration (Node l a r) = (enumeration l) ++ [a] ++ (enumeration r)

...
```

Nun definiert *OrderedTree* einen *abstrakten Datentyp*, denn jedes Modul, das diesen Typ importiert, kennt von *Tree* nur den Namen und die exportierten Operationen, weiß aber nichts über die Darstellung seiner Werte.

Das hat einerseits den kleinen Nachteil, dass in einem importiernden Modul weitere Funktionen auf *Tree* nur als Kombination der exportierten Operationen definiert werden können und nicht durch primitive Rekursion über den (dort nicht bekannten) Wertkonstruktoren.

Andererseits hat es den großen Vorteil, dass jeder importierende Modul nur noch geordnete Bäume konstruieren kann (mit *empty* und `insert`), so dass die Operationieren immer funktionieren. Außerdem kann die Implementierung

von `OrderedTree` geändert werden, ohne dass dies in den importierenden Modulen bemerkt wird, solange die Operationen das Gleiche tun. Dies ist eine wesentliche Voraussetzung für Programmentwicklung im Großen.

6.3 Klassische abstrakte Datentypen

In den folgenden Unterabschnitten werden wir zwei bekannte abstrakte Datentypen in HASKELL definieren.

6.3.1 Speicher

In diesem Abschnitt betrachten wir ein sehr einfaches Beispiel für einen abstrakten Datentyp, bei dem die Schnittstelle auf verschiedene Arten realisiert werden kann.

Beispiel 6.3 (Speicher). Ein Speicher ist ein Datentyp `Store a b`, der parametrisiert ist über einem *Indextyp* `a` (der vergleichbar oder besser noch total geordnet sein sollte) und mit einem *Wertetyp* `b`. Der Speicher hat drei Operationen:

- Die Konstruktor-Operation `initial` ::`Store a b` schafft einen leeren Speicher.
- Die Selektor-Operation `value` ::`Store a b`→`a`→ `Maybe b` liest den unter einem Index abgelegten Wert. Ihr Ergebnistyp ist `Maybe b`, weil der Wert undefiniert sein könnte.
- Die Operation `update` ::`Store a b`→`a` → `b`→ `Store a b` setzt den Wert für einen Index.

Wir schreiben den abstrakten Datentyp als einen Modul `Store` in eine Datei `Store.hs` und versehen den Modul mit folgender Export-Schnittstelle:

```
module Store(Store,   -- an abstract type
             initial, -- Store a b
             value,   -- Eq a => Store a b-> a-> Maybe b
             update,  -- Store a b-> a-> b-> Store a b
            ) where
```

Die semantischen Eigenschaften, die wir von einem Speicher erwarten, sind damit noch nicht beschrieben. Informell könnten wir sie so angeben:

1. Die Funktion `value` liefert just denjenigen Wert zurück, der für einen Index x zuletzt mit `update` abgespeichert wurde.
2. Wurde bisher kein Wert für x abgespeichert, ist dies ein Fehler.

Formal könnten wir den Zusammenhang der Operationen durch *algebraische Gleichungen* etwa so beschreiben

$$value\ \ initial = Nothing \tag{6.1}$$

$$value(update\ \ s\ a\ b)\ x = \begin{cases} Just\ b & \text{wenn } x = a \\ value\ \ s\ x & \text{sonst} \end{cases} \tag{6.2}$$

$$update(update\ \ s\ a\ b)\ a\ b' = update\ \ s\ a\ b' \tag{6.3}$$

Das geht aber nicht in Haskell, oder allenfalls als Kommentar. Mindestens eine informelle Beschreibung der Eigenschaften gehört aber zu jeder Schnittstelle dazu, damit ein abstrakter Datentyp korrekt benutzt werden kann, ohne die Implementierung zu kennen.

Die Schnittstelle kann nun implementiert werden, und zwar im Verborgenen. Insbesondere können wir *verschiedene* Implementierungen entwickeln und eine durch die andere ersetzen, ohne das die Benutzer des abstrakten Datentypen es bemerken.

Die *erste Implementierung* stellt den Speicher als Paarliste dar. (Der Typ muss als Datentyp mit als **data** oder **newtype** definiert werden, wenn seine Konstuktoren verborgen werden sollen, nicht mit **type**.)

```
data Store a b = St [(a, b)] deriving Show
```

Der leere Speicher wird als leere Liste dargestellt. Die Funktion `update` hängt ein neues Index-Werte-Paar vorne an die Liste an, und `value` sucht nach dem ersten Paar mit passenden Index. (Deshalb ist es unwichtig, ob `update` ein schon abgespeichertes Paar oder ein neues einträgt.)

```
initial  = St []

value (St       []  ) x = Nothing
value (St ((a,b):st)) x
            | a == x    = Just b
            | otherwise = value (St st) x

update (St ls) a b  = St ((a, b): ls)
```

Andere Implementierungen des Speichers sind denkbar: Geordnete Listen, Bäume oder sogar *Funktionen* – aber dass Funktionen auch Daten sein können, werden wir erst im nächsten Kapitel sehen.

Eine Schnittstelle kann also auf verschiedene Arten implementiert werden, ohne dass dies für den Benutzer relevant ist, weil er den Datentyp nur über die Operationen in der Schnittstelle benutzt.

Wenn wir den Modul `Store` in einem Modul `Main` benutzen wollen, müssen wir schreiben

```
module Main where
import Store
```

Im Rumpf von `Main` können wir nun die exportierten Namen von `Store` benutzen, um Werte vom Typ `Store a b` anzulegen und zu bearbeiten. Außer

diesen Namen wissen wir nichts über die Repräsentation des Speichers, insbesondere kennen wir seinen Wertkonstruktor und seinen Komponententyp nicht und können damit auch kein *pattern matching* benutzen, um weitere Funktionen auf dem Speicher zu definieren. Dazu können wir nur die exportierten Funktionen benutzen.

Deshalb ist es wichtig, für jeden abstrakten Datentyp einen *vollständigen* Satz von Operationen zu definieren:

- *Konstruktoren* wie `initial` und `update`, die Werte aufbauen.
- *Selektoren* wie `value`, die auf Komponenenten des Datentyps zugreifen.
- Ggf. *Prädikate*, die Eigenschaften des Datentyps bestimmen. Für den Speicher hätte man beispielsweise eine Funktion `isStored` vorsehen können, die ermittelt, ob für einen Index ein Wert abgespeichert ist. In der vorliegenden Schnittstelle kann man diese Funktion jedoch aus der Funktion `value` ableiten:

```
isStored :: Eq a ⇒ Store a b → a → Bool
isStored st x = value st x /= Nothing
```

6.3.2 Stapel und Warteschlangen

Stapel (oder "Keller", engl. *stacks*) und Warteschlangen (*queues*) sind die abstrakten Datentypen *schlechthin*. Wir stellen die Schnittstellen in einer Tabelle einander gegenüber:

Bestandteil	*Stack*	*Queue*
Typ	`St a`	`Qu a`
Initialwert	`empty` :: `St a`	`empty` :: Qu a
Wert eintragen	`push` :: `St a` → `a` → `St a`	`enq` :: `Qu a` → `a` → `Qu a`
Wert lesen	`top` :: `St a` → `a`	`first` :: `Qu a` → `a`
Wert löschen	`pop` :: `St a` → `St a`	`deq` :: `Qu a` → `Qu a`
Leerheitstest	`isEmpty` :: `St a` → `Bool`	`isEmpty` :: `Qu a` → `Bool`

Während ihre Signaturen annähernd gleich sind, haben Stapel und Schlange eine unterschiedliche Semantik: Der erste realisiert das Prinzip "*last in, first out*", die zweite das Prinzip "*first in, first out*". Dementsprechend unterscheiden sich auch ihre Implementierungen.

Beispiel 6.4 (Implementation eines Stapels als Liste). Dies ist sehr einfach, weil sich die Strukturen von Listen und Stapeln eins zu eins entsprechen: Der Konstruktor `empty` ist die leere Liste `[]`, der Konstruktor `push` ist der Listenkonstruktor `(:)`, die Selektoren `top` und `pop` entsprechen den Listenselektoren `head` und `tail` und das Prädikat `isEmpty` entspricht dem Leerheits-Prädikat `null`.

```
module Stack (St, empty, push, pop, top, isEmpty) where

data St a= St [a]  deriving (Show, Eq)

empty :: St a
empty =  St []

push :: St a → a → St a
push (St s) a = St (a:s)

top  :: St a→ a
top  (St [])  = error "Stack: top on empty stack"
top  (St s)   = head s

pop  :: St a→ St a
pop  (St [])  = error "Stack: pop on empty stack"
pop  (St s)   = St (tail s)

isEmpty :: St a→ Bool
isEmpty (St s)= null s
```

Beispiel 6.5 (Implementation von `Queue`). Die einfachste Darstellung einer Schlange ist ebenfalls eine Liste. Weil neue Elemente immer hinten angefügt werden müssen, hätte dann `enq` linearen Aufwand. Deshalb repräsentieren wir die Schlange als ein Paar von Listen:

- Die erste Liste enthält die zu entnehmenden Elemente.
- Die zweite Liste enthält die hinzugefügten Elemente in umgekehrter Reihenfolge.

Die Schlange ist leer, wenn beide Listen leer sind. Bei den "selektierenden" Operationen `first` und `deq` wird überprüft, ob die erste Liste leer ist; nötigenfalls wird dann die zweite Liste umgekehrt in die erste Liste gesteckt.

Damit sieht der Modul so aus:

```
module Queue(Qu, empty, isEmpty,
             enq, first, deq) where

data Qu a = Qu [a] [a]

empty :: Qu a
empty = Qu [] []

isEmpty :: Qu a→ Bool
isEmpty (Qu xs ys) = null xs && null ys
```

```
enq   :: Qu a → a→ Qu a
enq (Qu xs ys) x = Qu xs (x:ys)

first :: Qu a→ a
first (Qu (x:_) _ ) = x
first (Qu  []   ys) = first (Qu (reverse ys) [])
first (Qu  []   []) = error "Qu:␣first␣of␣empty␣Q"

deq   :: Qu a→ Qu a
deq (Qu (_:xs) ys) = Qu xs ys
deq (Qu  []    ys) = deq (Qu (reverse ys) [])
deq (Qu  []    []) = error "Qu:␣deq␣of␣empty␣Q"
```

Betrachten wir eine Sequenz von Operationen, die damit definierte Schlange und die Repräsentation mit zwei Listen:

Operation	Queue	Darstellung
`empty`		`Qu [], []`
`enq 9`	9	`Qu [9], []`
`enq 4`	$9 \rightarrow 4$	`Qu [9], [4]`
`deq`	4	`Qu [4], []`
`enq 7`	$4 \rightarrow 7$	`Qu [4], [7]`
`enq 5`	$4 \rightarrow 7 \rightarrow 5$	`Qu [4], [5, 7]`
`deq`	$7 \rightarrow 5$	`Qu [7, 5], []`
`deq`	7	`Qu [5], []`
`deq`		`Qu [], []`

Immer wenn die erste Liste leer ist, füllt `deq` sie wieder mit hinzugefügten Elementen auf.

6.4 Haskells Eck

6.4.1 Module

Module im Allgemeinen können zu nicht nur zur Definition von abstrakten Datentypen benutzt werden.

Module bilden die *Übersetzungseinheiten* eines HASKELL-Programms, in dem Teile des Programms als Datei abgespeichert und von einem HASKELL-System wie `hugs` übersetzt werden können. Jeder Modul kapselt eine Menge von Typen, Funktionen und Klassen. Ein Modul kann in einer Exportschnittstelle die Klassen, Typen und Funktionen nennen, die beim Import des Moduls sichtbar sind. Er kann andere Module ganz oder teilweise importieren.

Ein Modul in der Datei ⟨*Name*⟩`.hs` (oder ⟨*Name*⟩`.lhs`) ist so aufgebaut:

module *Name* [(*exportierte Bezeichner*)] **where** *Importe RumpfDefinitionen*

Wenn die geklammerte Liste der exportierten Bezeichner fehlt, werden *alle* Definitionen des Modulrumpfes exportiert; bei Typdefinitionen einschließlich ihrer Konstruktoren.

Wenn ein mit **newtype** oder **data** definierter Typkonstruktor T in der Liste der exportierten Definitionen auftaucht, wird nur der Name von T exportiert. Sollen auch die Wertkonstruktoren des Typs exportiert werden, muss man dies mit $T(..)$ angeben; ein anderer Modul, der T importiert, kennt dann die Repräsentation von T und kann Funktionen über T mit Mustervergleich (*pattern matching*) definieren. Die Repräsentation von mit mit **type** definierten Typsynonymen ist immer sichtbar.

Importe von Modulen haben die Form

import *Modulname* [**as** *Modulname*] [**qualified**] [**hiding**] (*Bezeichner*)

Ohne die geklammerte Liste "(*Bezeichner*)" wird alles importiert. Mit **as** wird der Modulname innerhalb des importierenden Moduls umbenannt. Wird **qualified** angegeben, muss jeder importierte Name n im importierenden Modul *qualifiziert* werden, also in der Form "*Modulname.n*" benutzt werden.Mit **hiding** wird erreicht, dass die in Klammern genannten Bezeichner *versteckt* werden:

```
import Prelude hiding (foldr)
foldr f e ls = ...
```

Mit **qualified** und **hiding** kann man also Namenskonflikte zwischen den Namen von Modulen auflösen.

7

Mehr Abstrakte Datentypen

Nachdem wir im letzten Kapitel die grundlegenden Eigenschaften abstrakter Datentypen beschrieben haben und dabei eher einfache Beispiele betrachtet haben, soll es in diesem Kapitel um Beispiele gehen, die für das praktische Programmieren von großer Bedeutung sind: Mengen, Felder und ...

7.1 Mengen als abstrakte Datentypen

Stapel und Schlange sind gut und schön, aber auch sehr einfach. Ein etwas nützlicherer abstrakter Datentyp ist die *Menge*, also eine Ansammlung von Elementen, in der kein Element doppelt auftritt und die Ordnung der Elemente nicht relevant ist. Als Basis-Operationen auf Mengen betrachten wir:

- die leere Menge,
- das Einfügen eines Elements,
- das Löschen eines Elements,
- das Leerheits-Prädikat,
- das Enthaltenseins-Prädikat, und
- eine Operation, die die Elemente einer Menge als Liste aufzählt.

Damit lassen sich dann weitere Operationen, wie Vereinigung, Durchschnitt und Mengendifferenz, Teilmengen-Prädikat usw. leicht definieren.

Mengen können als ein Typ `Set a` definiert werden, dessen Elemente mindestens vergleichbar sein müssen (um doppelte Einträge festzustellen). Für effizientere Implementierungen von Mengen werden wir dann geordnete Elementtypen brauchen. Die Schnittstelle des abstrakten Datentyps könnte so aussehen:

```
module Set (Set,     -- abstract datatype
  empty,             -- Set a
  isEmpty,           -- Set a-> Bool
  insert, delete,    -- Ord a => a-> Set a-> Set a
```

```
  contains,          -- Ord a => Set a-> a-> Bool
  enumeration        -- Set a-> [a]
  ) where
{--  properties  ------------------------------------
  isEmpty  empty       = True
  isEmpty (insert _ _) = False
  x /= y ⇒
  insert x (insert y s)   = insert y (insert x  s)
  insert x (insert x s)   = insert x s
  delete x  empty         = empty
  delete x (insert x s)   = s
  contains  empty       x = False
  contains (insert x _) x = True                -----}
```

7.1.1 Mengen als Listen

Die einfachste Implementierung von Mengen sind sicher die Listen. Doppelte Einträge werden beim Einfügen durch Aufruf der vordefinierten Funktion `nub:: Eq a ⇒ [a] → [a]` entfernt, die aus der Bibliothek `List` importiert werden muss. Die anderen Mengenoperationen können eins zu eins durch entsprechende Konstruktoren und vordefinierte Funktionen auf Listen implementiert werden. Die Operation `(\\) ::(Eq a) ⇒[a] → [a] → [a]` implementiert Listendifferenz; l_1 `\\` ℓ_2 entfernt also jeweils ein Vorkommen jedes Elements von ℓ_2 aus ℓ_1.[1]

Daher erfüllen alle Listen, die eine Menge darstellen, die Invariante

$$\forall \ell \in \texttt{Set a} : nub\ \ell = \ell \tag{7.1}$$

```
import List (intersperse, nub, (\\))

newtype Set a = Set [a]

empty :: Set a
empty = Set []

isEmpty :: Set a → Bool
isEmpty (Set xs) = null xs

insert :: Eq a ⇒ Set a → a → Set a
insert (Set xs) x = Set (nub (x: xs))

delete :: Eq a⇒ Set a → a → Set a
```

[1] Statt des Schlüsselworts **newtype** können wir hier **data** lesen; genaueres hierzu steht in Abschnitt 7.3.2.

```
delete (Set xs) x = Set (xs \\ [x])

contains :: Eq a ⇒ Set a→ a→ Bool
contains (Set xs) x = x `elem` xs

enumeration :: Set a→ [a]
enumeration (Set xs) = xs
```

Diese Implementierung ist nicht effizient, denn `insert`, `delete` und *contains* haben linearen Aufwand.

7.1.2 Mengen als geordnete Bäume

Eine etwas bessere Implementierung wären die *geordneten Bäume*, die wir in Abschnitt 6.2 entwickelt haben.

Dazu importieren wir *OrderedTree*. Weil dieses Modul eine Ordnung auf den Elementen braucht, müssen wir das auch für den Modul *SetAsOrderedTree* fordern

```
module SetAsOrderedTree (Set, -- an abstract type
  empty,           -- Set a
  isEmpty,         -- Set a-> Bool
  insert, delete,  -- Ord a => a-> Set a-> Set a
  contains,        -- Ord a => Set a-> a-> Bool
  enumeration      -- Set a-> [a]
  ) where

import qualified OrderedTree as T
```

Wir implementieren den Modul *qualifiziert*, so dass seine exportierten Namen mit dem Modulnamen qualifiziert werden müssen, weil sonst Namenskonflikte entstehen würden. Wir benennen *OrderedTree* in *T* um, damit die Namen handlich bleiben.

Wir definieren *Set* wieder als **newtype** und können danach die Operationen für geordnete Bäume eins zu eins übernehmen:

```
newtype Ord a ⇒ Set a = Set (T.Tree a)

empty =  Set T.empty

isEmpty (Set t)  = T.isEmpty t

insert x (Set t) = Set (T.insert t x)

delete x (Set t) = Set (T.delete t x)

contains (Set t) x = T.contains t x
```

```
enumeration (Set t) = T.enumeration t

instance (Ord a,Show a) ⇒ Show (Set a) where
  show t = "{" ++ concat (intersperse "," elements) ++ "}"
           where elements :: [String]
                 elements =  [show x | x ← enumeration t]
```

Wir bilden eine Instanz von `Show`, damit Mengen "zünftig" als Zeichenketten dargestellt werden können.

Geordnete Bäume führen dazu, dass die Operationen `insert`, `delete` und `contains` *im Mittel* den Aufwand $\mathcal{O}(\log n)$ haben, nämlich dann, wenn alle Knoten im Baum annähernd gleich hohe Teilbäume haben. Werden die Elemente aber in einer ungünstigen Reihenfolge eingefügt, kann der Baum zu einer Liste degenerieren, in dem die linken bzw. rechten Teilbäume der Knoten jeweils leer sind. (Das passiert, wenn die Elemente geordnet bzw. umgekehrt geordnet eingefügt werden.) Im *schlechtesten Fall* haben die Operationen dann doch wieder linearen Aufwand. Im nächsten Abschnitt werden wir das weiter verbessern.

7.1.3 AVL-Bäume

Das Degenerieren von geordneten Bäumen zu Listen kann verhindert werden, wenn die Knoten des Baumes nötigenfalls *balanciert* werden. Wir entwickeln eine Implementierung von *AVL-Bäumen*, die nach Adelson-Velskii und Landis [AVL62] benannt sind. An der Schnittstelle ändert sich – bis auf die Namen des Moduls und des Typs nichts:

```
module AVLTree (AVLTree,
  empty,            -- AVLTree a
  isEmpty,          -- AVLTree a-> Bool
  insert, delete,   -- Ord a => a-> AVLTree a-> AVLTree a
  contains,         -- Ord a => a-> AVLTree a-> Bool
  enumeration       -- AVLTree a-> [a]
  ) where
```

Ein AVL-Baum ist *ausgeglichen*, wenn der Höhenunterschied zwischen den zwei Unterbäumen eines Knotens höchstens eins beträgt und diese Unterbäume selbst auch ausgeglichen sind. Für alle Knoten $n =$`Node` $l\ a\ r$ in so einem Baum gilt also die Invariante

$$\mathit{balanced}\ n \iff |\mathit{height}\ l - \mathit{height}\ r| \leqslant 1 \wedge \mathit{balanced}\ l \wedge \mathit{balanced}\ r$$

Weil die Höhe eines Baums an vielen Stellen gebraucht wird, wird sie in jedem Knoten abgespeichert.

```
data AVLTree a = Null
```

```
  | Node Int (AVLTree a) a (AVLTree a)
    deriving Eq
```

Alle Operationen müssen Ausgeglichenheit als Invariante bewahren. Beim Einfügen und Löschen müssen dazu ggf. Unterbäume *rotiert* werden.

Zunächst jedoch die weitgehend unveränderten Operationen: die leere Menge, Leerheits- und Enthaltenseins-Prädikat und die Funktion zur Aufzählung der Elemente.

```
empty :: AVLTree a
empty =  Null

isEmpty      :: AVLTree a → Bool
isEmpty Null =  True
isEmpty _    =  False

contains :: Ord a⇒ AVLTree a → a→ Bool
contains Null e           = False
contains (Node _ l a r) e = a == e || e < a && contains l e
                                   || e > a && contains r e

enumeration :: AVLTree a→ [a]
enumeration Null           = []
enumeration (Node _ l a r) = enumeration l ++ [a] ++ enumeration r
```

Für die restlichen Operationen brauchen wir einige Hilfsfunktionen. Die Funktion *mkNode* legt einen Knoten an und setzt seine Höhe mit der Funktion *ht*, die die Höhe nicht rekursiv berechnet, sondern auf den in den Wurzelknoten der Unterbäume gespeicherten Wert zugreift.

```
mkNode :: AVLTree a→ a→ AVLTree a→ AVLTree a
mkNode l n r = Node h l n r where h = 1+ max (ht l) (ht r)

ht :: AVLTree a→ Int
ht Null           = 0
ht (Node h _ _ _) = h
```

Abbildung 7.1 stellt eine Situation dar, in der ein Baum nicht ausgeglichen ist. Das kann durch Löschen oder Einfügen passieren. Dann muss die Ausgeglichenheit durch Rotieren der Unterbäume wiederhergestellt werden.

Dafür gibt es zwei Basisoperationen, *Linksrotation* und *Rechtsrotation*, die in den Abbildungen 7.2 und 7.3 illustriert sind.

```
rotl :: AVLTree a→ AVLTree a
rotl (Node _ xt y (Node _ yt x zt)) =
  mkNode (mkNode xt y yt) x zt

rotr :: AVLTree a→ AVLTree a
```

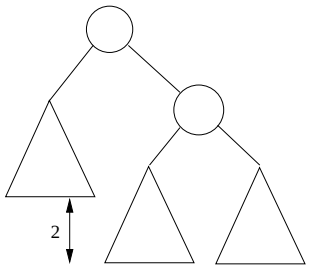


Abb. 7.1. ein nicht ausgeglichener Baum

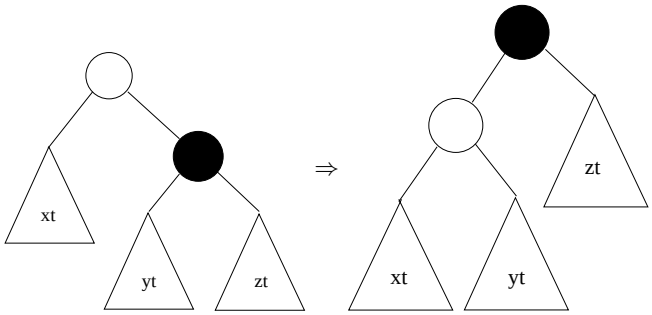


Abb. 7.2. Linksrotation

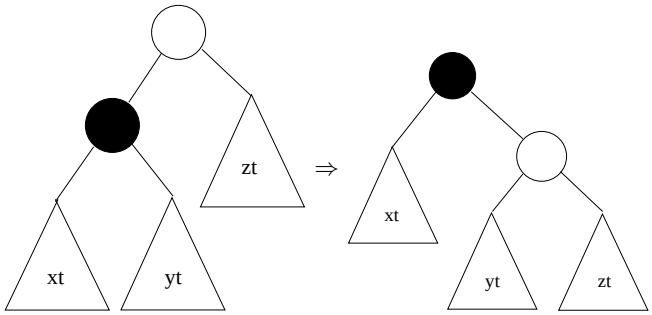


Abb. 7.3. Rechtsrotation

```
rotr (Node _ (Node _ xt y yt) x zt) =
  mkNode xt y (mkNode yt x zt)
```

Die Funktionen `rotl` und `rotr` müssen dazu benutzt werden, Ausgeglichenheit sicherzustellen. Folgende Fälle müssen unterschieden werden:

1. Der Unterbaum *rechts außen* ist zu hoch. Dann hilft einfache Linksrotation, siehe Abbildung 7.4.
2. Der Unterbaum *rechts innen* ist zu hoch oder gleich hoch wie der rechts außen. Dann muss der rechte Unterbaum durch Rechtsrotation ausgegli-

chen werden, bevor der Baum selber durch Linksrotation ausgeglichen werden kann. siehe Abbildung 7.5.

Die spiegelbildlichen Fälle 3 und 4, in denen der linke Baum zu hoch ist, können spiegelbildlich durch Rechtsrotation gelöst werden, wobei vorher der linke Unterbaum links rotiert werden muss, wenn dessen innerer Unterbaum zu hoch war.

Wir brauchen noch zwei weitere Hilfsfunktionen, um die Ausgeglichenheit sicherzustellen: `bias t` berechnet die *Balance* eines Baumes `t`, und `mkAVL lt x rt` konstruiert neuen AVL-Baum mit Knoten `x` (unter der Voraussetzung, dass der Höhenunterschied zwischen `lt` und `rt` höchstens zwei beträgt).

Dabei müssen folgende Fälle unterschieden werden:

1. Rechter Teilbaum zu hoch:
   a) Unterbaum rechts ist außen zu hoch:$bias < 0$
   b) Unterbaum rechts ist innen zu hoch: $bias \geq 0$
2. Linker Teilbaum zu hoch:
   a) Unterbaum links ist innen zu hoch: $bias > 0$
   b) Unterbaum links ist außen zu hoch: $bias \leq 0$

```
bias :: AVLTree a→ Int
bias (Node _ lt _ rt) = ht lt - ht rt

mkAVL :: AVLTree a→ a→ AVLTree a→ AVLTree a
mkAVL lt x rt
  | hl+1< hr  = if bias rt < 0
                then rotl (mkNode lt x rt)
                else rotl (mkNode lt x (rotr rt))
  | hr+1< hl  = if bias lt > 0
                then rotr (mkNode lt x rt)
                else rotr (mkNode (rotl lt) x rt)
  | otherwise = mkNode lt x rt
                where hl = ht lt
                      hr = ht rt
```

Nun können wir Einfügen und Löschen implementieren, wobei ggf. ausgeglichen wird.

```
insert :: Ord a⇒ a→ AVLTree a→ AVLTree a
insert a Null = mkNode Null a Null
insert x (Node n l a r)
  | x <  a = mkAVL (insert x l) a r
  | x == a = Node n l a r
  | x >  a = mkAVL l a (insert x r)

delete :: Ord a⇒ a→ AVLTree a→ AVLTree a
```

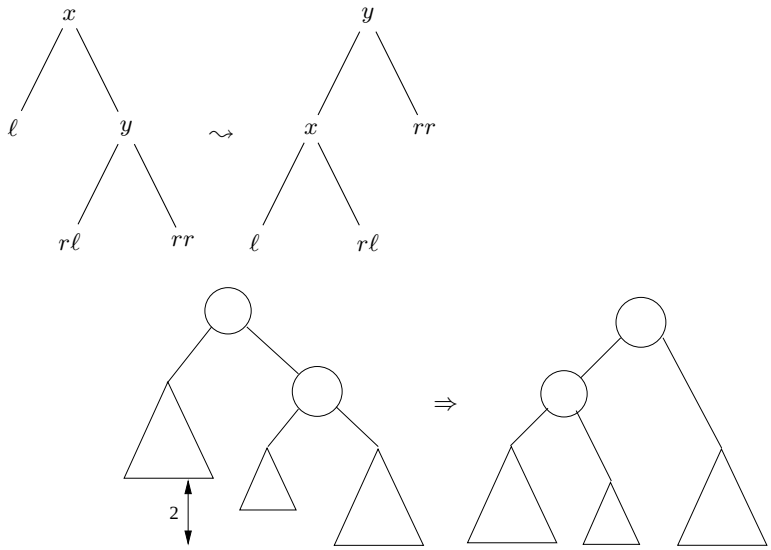


Abb. 7.4. Fall 1a. Unterbaum rechts ist außen zu hoch

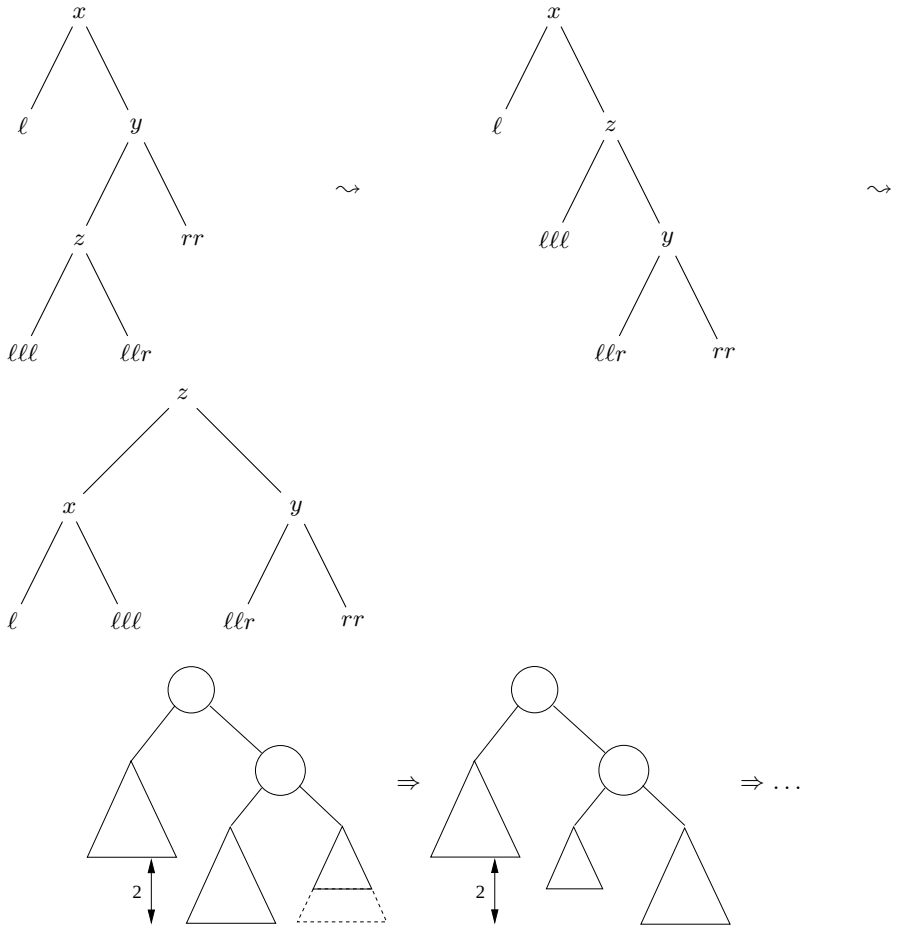


Abb. 7.5. Fall 1b. Unterbaum rechts ist innen zu hoch

```
delete x Null = Null
delete x (Node h l y r)
  | x <  y = mkAVL (delete x l) y r
  | x == y = join l r
  | x >  y = mkAVL l y (delete x r)
```

Die Verwendung *mkAVL* garantiert die Invariante (Ausgeglichenheit).

Die Hilfsfunktion `join` fügt zwei Bäume ordnungserhaltend zusammen, wie wir es schon bei geordneten Bäumen im vorigen Abschnitt gesehen haben.

```
join ::  AVLTree a→ AVLTree a→ AVLTree a
join lt Null = lt
join lt yt   = mkAVL lt u nu where
  (u, nu) = splitTree yt
  splitTree :: AVLTree a→ (a, AVLTree a)
  splitTree (Node h Null a t) = (a, t)
  splitTree (Node h lt a rt) =
    (u, mkAVL nu a rt) where
       (u, nu) = splitTree lt
```

Beispiel 7.1 (Strukturelle Rekursion über AVL-Bäumen). Auch über AVL-Bäumen kann die primitive Rekursion als Kombinator extRahiert werden:

```
foldAVL :: (Int→ b→ a→ b→ b)→ b→ AVLTree a→ b
foldAVL f e Null = e
foldAVL f e (Node h l a r) =
   f h (foldAVL f e l) a (foldAVL f e r)
```

Das Aufzählen der Menge kann dann als Inorder-Traversion (via *fold*) definiert werden.

```
enum :: Ord a⇒ AVLTree a→ [a]
enum = foldAVL (\h t1 x t2→ t1++ [x]++ t2) []
```

Das Enthaltensein- Prädikat kann ebenfalls mit *fold* definiert werden.

```
elem :: Ord a⇒ a→ AVLTree a→ Bool
elem e = foldAVL (\h b1 x b2→ e == x || b1 || b2) False
```

7.1.4 Mengen als AVL-Bäume

Nun können wir Mengen als AVL-Bäume realisieren.

```
module SetAsAVLTree (Set, -- an abstract type
  empty,           -- => Set a
  isEmpty,         -- => Set a-> Bool
  insert, delete,  -- Ord a => a-> Set a-> Set a
  contains,        -- Ord a => Set a-> a-> Bool
```

```
  enumeration,      -- => Set a-> [a]
  ) where

import qualified AVLTree as A
```

Auch hier wird das Modul *AVLTree* qualifiziert importiert und umbenannt, und *Set* als **newtype** definiert:

```
newtype  Set a = Set (A.AVLTree a)

empty =  Set A.empty

isEmpty (Set t) = A.isEmpty t

insert x (Set t) = Set (A.insert x t)

delete x (Set t) = Set (A.delete x t)

contains (Set t) x = A.contains t x

enumeration (Set t) = A.enumeration t
```

7.1.5 Reichhaltigere Mengen

Die Implementierungen von Mengen haben nur sehr wenige Operationen realisiert. Natürlich möchte man Mengen vereinigen, schneiden, ihre Differenz bilden.

Dies kann man immer noch tun. Es zeigt sich nämlich, dass die bereits implementierten Basis-Operationen ausreichen, um auch diese Mengenoperationen zu definieren, als *abgeleitete Operationen*. Dazu importieren wir eines der Module (*SetAsAVLTree*), die Mengen implementieren, in ein neues Modul *RichSet*. Dazu spendieren wir noch eine FUnktion *set*, mit der Listen in Mengen konvertiert werden können. (Da wir die Schnittstelle von *SetAsAVLTree* nur erweitern wollen, "re-exportieren" wir die gesamte Exportliste des Moduls.

```
module RichSet (set,          -- :: Ord a => [a]-> Set a
                union,
                intersection,
                difference,   -- :: Ord a=> Set a-> Set a-> Set a
                module SetAsAVLTree
  ) where

import SetAsAVLTree
```

Mithilfe der Operation *enumeration* können die Mengen in Listen umgewandelt werden, und die Listenkombinatoren `foldr`, `map` und `filter` benutzt werden, um die neuen Funktionen zu definieren.

```
set :: Ord a ⇒ [a]→ Set a
set = foldr insert empty

union, intersection, difference :: Ord a ⇒ Set a→ Set a→ Set a

union s = foldr ($) s ∘ map insert ∘ enumeration

difference s = foldr ($) s ∘ map delete ∘ enumeration

intersection s = foldr insert empty
               ∘ filter (contains s) ∘ enumeration
```

Diese Vorgehensweise hat den Vorteil, dass die Operationen von `RichSet` nicht von der konkreten Implementierung der Basisoperationen abhängen. Wenn man nur diese Basisoperationen neu implementiert, bekommt man "automatisch" eine neue reichhaltige Implementierung von Mengen.

7.2 Abschließende Bemerkungen

Auch die meisten Module in der HASKELL-Bibliothek definieren abstrakte Datentypen. Das gilt auch für vordefinierte Zahlentypen, denn die interne Darstellung der Zahlenwerte wird verborgen und wird allein über die auf ihnen definierten Operationen definiert:

- Ihre Konstruktoren sind einerseits die *Zahlenliterale* wie `42`, `3.1415` und `12.4e-8`, mit denen Zahlenwerte direkt hingeschrieben werden können, andererseits vordefinierte Konstanten wie `maxInt` und `pi`.
- Selektoren gibt es nicht, weil die Datentypen atomar sind. (Bei Gleitpunktzahlen könnte man darüber streiten; wirklich kann man für Werte der Klasse `Floating` auf Komponenten wie die Mantisse zugreifen.)
- Inspektionsoperationen sind beispielsweise die Gleichheits- und Ordnungsrelationen.

7.3 Haskells Eck

7.3.1 newtype-Definitionen

Ein abstrakter Datentyp darf nicht als Typsynonym (mit **type**) definiert werden, weil Synonyme reine Abkürzungen sind keine neuen Typen einführen, deren Darstellung man verbergeb könnte. Algebraische Datentypen (definiert mit **data**) führen einen neuen Typ ein, dessen Werte mit Konstruktoren konstruiert werden und deshalb von allen anderen Datentypen unterschieden werden können.

Gerade abstrakte Datentypen möchte man oft als neue Typen einführen, ohne den Mehraufwand für die Anwendung der Konstruktoren zu haben. Dafür gibt es eine weitere Art von Typdefinition:

$$\textbf{newtype}\ [\ \mathit{Context}\ \Rightarrow]\, T\ \alpha_1 \cdots \alpha_k = K\ t_1 \ldots t_m\ [\ \textbf{deriving}\ (C_1, ..., C_n)\ ]$$

Der Typ T ist ein neuer Typkonstruktor mit genau einem (Pseudo-) Wertkonstruktor K. Die Werte von T müssen im Programm mit K "verpackt" hingeschrieben werden – bei der Auswertung wird der Konstruktor K intern aber nicht auf die Werte angewendet, also kein Knoten erzeugt.

7.3.2 Klassen-Instantiierung

Dieser Abschnitt gehört thematisch noch in das Kapitel 5. Dort haben wir gesehen, wie wir die Operationen vordefinierter Typklassen für einen algebraischen Datentyp mit "... **deriving** ..." automatisch ableiten lassen können.

Hier wollen wir nachtragen, unter welchen Bedingungen Klasseneigenschaften automatisch abgeleitet werden können, und wie man selbst einen Typ zur Instanz einer Klasse machen kann.

Ableiten einer Klasseninstanz

Für einen algebraischen Datentyp

$$\begin{aligned}\textbf{data}\ \mathit{Context} \Rightarrow T\ \alpha_1 \cdots \alpha_k = {} & K_1\ \mathit{Typ}_{1,1} \ldots \mathit{Typ}_{1,k_1} \\ & \vdots \\ \mid {} & K_n\ \mathit{Typ}_{n,1} \ldots \mathit{Typ}_{n,k_n} \\ & \textbf{deriving}\ (C_1, \ldots, C_m)\end{aligned}$$

kann eine Instanz für eine der vordefinierten Typklassen

$$C \in \{\mathit{Eq}, \mathit{Ord}, \mathit{Enum}, \mathit{Bounded}, \mathit{Show}, \mathit{Read}\}$$

unter folgenden Bedingungen automatisch abgeleitet werden:

1. Es gibt einen Kontext $\mathit{Context}'$, so dass sich für alle Komponententypen ableiten lässt:

   $$\mathit{Context}' \Rightarrow C\ \mathit{Typ}_{i,j}$$

2. Wenn $C = \mathit{Bounded}$, muss T ein Produkttyp oder ein Aufzählungstyp sein; d.h., $n = 1$ oder $k_j = 0$ für $1 \leqslant j \leqslant n$.
3. Wenn $C = \mathit{Enum}$, muss T ein Aufzählungstyp sein; d.h., $k_j = 0$ für $1 \leqslant j \leqslant n$.
4. Es darf keine explizite Instanzvereinbarung geben, die T zu einer Instanz von C macht.

Analoges gilt für eine **newtype**-Vereinbarung, die die allgemeine Form hat:

$$\textbf{newtype}\ \mathit{Context} \Rightarrow T\ \alpha_1 \cdots \alpha_k = K\ \mathit{Typ}_1 \ldots \mathit{Typ}_k \quad \textbf{deriving}\ (C_1, \ldots, C_m)$$

Instantiierung einer Klasse

Alle algebraischen Typen können zur Instanz einer Klasse gemacht werden, indem sie explizit als Instanz vereinbart werden. Eine Instanzvereinbarung hat die Form

$$\begin{array}{l}\textbf{instance}\ \mathit{Context} \Rightarrow C\ \mathit{Typ}\ \textbf{where}\\ \quad \mathit{function\ declaration}_1\\ \qquad\qquad \vdots\\ \quad \mathit{function\ declaration}_k\end{array}$$

Für die Instanz müssen wenigstens die für C *geforderten* Funktionen definiert werden; es können auch alle Funktionen der Klasse definiert werden. Damit werden die *default*-Vereinbarung der Klasse im objektorientierten Sinne *überschrieben*. Eine Instanz hängt von einem *Context* ab, wenn der Typ *Typ* parametrisiert ist und die Instantiierung nur gemacht werden kann (soll), wenn die Parameter ihrerseits Instanzen von C oder von anderen Klassen sind.

8

Verzögerte Auswertung

In diesem Kapitel beschäftigen wir uns noch einmal mit der *Auswertungsreihenfolge* in funktionalen Programmen. Grundsätzlich können funktionale Programme strikt, normalisierend oder verzögert ausgewertet werden. Haskell benutzt verzögerte Auswertung.

Wir werden deshalb die genauen Regeln für verzögerte Auswertung ansehen und untersuchen, welche praktischen Vorteile das fürs Programmieren hat. Unter anderem können mit verzögerter Auswertung unendliche Datenstrukturen und nicht abbrechende Rekursion benutzt werden. Als praktisches Beispiel werde n wir Parser-Kombinatoren betrachten.

8.1 Auswertung

8.1.1 Strategien

Als Benutzer können wir uns die Ausführung eines funktionalen Programms als die *Auswertung* eines Ausdrucks e vorstellen. Dazu werden auf alle Funktionsaufrufe solange ihre Definitionen angewendet, bis der Ausdruck *reduziert* ist, h. h., keine Funktionsaufrufe mehr enthält, sondern nur noch Wertkonstruktoren und vordefinierte einfache Werte (Zahlen, Zeichen, Wahrheitswerte). So ein Ausdruck wird als eine *Normalform* von e bezeichnet.

Sobald ein Ausdruck e mehr als nur einen Funktionaufruf enthält, stellt sich die Frage, in welcher Reihenfolge sie ausgewertet werden. Dazu gibt es zwei wesentliche Strategien:

- *Strikte Auswertung* wertet immer die *innerste* Funktionsanwendung $f\ a$ aus (bezüglich der Klammerstruktur des Ausdrucks). Gibt es mehrere "innerste" Anwendungen, wird meist die *linkeste* genommen. In diesem Fall kann das Argument a von f keine Funktionsanwendungen mehr enthalten; es ist also schon ausgewertet. Deshalb heißt diese Reihenfolge auch *call-by-value*; ein anderer Name ist *leftmost innermost evaluation.*

- *Normalisierende Auswertung* wertet immer die *äußerste* Funktionsanwendung aus, und wählt von mehreren "äußersten" immer die *linkeste*. In diesem Fall kann das Argumente a von f noch weitere Funktionsanwendungen enthalten; die Funktion f erhält sie unausgewertet. Deshalb heißt diese Reihenfolge auch *call-by-name*; ein anderer Name ist *leftmost innermost evaluation*.

Wegen der Form der Funktionsdefinitionen lässt sich zeigen, dass die Ergebnisse der Auswertung für beide Strategien gleich sind, wenn alle Funktionsanwendungen definiert sind, also *terminieren*.

Anders sieht es aus, wenn mindestens eine Funktionsanwendung in e nicht terminiert, also den fiktiven Wert $\perp$ liefert:

- Bei strikter Auswertung ist der Wert des gesamten Ausdrucks dann undefiniert, d. h., $e \overset{*}{\rightsquigarrow} \perp$.
- Normalisierende Auswertung liefert immer dann eine definierte Normalform, wenn das nur irgendwie möglich ist. (Deshalb heißt sie "normalisierend".)

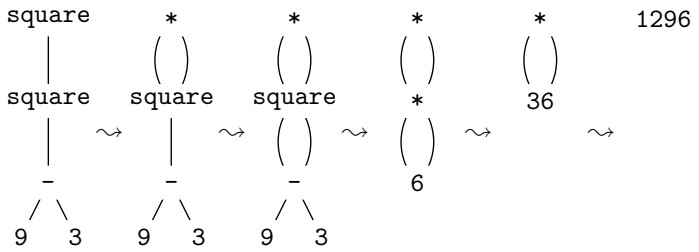
Die Übergabe von ausgewerteten Argumenten an Funktionen ist effizienter als der von unausgewerteten Ausdrücken; die besseren Terminationseigenschaften der normalisierenden Auswertung haben also ihren Preis.

Wir vergleichen die Auswertung eines Ausdrucks für die Funktion $square\, x = x * x$ mit den beiden Auswertungsstrategien:

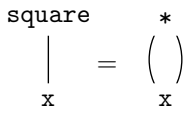
$$
\begin{array}{l}
square(square(9-3)) \\
\rightsquigarrow square(square\ 6) \\
\rightsquigarrow square(6*6) \\
\rightsquigarrow square\ 36 \\
\rightsquigarrow 36*36 \\
\rightsquigarrow 1296
\end{array}
\qquad
\begin{array}{l}
square(square(9-3)) \\
\rightsquigarrow (square\ (9-3)) * (square\ (9-3)) \\
\rightsquigarrow ((9-3)*(9-3)) * (square\ (9-3)) \\
\rightsquigarrow (6*(9-3)) * (square\ (9-3)) \\
\rightsquigarrow (6*6) * (square\ (9-3)) \\
\rightsquigarrow 36 * (square\ (9-3)) \\
\rightsquigarrow 36 * ((9-3)*(9-3)) \\
\rightsquigarrow 36 * (6*(9-3)) \\
\rightsquigarrow 36 * (6*6) \\
\rightsquigarrow 36 * 36 \\
\rightsquigarrow 1296
\end{array}
$$

Das Übergeben unausgewerteter Ausdrücke führt in diesem Beispiel dazu, dass *Arbeit kopiert wird*: Dann müssen insgesamt 4 Kopien des Teilausdrucks $9-3$ ausgewertet werden. Dies ist vollkommen unnötig, denn es gilt ja *referentielle Transparenz*: Jede Auswertung von $9-3$ und von $square(9-3)$ liefert das gleiche Ergebnis. Also reicht es aus, ihre Werte einmal auszurechnen und diesen Wert danach zu benutzen. Dies kann man erreichen, wenn die Anwendung von *square* das Argument nicht kopiert, sondern zwei Referenzen auf das Argument macht. Der Ausdruck wird also als Graph dargestellt, in dem gleiche Teilausdrücke mehrfach benutzt werden können. (Auf englisch nennt

man dies "*sharing*".) Die Auswertung des Beispiels mit *sharing* sieht dann so aus:



Diese Verfeinerung der normalisierenden Auswertung heißt *verzögerte Auswertung* (engl. *lazy evaluation* oder *call-by-need evaluation*). Ihr liegt eine Darstellung der Ausdrücke als Graphen, und der Funktionsgleichungen als Graphersetzungsregeln zu Grunde. Die Definition von *square* muss man sich also so vorstellen:



Die verzögerte Auswertung ist auch normalisierend (liefert definierte Werte wann immer möglich), vermeidet aber das Kopieren von Arbeit – allerdings nicht immer. Auf jeden Fall wird ein Teilausdruck nur dann wirklich ausgewertet, wenn er benötigt wird, um das Ergebnis des Ausdrucks zu bestimmen.

8.1.2 Striktheit

Betrachten wir die – zugegeben: dumme – Funktion:

```
silly x y = y
```

Bei der Auswertung des Ausdrucks

$$silly(square\ 3)(square\ 4) \rightsquigarrow square\ 4 \rightsquigarrow 4 * 4 \rightsquigarrow 16$$

stellen wir fest:

- Das erste Argument von `silly` wird *gar nicht* ausgewertet.
- Das zweite Argument von `silly` wird erst *im Funktionsrumpf* ausgewertet.

Was passiert, wenn eines der Argumente undefiniert ist?

- Weil das erste Argument nicht gebraucht wird, macht es nichts, wenn sein Wert undefiniert ist.

- Da das zweite Argument von *silly* gebraucht wird, wird sein Wert immer dann berechnet, wenn das Ergebnis von *silly* gebraucht wird; ist es undefiniert, wird auch der Wert von *silly* undefiniert sein.

Diese Beobachtung führt zu einer Definition von Striktheit. Eine n-stellige Funktion f ist *strikt* in ihrem i-ten Argument ($1 \leqslant i \leqslant n$, wenn der Wert der Funktion *immer* undefiniert ist, wenn dieses Argument undefiniert ist:

$$a_i = \bot \Rightarrow f\ a_1 \cdots a_{i-1}\ a_i\ a_{i+1} \cdots a_n = \bot$$

Eine Funktion heißt einfach strikt, wenn sie in allen Argumenten strikt ist. Beispiele für strikte (Argumente von) Funkionen in HASKELL sind:

- Die vordefinierte Addition "`(+)`" ist strikt, wie fast alle vordefinierte Funktionen auf Zahlen.
- Die logische Konjunktion "`(\&\&)`" und die logische Disjunktion "`(||)`" sind jeweils nur in ihrem ersten Argument strikt, nicht aber in ihrem zweiten.

  `False && (1/0 ==0)` $\rightsquigarrow$ `False`

- *`silly`* ist strikt im zweiten Argument und nicht-strikt im ersten Argument.

  `silly` `(1/0) 3` $\rightsquigarrow$ `3`

In HASKELL kann strikte Auswertung *erzwungen* werden. Die eingebaute Funktion `seq` ist strikt in ihrem ersten Argument:

```
seq :: a → b → b
seq = ...        -- Eingebaute Primitive
```

Damit ist die ebenfalls vordefinierte strikte Funktionsanwendung `(\$!)` definiert:

```
($!) :: (a → b) → a → b
f $! x    =  x `seq` f x
```

8.2 Daten-orientierte Programme

Funktionen werden häufig so definiert, dass sie eine Datenstruktur in mehreren Schritten *transformieren*. Betrachten wir beispielsweise die Funktion, die die Summe aller Kehrwerte einer Liste berechnet:

$$\sum_{i=1}^{n} \frac{1}{i}$$

Dies wird in drei Schritten berechnet:

- Zuerst wird die Liste der Zahlen von 1 bis n aufgebaut, mit der vordefinierten Funktion *fromTo* `1 n`. (Syntaktisch verzuckert kann sie als "`[1..n]`" geschrieben werden.)
- Dann wird mit dem Kombinator `map (1/)` die Liste der Kehrwerte gebildet.
- Schließlich wird die Summe durch Falten mit `(+)` und `0` gebildet.

```
foldr op c []     = c
foldr op c (h:t) = h `op` foldr op c t
map f [] = []
map f (h:t) = f h : map f t
fromTo m n | m > n      = []
           | otherwise = m : fromTo (succ m) n

reciprocalSum :: (Enum a,Ord a,Fractional a) ⇒ a → a
reciprocalSum n = foldr (+) 0 (map (1/) (fromTo 1 n))
```

Wenn wir die Auswertung verfolgen, stellen wir fest, dass weder die Liste $[1, 2, \ldots n]$ noch die Liste der Kehrwerte $[\frac{1}{1}, \frac{1}{2}, \ldots \frac{1}{n}]$ je aufgebaut wird, sondern immer nur jeweils das nächste Listenelement aufgezählt, sein Kehrwert gebildet und aufsummiert wird:

```
    foldr (+) 0 (map (1/) (fromTo 1 n))
⇝ foldr (+) 0 (map (1/) (1 : fromTo (succ 1) n))
⇝ foldr (+) 0 (1 : map (1/) (fromTo (succ 1) n))
⇝ 1 +foldr (+) 0 (map (1/) (fromTo (succ 1) n))
⇝ 1 +foldr (+) 0 (map (1/) (fromTo 2 n))
⇝ 1 +foldr (+) 0 (map (1/) (2 : fromTo (succ 2) n))
⇝ 1 +foldr (+) 0 (0.5 : map (1/) (fromTo (succ 2) n))
⇝ 1 +(0.5 +foldr (+) 0 (map (1/) (fromTo (succ 2) n)))
⇝ 1 +(0.5 +foldr (+) 0 (map (1/) (fromTo (3) n)))
⇝* 1 +(0.5 +... +1/n) ...
```

Bei sequentiellen Transformationen von Listen oder anderen rekursiven Datenstrukturen vermeidet die verzögerte Auswertung das Aufbauen von Zwischenergebnissen. In diesem Beispiel ist es ungünstig, dass `foldr` die Summen nach rechts klammert, weil so erst alle Summenglieder berechnet werden müssen, bevor die erste Addition angewendet werden kann. Wir sollten statt dessen besser die vordefinierte Funktion `foldl` nehmen:

```
foldl           :: (b → a → b) → b → [a] → b
foldl f z []    =  z
foldl f z (x:xs) =  foldl f (f z x) xs
```

Dann sähe die Auswertung so aus:

```
   foldl (+) 0 (map (1/) (fromTo 1 n))
↝ foldl (+) 0 (map (1/) (1 : fromTo (succ 1) n))
↝ foldl (+) (0+1) (map (1/) (fromTo (succ 1) n))
↝ foldl (+) 1 (map (1/) (fromTo (succ 1) n))
↝ foldl (+) 1 (map (1/) (fromTo 2 n))
↝ foldl (+) 1 (0.5 : map (1/) (fromTo (succ 2) n))
↝ foldl (+) (1+0.5) (map (1/) (fromTo (succ 2) n)))
↝ foldl (+) 1.5 (map (1/) (fromTo (succ 2) n)))
↝ foldl (+) 1.5 (map (1/) (fromTo (3) n)))
↝* ...
```

Knapper kann man *reciprocalSum* übrigens mit einer Listenumschreibung definieren:

```
reciprocalSum n = foldl (+) 0 [ 1/i | i← [1..n]]
```

Verzögerte Auswertung kann auch zu fast skurrilen Lösungen führen, wie zur Definition des Minimums einer Liste durch *Mergesort*:

```
msort :: Ord a⇒ [a]→ [a]
msort xs
  | length xs ≤ 1 = xs
  | otherwise = merge (msort front) (msort back) where
    (front, back) = splitAt ((length xs) `div` 2) xs
    merge :: Ord a⇒ [a]→ [a]→ [a]
    merge [] x = x
    merge y [] = y
    merge (x:xs) (y:ys) | x≤ y     = x:(merge xs (y:ys))
                        | otherwise = y:(merge (x:xs) ys)

min' :: Ord a⇒ [a]→ a
min' xs = head (msort xs)
```

Die Beispielauswertung zeigt, dass die Liste gar nicht vollständig sortiert wird – es wird wenig mehr getan als binäre Suche in der Liste:

```
   min' [4,2,1,3]
↝ head (msort [4,2,1,3])
↝* head (merge (msort [4,2]) (msort [1,3]))
↝* head (merge (merge (msort [4]) (msort [2])) (msort [1,3]))
↝* head (merge (merge [4] [2]) (msort [1,3]))
↝ head (merge (2:merge [4] []) (msort [1,3]))
↝* head (merge (2:merge [4] []) (merge (msort [1]) (msort [3])
↝* head (merge (2:merge [4] []) (merge [1] [3])
↝ head (merge (2:merge [4] []) (1:merge [] [3])
↝ head (1:merge (2:merge [4] []) (merge [] [3])
↝ 1
```

8.3 Ströme

Ströme (engl. *streams*) sind *unendliche Listen*. In einer verzögert ausgewerteten Sprache können unendliche Listen definiert werden, weil nicht gleich versucht wird, sie vollständig auszurechnen, sondern immer nur so weit, wie es für das Resultat des Programms nötig ist. Die unendliche Liste $[2, 2, 2, \ldots]$ von Zweien kann so definiert werden

```
twos = 2 : twos
```

Diese Liste könnte man in HASKELL auch mit dem Ausdruck `enumFromBy 2 0` oder der äquivalenten Notation `[2,2 ..]` definieren.

Auch die Liste aller natürlichen Zahlen kann definiert werden:

```
from :: Integral a ⇒ a → [a]
from n = n : from (n+1)
```

Die Funktion `cycle` bildet eine unendliche Verkettung ihres Arguments:

```
cycle :: [a] → [a]
cycle xs = xs ++ cycle xs
```

Weil Funktionsdefinitionen *Graphersetzungsregeln* sind, werden die unendlichen Listen `twos` und `cycle xs` intern durch endliche, aber zyklische Graphen dargestellt. Die Auswertung von `cycle` terminiert also. (Das Ausgeben als Zeichenkette aber nicht!)

Etwas interessantere Beispiele sind für unendliche Listen sind Primzahlen und Fibonacci-Zahlen.

Beispiel 8.1 (Sieb des Erathostenes). Wir hatten das *Sieb des Erathostenes* schon kennengelernt, um die Primzahlen in der Menge $[1..n]$ zu berechnen.

Wenn wir die ersten n Primzahlen berechnen wollen, ist nicht einfach zu bestimmen, bis wohin gesiebt werden muss. Dank verzögerter Auswertung können wir das Sieb für die Berechnung *aller* Primzahlen definieren. Von dieser unendlichen Liste nehmen wir nur die ersten n, und genau so viele werden dann auch nur gesiebt.

```
sieve :: [Integer]→ [Integer]
sieve (p:ps) =
  p:(sieve (filter (\n→ n `mod` p /= 0) ps))

primes :: [Integer]
primes = sieve (from 2)
```

Es ist typisch für Funktionen auf Strömen, dass sie *keine* Rekursionsverankerung haben, wie `sieve` in diesem Beispiel.

Beispiel 8.2 (Fibonacci-Zahlen). Diese Funktion aus der Kaninchenzucht sollte jeder Informatiker kennen:

```
fib :: Integer→ Integer
fib 0 = 1
fib 1 = 1
fib n = fib (n-1)+ fib (n-2)
```

Diese Definition hat exponentiellen Aufwand, weil die letzte Gleichung *zwei* rekursive Aufrufe enthält. Leider kann nicht direkt ausgenutzt werden, dass dabei viele schon vorher bestimmten Werte unnütz erneut berechnet werden. (Eine allgemeine Technik für das Tabellieren solcher Funktionen, *Memoïsierung*, werden wir in Kapitel 13 kennenlernen.)

Wir können die zuvor berechnete Teilergebnisse aber auch wiederverwenden, wenn wir den *Strom* `fibs` aller Fibonaccizahlen betrachten und geeignet "aneinander legen":

```
          fibs  1  1  2  3  5  8 13 21 34 55
tail fibs       1  2  3  5  8 13 21 34 55
```

Dann können wir die Funktion so definieren:

```
fibs :: [Integer]
fibs = 1 : 1 : zipWith (+) fibs (tail fibs)
```

Der Aufwand ist nun linear, da die Liste `fibs` nur einmal ausgewertet wird.

Die n-te Fibonaccizahl können wir dann mit der Indizierungsfunktion `(!!) :: [a] → Int → a` auswählen:

```
ffib :: Int→ Integer
ffib = (fibs !!)
```

Es soll nicht verschwiegen werden, dass verzögerte Auswertung auch ihre Tücken hat. Ein Beispiel sind Fehlermeldungen, wie bei der folgenden Funktion:

```
remove :: Eq a ⇒ a → [a] → [a]
remove x []       = error "remove␣unexisting␣element"
remove x (y:ys)   = if x == y then ys else y : remove x ys
```

Damit gibt es folgende Auswertung:

```
(remove 7 [1,2,3,4,5,6]) !! 5 ⇝* 5
```

Auch *Fehlermeldungen* werden verzögert. In diesem Fall muss `remove` nicht vollständig ausgewertet werden, um den Zugriff auf das Listenelement auszuwerten, und der Fehler wird nicht gemeldet.

8.4 Fallstudie: Parsieren

In diesem Abschnitt wollen wir mit einem größeren Beispiel zeigen, wie systematisch Parsierer (im Folgenden auch oft *Parser* genannt) systematisch aus einer Grammatik hergeleitet werden können und dann – auch dank verzögerter Auswertung – immer noch relativ effizient implementiert werden können

8.4.1 Parsieren im Allgemeinen

Parsierer haben die Aufgabe, zu erkennen, ob ein Text gemäß den Regeln einer Sprache aufgebaut ist, und einen *Syntaxbaum* des Textes zu erstellen, wenn dies der Fall ist.

Die Regeln der Sprache werden üblicherweise mit einer *kontextfreie Grammatik* definiert, die in erweiterter Backus-Naur-Form oder einer ähnlichen Notation aufgeschrieben wird.

Beispiel 8.3 (Arithmetische Ausdrücke). Die Syntax *arithmetischer Ausdrücke* kann mit folgender kontextfreien Grammatik beschrieben werden.

$$
\begin{aligned}
E &::= T\texttt{+}E \\
&\mid\ T\texttt{-}E \\
&\mid\ T \\
T &::= F\texttt{*}T \\
&\mid\ F\texttt{/}T \\
&\mid\ F \\
F &::= N \mid \texttt{(}E\texttt{)} \\
N &::= D \mid D\ N \\
D &::= \texttt{0} \mid \cdots \mid \texttt{9}
\end{aligned}
$$

Darin sind E, T, F, N, D *Nichtterminalsymbole* für Ausdrücke (engl. *expression*), Terme, Faktoren, Zahlenkonstanten (engl. *number*) und Ziffern (engl. *digit*), und Terminalsymbolen "+", "`-`", "*", "/", "`(`", "`)`", "`0`", ... "`9`".

In den Regeln steht "|" für alternative syntaktische Formen; Hintereinanderschreiben von Nichtterminalen und Terminalen steht für sequenzielle syntaktische Komposition.

Arithmetische Ausdrücke sollen als Bäume der *abstrakten Syntax* vom Typ `Expr` ausgegeben werden:

```
data Expr = Plus Expr Expr
          | Minus Expr Expr
          | Times Expr Expr
          | Div Expr Expr
          | Number Int
          deriving (Eq, Show)
```

Die Nichtterminale E, T und F der kontextfreien Syntax unterscheiden Ausdrücke mit Operatoren verschiedenen Vorrangs; eine solche Unterscheidung ist bei den Bäumen der abstrakten Syntax unnötig.

8.4.2 Der Typ von Parsierfunktionen

Ein Parser wäre im Prinzip eine Funktion von Text in Syntaxbäume. Aus praktischen Gründen wird allerdings meistens ein *Scanner* benutzt, um den Text zunächst zu einer Sequenz von Lexemen (engl. *token*) zusammen zu fassen, bevor der eigentliche Parser in Aktion tritt.

Welchen Typ hat also ein Parser? Der erste Versuch wäre

```
type Parse1 a b = [a] → b
```

Hierbei ist `a` der Typparameter für die Lexeme (bzw. Zeichen, wenn auf den Scanner verzichtet wird), und `b` der Typ der Syntaxbäume. Betrachten wir Auswertungen dieser Funktionen, wobei wir annehmen, `number` und `bracket` seien Funktionen dieses Typs (mit `a = b = String`):

```
bracket "(23+4)" ⇝* "("
   number "23+4" ⇝* "2" oder "23"
bracket "23+4)" ⇝* ??
```

Ein Parser kann also mehrere Ergebnisse finden oder auch keines. Deshalb ist es günstiger, als Ergebnis eine *Liste* von Zerlegungen liefern zu lassen; die leere Liste zeigt dann an, dass keine Zerlegung gefunden wurde. Dann bleibt immer noch ein anderes Problem: Ein Parser erkennt nicht immer die komplette Eingabe, sondern nur einen Teil davon (einen *Präfix*, weil wir von links nach rechts parsieren). Da wir wissen müssen, was "übrig geblieben" ist, sollte jede Zerlegung also aus dem Syntaxbaum und dem nicht erkannten Teil der Eingabe bestehen. Der allgemeine Typ eines Parsers ist also:

```
type Parse a b = [a] → [(b,[a])]
```

Dann liefert die Auswertung für die oben genannten Beispiele:

```
bracket "(23+4)" ⇝* [("(","23+4)")
   number "23+4" ⇝* [("2","3+4"),("23","+4")]
bracket "23+4)" ⇝* []
```

Die Parsierungen eines Ausdrucks könnte also für unser Beispiel so aussehen.

```
parse "3+4*5" ⇝ [(3, "+4*5"),
                 (Plus 3 4, "*5"), (Plus 3 (Times 4 5), "")]
```

8.4.3 Kombination von Parsern

In diesem Abschnitt betrachten wir einige *primitive Parser* und einige *Parserkombinatoren*, mit denen aus Parsern neue Parser zusammengestellt werden können.

Der primitive Parser `none` erkennt nichts:

```
none :: Parse a b
none = const []
```

Der primitive Parser `succeed b` erkennt alles als `b`:

```
succeed :: b→ Parse a b
succeed b inp = [(b, inp)]
```

Die primitive Parser `spot` und `token` erkennen einzelne Symbole in einer Symbolkette:

```
spot  :: (a→ Bool)→ Parse a a
spot p []     = []
spot p (x:xs) = if p x then [(x, xs)] else []

token :: Eq a⇒ a→ Parse a a
token t = spot (== t)
```

Die Parser `none` und `succeed` werden nicht mit `spot` implementiert, weil `spot` nur für leere Eingabe nichts liefert, und nur für nicht-leere Eingabe ein Ergebnis findet.

Jetzt betrachten wir allgemeine Kombinatoren für Parser. Seien p_1 und p_2 Parser für Sprachen A bzw. B:

- Der Kombinator $p_1 \langle|\rangle p_1$ konstruiert einen Parser, der alternativ entweder A oder B parsiert.

  ```
  infixl 3 <|>
  (<|>) :: Parse a b→ Parse a b→ Parse a b
  (p1 <|> p2) i = p1 i ++ p2 i
  ```

- Der Kombinator $p_1 > . > p_1$ konstruiert einen Parser, der hintereinander erst A und dann B parsiert. Dabei ist der Rest des ersten Parsers die Eingabe für den zweiten

  ```
  infixl 5 >.>
  (>.>) :: Parse a b→ Parse a c→ Parse a (b, c)
  (p1 >.> p2) i = [((y, z), r2) | (y, r1) ← p1 i,
                                  (z, r2) ← p2 r1]
  ```

- Mit dem Kombinator $p_1 \langle * \rangle f$ wird die Eingabe des Parsers zu einem Syntaxbaum weiterverarbeitet:

```
infix 4 <*>
(<*>) :: Parse a b→ (b→ c)→ Parse a c
(p <*> f) inp = [(f x, r) | (x, r)← p inp]
```

Mit "$\langle * \rangle$" kann das Hintereinanderschalten von Parser so definiert werden, dass das Ergebnis des ersten bzw. zweiten Parsers "vergessen" wird.

```
infixl 5 .>, >.
(.>) :: Parse a b→ Parse a c→ Parse a c
(>.) :: Parse a b→ Parse a c→ Parse a b
p1 .> p2 = p1 >.> p2 <*> snd
p1 >. p2 = p1 >.> p2 <*> fst
```

Mit den drei allgemeinen Kombinatoren und den speziellen Varianten des Hintereinanderschaltens können weitere Kmbinatoren für Parser definiert werden: (z.B. Sequenzen A^*, nicht-leere Sequenzen A^+)

- Der Listen-Kombinator `list` baut aus einem Parser für die Sprache A einen für beliebige Sequenzen A^*. Dabei wird die Definition von Sequenzen als $A^* ::= AA^* \mid \varepsilon$ direkt umgesetzt:

```
list :: Parse a b→ Parse a [b]
list p =  p >.> list p <*> uncurry (:)
      <|> succeed []
```

- Analog kann der Kombinator `some` zum Parsieren nicht leerer Listen A^+ aus deren Definition als $A^+ ::= AA^*$ abgeleitet werden:

```
some :: Parse a b→ Parse a [b]
some p = (p >.> list p) <*> uncurry (:)
```

(Zur Erinnerung: `uncurry (:) (a,b) =a : b` macht aus einer Funktion mit zwei Parametern eine, die ein Paar von diesen Parametern erwartet.) Mit den **infix**-Definitionen haben wir Vorrang und Assoziativität der Kombinatoren so definiert, dass wir beim Schreiben von Ausdrücken Klammern sparen: `>.>` (5) vor `(<*>)` (4) vor `<|>` (3)

8.4.4 Christian Maeders Parser

```
type Parse2 a b = [a] → [(b, [a]]
```

Eine Liste von erfolgreichen Parsierungsteilschritten zu verwalten ist ineffizient, deshalb behält man immer nur das längste Ergebnis (longest match).

Ein möglicher Misserfolg wird durch den Datentyp `Maybe` verwaltet. ("`(Maybe b, [a])`" wäre auch möglich)

```
type Parse3 a b = [a] → Maybe (b, [a])

bracket "(23+4" → Just ("(", "23+4")
number "23+4" → Just ("23", "+4")
bracket "23+4)" → Nothing
```

Eine erfolgreiche Parsierung könnte so aussehen:

$$parse\ "3 + 4 * 5" \rightsquigarrow (Just(Plus\ 3(Times\ 4\ 5)),)$$

Man beachte, dass durch die Grammatikregeln

$$E ::= T + E$$

Punktrechnung vor Strichrechnung gewährleistet wird. “4*5” ist der längste legale Term für das rechte Argument von “+”. (Allerdings werden mehrere Summanden auch rechtsassoziativ geklammert, was der Linksassoziativität von Addition und Subtraktion widerspricht.)

In der Praxis ist “`Parse`” ein abstrakter Datentyp mit besserer Fehler- und Positionsverwaltung, als durch “`Maybe`” möglich ist.

Außerdem möchte man z.B. in einem Fehlerfall, bei dem kein Input konsumiert wird (wie der obige zweite, Aufruf von “`bracket`”), einen alternativen Parser aufrufen können:

```
data Parse a b = Parser (State a → Reply a b)

data State a = State [a] Int  -- Position

data Reply a b = OkReply b (State a) Consumption
               | FailReply Consumption String  -- Fehlermeldung

data Consumption = NoInputConsumed | Consumed deriving (Eq, Ord)
```

Elementare Parser

Der primitive Parser, der nichts erkennt und mit einer Meldung fehlschlägt.

```
failP :: String → Parse a b
failP str = Parser ( \ _ → FailReply NoInputConsumed str)
```

Der primitive Parser, der nichts konsumiert aber seine Eingabe als erfolgreiches Ergebnis ausgibt.

```
returnP :: b → Parse a b
returnP b =  Parser ( \ s → OkReply b s NoInputConsumed)
```

Der primitive Parser, der das nächste Token konsumiert und als Ergebnis ausgibt oder einen Fehler bei leerem Eingabestring meldet.

```
anyToken :: Parse a a
anyToken = Parser ( \ (State l i) → case l of
         [] → FailReply NoInputConsumed ("end␣of␣input␣at:␣" ++
show i)
         h : t → OkReply h (State t (i+1)) Consumed)
```

Der Kombinator (<|>) parsiert eine Eingabe mit seinem ersten Argument p_1. Nur wenn der erste Parser p_1 fehlschlägt, ohne Input zu konsumieren, wird der alternative Parser p_2 aufgerufen. Der Anfang von p_1 muss verschieden sein von dem von p_2, weil p_1 sonst "stecken bleibt", wenn eigentlich p_2 erkannt werden soll.

```
(<|>) :: Parse a b → Parse a b → Parse a b
Parser p1 <|> Parser p2 = Parser ( \ s → case p1 s of
    FailReply NoInputConsumed _ → p2 s
    r → r)
```

Der `try`-Parser versucht die Eingabe zu parsieren und, wenn das fehlschlägt, werden keine Tokens konsumiert, ach wenn der Fehler erst durch das Parsieren von nachfolgenden Tokens festgestellt wurde. Der `try`-Parser muss mit Bedacht eingesetzt werden, da er es zusammen mit dem Kombinator (<|>) ermöglicht, dieselbe Eingabe mehrmals zu parsieren.

```
try :: Parse a b → Parse a b
try (Parser p) = Parser ( \ (State l i) →
    case p (State l i) of
      OkReply b s c → OkReply b s c
      FailReply _ s → FailReply NoInputConsumed s)
```

Der Kombinator *bindP* konstruiert einen Parser, der zwei Parser nacheinander anwendet, wobei der zweite Parser vom Ergebnis des ersten Parsers abhängen darf. (Die Funktion "`max`" ist durch die Instanz "`Ord Consumption`" bekannt.)

```
bindP :: Parse a b → (b → Parse a c) → Parse a c
Parser p1 `bindP` f = Parser ( \ s →
                      case p1 s of
                        OkReply b t con1 →
                          let Parser p2 = f b
                          in case p2 t of
                            OkReply c u con2 → OkReply c u (max con1 con2)
                            FailReply con2 msg → FailReply (max con1 con2) msg
                            FailReply con1 e → FailReply con1 e)
```

Durch eine Monad Instanz kann man statt *bindP* das Infixsymbol (>>=) verwenden (und das "P" hinter `return` und `fail` weglassen).

```
instance Monad (Parse a) where
  return b = returnP b
  p1 >>= f = p1 `bindP` f
  fail s = failP s
```

Abgeleitete Parser

Die Abkürzung `(≫)` von `(≫=)` ignoriert die Eingabe des ersten Parsers:

```
(≫) :: Parse a b → Parse a c → Parse a c
p1 ≫ p2 = p1 ≫= (\ _ → p2)
```

`liftM` verändert das Ergebnis eines Parsers

```
liftM :: (b → c) → Parse a b → Parse a c
liftM f p = p ≫= (\ b → return (f b))
```

`liftM2` kombiniert die Ergebnisse von zwei unabhängigen Parsern, die nacheinander aufgerufen werden

```
liftM2 :: (b → c → d) → Parse a b → Parse a c → Parse a d
liftM2 f p1 p2 = p1 ≫= (\ b → p2 ≫= (\ c → return (f b c)))
```

`(≫)`, `liftM`, `liftM` müssen nicht explitit definiert werden, sondern sind abgeleitete Funktionen aus der `Monad`-Instanz. Daraüberhinaus sind diese Kombinatoren mit **do**-Notation im Allgemeinen gar nicht nötig:

$$\begin{aligned} \mathit{liftM2}\ f\ p_1\ p_2 = \mathbf{do}\ & b \leftarrow p_1 \\ & c \leftarrow p_2 \\ & \texttt{return}(f\ b\ c) \end{aligned}$$

Eine weitere Abkürzung "(≪)" ignoriert das (erfolgreiche) Ergebnis des zweiten Parsers. Man beachte, dass "(≪)" nicht das Gleiche wie "(≫)" mit vertauschten Argementen ist. In $p_1 \ll p_2$ wird (wie bei $p_1 \gg p_2$) erst p_1 und dann p_2 parsiert. Nur das Resultat ist das von p_1 (statt von p_2).

(Im alten Skript wurden statt ≫ und ≪ > . und . > benutzt.)

```
infixl 1 ≪  -- so wie >>
(≪) :: Parse a b → Parse a c → Parse a b
p1 ≪ p2 = liftM2 (\ b _ → b) p1 p2
```

In **do**-Notation

$$\begin{aligned} p_1 \ll p_2 = \mathbf{do}\ & b \leftarrow p_1 \\ & p_2 \\ & \mathbf{return}\ b \end{aligned}$$

Weitere abgeleitete elementare Parser:

Der *eof*-Parser testet, ob die Eingabe leer ist. Falls ein Token von `anyToken` konsumiert werden kann, schlägt der Parser fehl. Der `try`-Parser verhindert, dass das Token konsumiert wird. Falls der `anyToken`-Parser fehlschlägt, wird keine Eingabe konsumiert und durch die zweite Alternative von `(<|>)` wird eine erfolgreich Parsierung signalisiert.

```
eof :: Parse a ()
eof = try ((anyToken ≫ fail "no␣end␣of␣input") <|> return ())
```

Der satisfy-Parser erkennt einzelne Tokens, die ein Prädikat erfüllen. Durch den äußeren `try`-Parser wird ein unpassendes Zeichen nicht konsumiert.

```
satisfy :: Show a ⇒ (a → Bool) → Parse a a
satisfy p = try (anyToken >>= (\ t →
            if p t then return t
                   else fail ("unexpected␣token:␣" ++ show t)))
```

Der option-Parser versucht einen optionalen Teil zu parsieren. Schlägt die Parsierung mit p fehlt wird der Defaultwert b zurückgegeben

```
option :: b → Parse a b → Parse a b
option b p = p <|> return b
```

Der *many*-Parser versucht beliebig oft hintereinander mit p zu parsieren. Achtung: p sollte dabei immer mindestens ein Token konsumieren, um eine Endlos-Schleife zu vermeiden.

```
many :: Parse a b → Parse a [b]
many p = liftM2 (:) p (many p) <|> return []
```

Der *many1*-Parser ist wie der *many*-Parser parsiert aber p mindestens einmal

```
many1 :: Parse a b → Parse a [b]
many1 p = liftM2 (:) p (many p)
```

In **do**-Notation:

```
many1 p = do
   h <- p
   t <- many p
   return (h : t)
```

?? Parser ausführen. Es wird nur das Ergebnis extrahiert, ggf. sind noch Eingabesymbole übrig.

```
parse :: Parse a b → [a] → b
parse (Parser p) l = case p (State l 1) of
    OkReply b _ _ → b
    FailReply _ s → error s
```

Parser mit `Char` als Tokens

```
type CharParser a = Parse Char a
```

Parser für Zahlen (ohne Vorzeichen)

```
number :: CharParser Int
number = liftM read (many1 (satisfy isDigit))
```

Ignorieren von nachfolgenden Leerzeichen:

```
lexeme :: CharParser a → CharParser a
lexeme p = p << many (satisfy isSpace)
```

```
char :: Char → CharParser Char
char c = satisfy (== c)

string :: String → CharParser String
string str = case str of
    "" → return ""
    c : r → liftM2 (:) (char c) (string r)

symbol :: String → CharParser String
symbol str = lexeme (string str)
```

Der Datentyp für Ausdrücke:

```
data Expr = Expr InfixOp Expr Expr | Number Int deriving Show

data InfixOp = Div | Mult | Plus | Minus deriving Show
```

Für eine effiziente Parsierung ist eine Linksfaktorisierung der Grammatik nötig. Dabei werden Regeln mit gemeinsamen Anfängen zusammen gefasst.

$E ::= T + E|T - E|T$
$T ::= F * T|F/T|F$
$F ::= i|(E)$

wird:

$E ::= TS$
$S ::= +E| - E|\varepsilon$
$T ::= FG$
$G ::= *T|/T|\varepsilon$
$F ::= i|(E)$

Für die veränderte Grammatik wird der abstrakte Syntaxbaum wie folgt konstruiert. An den Parser für S (bzw G) wird das `Expr`-Ergebnis von T (bzw. F) übergeben. Zusammen mit einem Infixsymbol (abstrakt und als String) und einem `Expr`-Parser wird der Rest parsiert und ein Infixergebnis konstruiert.

```
infixExpr :: InfixOp → Expr → String → CharParser Expr → CharParser Expr
infixExpr i e s p =  liftM (Expr i e) (symbol s ≫ p)
```

Die Umsetzung der Regeln sieht dann so aus:

```
expr :: CharParser Expr
expr = term ≫= opExpr

opExpr :: Expr → CharParser Expr  -- S
opExpr t =
```

```
     infixExpr Plus t "+" expr
 <|> infixExpr Minus t "-" expr
 <|> return t

term :: CharParser Expr
term = factor >>= opTerm

opTerm :: Expr → CharParser Expr  -- G
opTerm f =
     infixExpr Mult f "*" term
 <|> infixExpr Div f "/" term
 <|> return f

factor :: CharParser Expr
factor =
      liftM Number (lexeme number)
  <|> (symbol "(" >> expr << symbol ")")
```

Aufruf mit Test, ob die Eingabe vollständig konsumiert wurde

```
parse (expr << eof) "(12 + 3) - 4 / 5 + 6 * 7"
```

8.4.5 Der Kern des Parsers für Ausdrücke

Nun definieren wir den Parser für Ausdrücke mithilfe der Kombinatoren. Alle Parser haben den Typ `Parse Char Expr`.

```
expression :: Parse Char Expr
expression =  term >. token '+' >.> expression <*> uncurry Plus
          <|> term >. token '-' >.> expression <*> uncurry Minus
          <|> term
```

Beim sequenziellen Parsieren der Terme eines Ausdrucks werden die Operatoren "vergessen" und die Operanden mit den Konstruktoren `Plus` und `Minus` des verknüpft. Da diese Konstruktoren zweistellig ("gecurryt") sind, muss die Funktion `uncurry` auf sie angewendet werden, bevor sie auf das von ">.>" gelieferte Paar passen.

Der Parser für Terme sieht ganz analog aus:

```
term :: Parse Char Expr
term =  factor >. token '*' >.> term <*> uncurry Times
    <|> factor >. token '/' >.> term <*> uncurry Div
    <|> factor
```

Beim Parsieren von Faktoren werden entweder Sequenzen von Ziffern als Zahlen erkannt, oder ein geklammerter Ausdruck.

```
factor :: Parse Char Expr
factor =
       some (spot isDigit) <*> Number. read
       <|> token '(' .> expression >. token ')'
```

Die Hauptfunktion `parse` ruft `expression` auf, entfernt aber vorher alle Zwischenräume mit der vordefinierten Funktion `isSpace`. Sie stellt sicher, dass die Eingabe vollständig parsiert wird und genau ein Syntaxbaum geliefert wird.

```
parse :: String→ Expr
parse i = case filter (null. snd) (expression[c| c←i, not(isSpace c)])
          of []      → error "Wrong␣input."
             [(e,_)] → e
             _       → error "Ambiguous␣input."
```

8.4.6 Nochmal Baumaufbau

Der oben definierte Parser hat noch einen kleinen Fehler: Die Bäume werden nicht so aufgebaut, wie es der Assoziativität der Operationen entspricht.

```
parse "9-4-3" ⇝ Minus (Number 9) (Minus (Number 4) (Number 3))
```

Solche Ausdrücke müssen links geklammert sein, denn sonst liefert die Auswertung von Substraktion und Division nicht das erwartete Ergebnis. (Im obigen Beispiel würde $9-(4-3) \rightsquigarrow 9-1 \rightsquigarrow 8$ ausgewertet statt $(9-4)-3 \rightsquigarrow 5-3 \rightsquigarrow 2$.

Zunächst formen wir die Regeln der Grammatik um, indem wir *erweiterte Backus-Naur-Form* benutzen. Darin steht "$\{X\}$" für eine Sequenz von $n \geqslant 0$ X'en.

$$E ::= T\ \{(+ \mid \texttt{-})\ T\} \qquad N ::= D \mid D\ N$$
$$T ::= F\ \{(* \mid /)\ F\} \qquad D ::= \texttt{0} \mid \cdots \mid \texttt{9}$$
$$F ::= N \mid \texttt{(}\ E\ \texttt{)}$$

Die abstrakte Syntax bleibt dieselbe.

Nach den entsprechenden Änderung in `expression` und `term` sieht der Parser dann so aus:

```
expression, term :: Parse Char Expr
expression = term >.> list( (token '+' <|> token '-')
                          >.> term                ) <*> leftExpr

term = factor >.> list( (token '*' <|> token '/')
                      >.> factor              ) <*> leftExpr

leftExpr:: (Expr,[(Char,Expr)]) → Expr
```

```
leftExpr (e,[]          ) = e
leftExpr (e,('+', e'):es) = leftExpr (Plus  e e',es)
leftExpr (e,('-', e'):es) = leftExpr (Minus e e',es)
leftExpr (e,('*', e'):es) = leftExpr (Times e e',es)
leftExpr (e,('/', e'):es) = leftExpr (Div   e e',es)
```

Die Funktionen `factor` und `parser` bleiben, wie sie waren.

Abschließend soll nicht verschwiegen werden, dass "industriell anwendbare" Parserkombinatoren noch etwas anders aussehen. Die Kombinatoren der Bibliothek `parseq`[1] liefern immer nur die längste der gefundenen Parsierungen (aus Effizienzgründen), und haben einen Ergebnistyp, der genauere Fehlermeldungen ermöglicht.

8.5 Haskells Eck

In Haskell können beliebige Folgen der Sonderzeichen

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

als *Operatorsymbole* definiert werden, solange sie sich von den folgenden reservierten Operatorsymbolen unterscheiden:

`reservedop` ::= `..` | `:` | :: | = | \ | | | ← | → | `@` | `~` | ⇒

Die Operatoren der vordefinierten Zahlentypen sind nicht reserviert, sondern nur *vordefiniert*: Sie können neu definiert werden, wenn dies nicht zu Namenskonflikten führt. (Namenskonflikte kann man vermeiden, indem man bestimmte Definitionen aus dem `Prelude` beim Import *versteckt*, oder den `Prelude` qualifiziert importiert. Vergleiche Abschnitt 7.3.)

Die Möglichkeit der Infixschreibweise erhöht die Mächtigkeit einer Sprache nicht. Trotzdem kann sie sinnvoll sein:

- Manche Operatoren wie "+" sind aus der Mathematik geläufig. Verwendet man sie auch in Programmen (mit der gleichen Bedeutung!), macht dies Programme lesbarer.
- Infix geschriebene Ausdrücke brauchen weniger Klammern, weil man sich den verschiedenen *Vorrang* (*Präzedenz*) von Operatoren und ihre *Assoziativität* zu Nutze machen kann.

Die Operationen in Haskell können Vorränge von 0 bis 10 haben, und sie können links-assoziativ, rechts-assoziativ oder gar nicht assoziativ sein. Ein Ausdruck "$x \oplus y \oplus z$" ist implizit geklammert als "$(x \oplus y) \oplus z$", wenn der Operator $\oplus$ links-assoziativ ist, bzw. als "$x \oplus (y \oplus z)$", wenn er rechtsassoziativ ist; ist $\oplus$ nicht assoziativ, wäre der Ausdruck syntaktisch fehlerhaft. Vorrang und Assoziativität wird mit einer Spezifikation der Form

[1] Im Web unter: `http://legacy.cs.uu.nl/daan/parsec.html`.

(**infix** | **infixl** | **infixr**) $n \oplus$

angegeben. Dabei ist $0 \geqslant n \geqslant 9$ der Vorrang für den Operator $\oplus$, und das "**l**" bzw. "**r**" spezifizieren sie ggf. als links- bzw. rechts-assoziativ.

Von den vordefinierten Operationen hat die *Funktionsanwendung* als einzige die höchste Priorität (10); sie ist linksassoziativ. Für die anderen gelten folgende Vorränge und Assoziativitäten:

```
infixr 9  .
infixr 8  ^, ^^, **
infixl 7  *, /, `quot`, `rem`, `div`, `mod`
infixl 6  +, -
infixr 5  : ++                                  >.> >. .>
infix  4  ==, /=, <, <=, >=, >                  <*>
infixr 3  &&                                    <|>
infixr 2  ||
infixl 1  >>, >>=
infixr 0  $, $!, `seq`
```

(Der Infix-Typkonstruktor "→"ist übrigens rechtsassoziativ.) Rechts sind die Prioritäten für die Parsing-Kombinatoren genannt.

Auch Wertkonstruktoren können infix geschriebene Operatoren sein, wie die vordefinierte Listenkonstruktion ":". Die Namen von Wertkonstruktions-Operatoren müssen mit einem Doppelpunkt beginnen. Eine alternative Definition für Ausdrücke könnte also lauten:

```
data Expr'   = Expr' :+: Expr'
             | Expr' :-: Expr'
             | Expr' :*: Expr'
             | Expr' :/: Expr'
             | Number Int
               deriving (Eq, Show)
```

9

Ein-Ausgabe

Alle Funktionen in Haskell sind *referentiell transparent*: Die Anwendung einer Funktion F auf ein Argument A liefert stets dasselbe Ergebnis, ist also unabhängig vom Zustand des Systems. Klassische Ein-Ausgabe benutzt Dateien eines Betriebssystems und ist deshalb immer zustandsabhängig. Deshalb kann Ein-Ausgabe – und jeder andere Zugriff auf das Betriebssystem – in einer reinen funktionalen Sprache nicht so einfach realisiert werden.

In diesem Kapitel skizzieren wir zunächst mögliche Realisierungen von Ein-Ausgabe in funktionalen Sprachen und beschäftigen uns dann detailliert mit *Aktionen* (Funktionen mit Seiteneffekten), die in Haskell für die Implementierung der Ein-Ausgabe und anderer Schnittstellen zum Betriebssystem benutzt werden. Mit Aktionen lässt es sich einerseits genau so programmieren wie mit anderen Funktionen, andererseits bietet Haskell eine besondere Schreibweise – die **do**-Notation – mit der Aktionen fast wie imperative oder objektorientierte Programme geschrieben werden können.

9.1 Zustände und Seiteneffekte

Ein-Ausgabe ist das wichtigste Konzept zur Kommunikation zwischen Programmen und ihrer Umgebung, den *Systemen* oder *Plattformen* auf denen sie ausgeführt werden. Alle real existierenden Systeme und Plattformen sind *zustandsbasiert*. Ihr Verhalten lässt sich beschreiben als eine Folge von Befehlen zum Ändern dieses Zustands – gegeben durch das Dateisystem, die Werte in den Registern und im Speicher der unterliegenden Hardware usw.

Befehle wie `getLine` (zum Lesen einer Zeile aus der Standard-Eingabedatei), `putStr` (zum Schreiben einer Zeichenkette ans Ende der Standard-Ausgabedatei), aber auch *`getCurrentTime`* zum Bestimmen der aktuellen Uhrzeit sind nicht referentiell transparent: Verschiedene Ausführungen der Befehle können verschiedene Ergebnisse liefern, selbst für das gleiche Argument. Das liegt daran, dass sie vom Systemzustand abhängen, der für gewöhnlich aber nicht explizit als Argument angegeben wird (und auch nicht explizit als

– weiteres – Ergebnis, wenn ein Befehl den Zustand verändert). Die Veränderung des Systemzustands ist ein *Seiteneffekt*; dies ist ein etwas irreführender Name, denn oft sind die Seiteneffekte das einzig interessante an Befehlen.

In einer rein funktionalen, referentiell transparenten Sprache müssten Ein-Ausgabe-Befehle als Funktionen vom Typ

```
getLine  :: State → (State, String)
putStrLn :: a → State → State
```

definiert werden. Aber das alleine reicht nicht, denn Zustände dürfen nur auf bestimmte Weise verändert werden:

- Zustandsveränderungen müssen immer einer nach dem anderen vollzogen werden, denn das System hat zu jeder Zeit genau einen Zustand. Bei der Anwendung von Funktionen wie `getLine` muss der Zustandsparameter also immer von einem zum nächsten Aufruf *durchgefädelt* werden. Man dürfte also nicht schreiben

  ```
  (st', x) = getLine st
  st''     = putStrLn st x
  ```

  weil damit der Zustand `st` in zwei verschiedene Folgezustände überführt würde.
- Weil Zustandsveränderungen in einer genau festgelegten Reihenfolge passieren sollen, taugt die verzögerte Auswertung für "Befehle" nicht, weil dann unklar wäre, wann – wenn überhaupt – eine Funktion ausgewertet wird. Im Beispiel oben hängt es davon ab, wann die Ergebnisse `(st',x)` und `st''` benötigt werden, welches zuerst berechnet wird.

Ein-Ausgabe (und ähnliche Operationen mit Seiteneffekten) wurden in funktionalen Sprachen verschieden realisiert:

- Die Sprache ML ist nicht referentiell transparent. Also können alle Funktionen prinzipiell Seiteneffekte haben. Dies ist in vielen "frühen" funktionalen Sprachen der Fall, auch in LISP.
- Einige Sprachen haben verzögert ausgewertete Listen, also *Ströme* (*streams*, vgl. 8.3) für die Ein-Ausgabe benutzt.
- Andere benutzen *fortsetzungsbasierte Ein-Ausgabe*, bei der jede E/A-Funktion als zusätzlichen Parameter die danach auszuwertende Funktion erhält, um sicherzustellen, dass die Operationen in einer festgelegten Reihenfolge abgearbeitet werden.
- In der Sprache CLEAN werden Zustandswerte von "normalen" Werten unterschieden und durch die Typüberprüfung sichergestellt, dass die oben genannten Bedingungen gelten (*Uniqueness Typing*).
- In Haskell wird Ein-Ausgabe in einem abstrakten Datentypen gekapselt, dessen Operationen die konsistente Benutzung der Operationen garantieren. Diesem Datentyp liegt das kategorientheoretische Konzept der *Monaden* zu Grunde.

9.2 Aktionen

In Haskell werden die reinen Funktionen strikt getrennt von den Funktionen mit Seiteneffekten, die *Aktionen* genannt werden. Aktionen sind dadurch ausgezeichnet, dass ihr Ergebniswert in den vordefinierten abstrakten Typkonstruktor `IO` (für *input-output*) gekapselt ist.

Die Beispiele für Funktionen mit Seiteneffekten aus dem vorigen Abschnitt werden also zu Aktionen mit folgenden Typen:

- `getLine ::IO String`
- `putStrLn ::String →IO ()`
- *`getCurrentTime`* `::IO` *`ClockTime`*
- `randomIO ::Random a ⇒IO a`

Diese Aktionen sind ausnahmslos vordefiniert: `getLine` und `putStr` im standardmäßig importierten Modul `Prelude`, bzw. *`getCurrentTime`* im Modul *`Data.Time`* und `randomIO` im Modul `System.Random`. Den Typ `IO ()` verwenden wir – wie bei `putStr` – immer dann, wenn eine Aktion keinen Wert liefert, sondern nur eine Zustandsveränderung bewirkt.[1]ebenfalls als `()` geschrieben wird. In Java, C++ und C heißt der Einheitstyp **void**.

Im Folgenden werden wir uns zunächst mit Aktionen für die Ein-Ausgabe beschäftigen. Mit `putStrLn` können wir eine sehr populäre Aktion definieren:

```
main :: IO ()
main = putStrLn "Hello␣World!"
```

Vergleichen wir dies mit einer Funktion, die wir schon vorher schreiben konnten:

```
hello :: String
hello = "Hello␣World!"
```

Wir können sowohl *`main`* als auch *`hello`* im *`hugs`*-Interpreter aufrufen:

```
EinAus> hello
"Hello␣World!"
EinAus> main
Hello World!
```

Die Funktion *`hello`* liefert ein Zeichenketten-Literal, also die Zeichenkette mit den Anführungsstrichen, die Zeichenketten-Literale einschließen; die Aktion *`main`* druckt nur die Zeichenkette selbst.

Die vordefinierten Aktionen für Ein-Ausgabe müssen *kombiniert* werden, wenn interaktive Programme realisiert werden sollen. Dazu gibt es eine Operation *bind*, die in Haskell als binärer Operator `(≫=)` mit folgendem Typ vordefiniert ist:

[1] Zur Erinnerung: Der Typ `()` wird *Einheitstyp* (engl. *Unit*) genannt, weil er einen einzigen Wert enthält, der in Haskell

```
(>>=)  :: IO a → (a → IO b) → IO b
```

Der Aktionskombinator *bind* realisiert die *Komposition* von Aktionen, die der Funktionskomposition entspricht: Für zwei Aktionen c und d führt $c \gg\!= b$ zuerst die Aktion c und dann die Aktion d aus, wobei der Ergebniswert von c (vom Typ a) an den Parameter von d gebunden wird.[2]

Damit können wir eine Aktion realisieren, die eine Zeile einliest und wieder ausgibt:

```
echo1 :: IO ()
echo1 = getLine >>= putStrLn
```

Manchmal ist eine Variante von *bind* nützlich, in der die zweite Aktion kein Ergebnis von der ersten benötigt – zum Beispiel weil es vom Typ `()` ist. Diese Variante wird mit $\gg$ bezeichnet und ist so vordefiniert:

```
(>>) :: IO a → IO b → IO b
c >> d = c >>= \_ → d
```

Damit können wir (die an sich vordefinierte) Aktion `putStrLn` definieren, das wir oben schon benutzt haben:

```
putStrLn :: String → IO ()
putStrLn s = putStr s >> putStr "\n"
```

Aktionen können aufgerufen werden wie Funktionen, auch rekursiv. Damit kann die Eingabe beliebig oft geechot werden.

```
echo :: IO ()
echo = echo1 >> echo
```

Wir können auch eine reine Funktion auf das Ergebnis einer Aktion anwenden. Die Aktion *ohce* echot die eingegebenen Zeilen jeweils in umgekehrter Reihenfolge:

```
ohce :: IO ()
ohce = getLine >>= putStrLn ∘ reverse >> ohce
```

Die Operationen `(>>=)` und `(>>)` sind linksassoziativ und binden schwächer als alle Operationen auf Wahrheitswerten und Zahlen, damit Klammern gespart werden. (Vergleiche Abschnitt 8.5.)

Manchmal wollen wir eine Aktion schreiben, die gar keinen Seiteneffekt hat. Dafür gibt es die vordefinierte Aktion `return`, die jeden Wert vom Typ `a` zu einer Aktion macht, die einen Wert diesen Typs liefert.

```
return :: a → IO a
```

Anwendungen dieser Operation werden wir bald kennen lernen. Jeder "reine" Wert kann (durch `return`) in eine Aktion verwandelt werden, aber nicht

[2] Weil Aktionen den *Befehlen* (engl. *commands*) imperativer und objektorientierter Sprachen entsprechen, bezeichnen wir Aktionen mit c und d.

umgekehrt. Jede Haskell-Funktion mit einem `IO`-Parameter hat auch ein `IO`-Ergebnis. Es gilt also das Prinzip

einmal `IO`, immer `IO`!

Wer einmal das Paradies der reinen Funktionen verlassen hat, muss sich von da an immer mit den Problemen der realen Welt auseinandersetzen.

9.3 Die do-Notation

In Haskell gibt es eine spezielle Schreibweise, in der Aktionen mit den Kombinatoren *bind* und *return* fast wie in imperativen Programmiersprachen geschrieben werden können. In der **do**-Notation werden Aktionen so geschrieben:

$$\begin{aligned} A ::= \mathbf{do}\ & [P_1 \leftarrow]\, E_1 \\ & \vdots \\ & [P_n \leftarrow]\, E_n \\ & E_{n+1} \end{aligned}$$

Hierbei sind (für $1 \leqslant i \leqslant n$) die P_i *Muster* (*pattern*), die wegfallen können. Die E_i sind Aktionsausdrücke von einem Typ $\texttt{IO}\alpha_i$ wenn das Muster P_{i-1} fehlt bzw. vom Typ von $\alpha_{i-1} \to \texttt{IO}\alpha_i$ wenn P_{i-1} vorhanden ist. Die Muster P_i haben den Typ α_i; alle in ihnen auftretenden Variablen werden in der Anweisung $P_i \leftarrow E_i$ definiert und können in den nachfolgenden Ausdrücken E_{i+1} bis E_{n+1} benutzt werden. Der letzte Ausdruck ist entweder ein Aktionsausdruck vom Typ $\texttt{IO}\tilde{}\alpha_{n+1}$ oder ein anderer Ausdruck vom Typ α_{n+1}.

Die Anweisungen $x_i \leftarrow E_i$ ähneln den Wertzuweisungen in imperativen Sprachen; allerdings kann jede "Variable" – die ja eigentlich Parameter von E_{i+1} sind – nur *einmal* ein Wert zugewiesen werden. Wenn ein Variablenname mehrfach im **do**-Ausdruck auftritt, so bezeichnet er jeweils eine andere Variable! Für die Anweisungen im **do**-Ausdruck gilt die Abseitsregel.

Betrachten wir, wie die vorher gezeigten Beispiele sich mit dieser Schreibweise ausdrücken lassen:

$$\begin{aligned} \mathit{echo1} = {} & \mathit{getLine} \gg\!= \\ & \mathit{putStrLn} \end{aligned} \iff \begin{aligned} & \mathit{getLine} \gg\!= \\ & \lambda s \to \mathit{putStrLn}\ s \end{aligned} \iff \begin{aligned} \mathbf{do}\ & s \leftarrow \mathit{getLine} \\ & \mathit{putStrLn}\ s \end{aligned}$$

$$\begin{aligned} \mathit{echo} = {} & \mathit{echo1} \gg \\ & \mathit{echo} \end{aligned} \iff \begin{aligned} \mathbf{do}\ & \mathit{echo1} \\ & \mathit{echo} \end{aligned}$$

$$\begin{aligned} \mathit{ohce} = {} & \mathit{getLine} \gg\!= \\ & \mathit{putStrLn} \circ \mathit{reverse} \gg \\ & \mathit{ohce} \end{aligned} \iff \begin{aligned} \mathbf{do}\ & s \leftarrow \mathit{getLine} \\ & \mathit{putStrLn}\ (\mathit{reverse}\ s) \\ & \mathit{ohce} \end{aligned}$$

Mit Rekursion und Fallunterscheidungen können auch Dialoge gestaltet werden:

```
echo2 :: IO ()
echo2 =  do str ← getLine
            if str /= "quit"
               then do putStrLn str
                       echo2
               else return ()
```

Wegen der Abseitsregel müssen **then** und **else** gegenüber dem **if** eingerückt werden, damit sie als zueinander gehörig erkannt werden.

Grundsätzlich können sowohl *bind* und *return* als auch **do** benutzt werden – auch durcheinander. In den meisten Fällen ist die **do**-Notation übersichtlicher.

9.4 Das Nim-Spiel

Wir wollen nun Ein-Ausgabe und weitere Aktionen benutzen, um die in Beispiel 2.1 entwickelten Funktionen für das *Nim-Spiel* in ein interaktives Programm einzubetten.

Die Benutzungsschnittstelle soll so aussehen:

1. Zu Anfang wird eine Anzahl von Hölzchen "ausgelost" (mit Zufallszahlen).
2. Dann wird eine Eingabe des Spielers eingelesen.
3. Wenn nach dieser Eingabe nicht zu gewinnen ist, soll aufgegeben werden; ansonsten werden Hölzchen gezogen.
4. Wenn nur noch ein Hölzchen übrig ist, hat der Spieler verloren.

Wir verändern die Funktion `wins` aus Beispiel 2.1 so, dass sie `Just` n liefert, wenn n ein gewinnender Zug ist, und `Nothing` sonst.

```
wins :: Int → Maybe Int
wins n = if m == 0 then Nothing else Just m
         where m = (n-1) `mod` 4
```

Ein Spiel mit n Hölzchen kann mit folgender Aktion realisiert werden:

```
play :: Int → IO ()
play n =
  do putStrLn ("Der␣Haufen␣enthält␣"++show n++"␣Hölzchen.")
     if n==1 then putStrLn ("Ich␣habe␣gewonnen!")
       else
         do m ← getInput
            case wins (n-m) of
              Nothing → putStrLn "Ich␣gebe␣auf."
              Just l  → do putStrLn ("Ich␣nehme␣"++show l++"␣Hölzchen.")
                           play (n - m - l)
```

Jetzt braucht nur noch die Benutzereingabe (`getInput`) implementiert zu werden.

```
getInput' :: IO Int
getInput' =
  do putStrLn "Wieviele␣Hölzchen␣nehmen␣Sie?"
     n ← do s ← getLine
            return (read s)
     if n < 1 || n > 3
       then do putStrLn "Falsche␣Zahl!"
               getInput'
       else return n
```

Diese Lösung ist noch unbefriedigend, weil falsche Eingabe (einer Zeichenkette, die keine Zahl darstellt) nicht abgefangen wird, sondern einfach (durch den HASKELL-Interpreter) das Programm beendet wird. Dies können wir durch eine Ausnahmebehandlung ähnlich der von JAVA beheben. Die Aktion

```
ioError :: IOError → IO a
```

wirft einen Ein-Ausgabefehler entspricht dem **throw** von JAVA. Die Aktion

```
readIO :: Read a ⇒ String → IO a
```

meldet einen `IOError`, statt das Programm abzubrechen. Dann kann mit der Aktion

```
catch :: IO a → (IOError → IO a) → IO a
```

ein Ein-Ausgabefehler "gefangen" und behandelt werden.

Eine robuste Benutzereingabe für das Nim-Spiel könnte dann so aussehen:

```
getInput :: IO Int
getInput =
  do putStrLn "Wieviele␣Hölzchen␣nehmen␣Sie?"
     n ← catch (do s ← getLine
                   readIO s    )
               (\_ → do putStrLn "Eingabefehler!"
                        getInput                 )
     if n < 1 || n > 3
       then do putStrLn "Falsche␣Zahl!"
               getInput
       else return n
```

Die Hauptaktion begrüßt den Spieler, "würfelt" die Anzahl der Hölzchen aus und startet das Spiel.

```
main :: IO ()
main = do putStrLn "\nWillkommen␣bei␣Nim!\n"
          n ← randomRIO (5,49)
          play n
```

Die Aktion `randomRIO` ist im Modul `Random` vordefiniert. Zufallszahlen sind Instanzen einer Klasse `Random`:

```
class Random a where
  randomIO  :: IO a
  randomRIO :: (a,a) → IO a
```

Alle Basisdatentypen sind als Instanz dieser Klasse implementiert. Dabei liefert `randomIO` eine beliebige Zahl z des Basisdatentyps, und `randomRIO`(u, o) eine Zufallszahl $u \leqslant z \leqslant o$.

Hier noch ein abschließendes Beispiel zu Zufallswerten: Die Funktion `randStr` liefert eine zufällige Zeichenkette mit bis zu 20 Zeichen:

```
randStr :: Int → IO String
randStr n =
  if n ≤ 0
     then return ""
     else do c ← randomIO
             if isAlpha c
               then do cs ← randStr (n-1)
                       return (c:cs)
               else randStr n

main = do n ← randomRIO (0,20)
          s ← randStr n
          putStrLn s
```

9.5 Ein- Ausgabe von Dateien

Programme müssen auch mehr als eine Datei für Ein-Ausgabe benutzen können. Mit den folgenden vordefinierten Typen und Aktionen können ganze Textdateien (verzögert) gelesen und geschrieben werden:

```
type FilePath = String
readFile              ::  FilePath           → IO String
writeFile, appendFile ::  FilePath → String → IO ()
```

Wir wollen ein Programm schreiben, dass die in einer Datei enthaltenen Zeilen, Wörter und Zeichen zählt. Die erste Fassung lautet:

```
wc     :: String → IO ()
wc fn =
   do file ← readFile fn
      putStrLn("Die␣Datei␣"++ fn ++"␣enthält␣"++
               show (length (lines file)) ++"␣Zeilen,␣"++
               show (length (words file)) ++"␣Wörter␣und␣"++
               show (length file) ++ "␣Zeichen.")
```

Diese Fassung ist ineffizient, weil der komplette Datei-Inhalt im Speicher gehalten werden muss. Die folgende Fassung entspricht eher der “imperativen Programmierweise”:

```
wc'     :: String → IO ()
wc' fn =
    do file ← readFile fn
       let (l,w,c) = cnt (0,0,0) False (dropWhile (isSpace) file)
       putStrLn("Die␣Datei␣"++ fn ++"␣enthält␣"++
                show l ++"␣Zeilen,␣"++
                show w ++"␣Wörter␣und␣"++
                show c ++ "␣Zeichen.")
cnt :: (Int,Int,Int) → Bool → String → (Int,Int,Int)
cnt (l,w,c) _     []     = (l,w,c)
cnt (l,w,c) blank (x:xs)
  | isSpace x && not blank = cnt (l',w+1,c+1) True xs
  | isSpace x &&     blank = cnt (l',w  ,c+1) True xs
  | otherwise              = cnt (l',w  ,c+1) False xs
                             where l' = if x == '\n'
                                        then l+1 else l
```

Die Funktion `cnt` "liest" den Inhalt der Datei einmal von vorne nach hinten, so dass er nicht vollständig im Speicher gehalten werden muss.

Dem Hauptprogramm soll die auszuwertende Datei auf der Kommandozeile übergeben werden; dazu benutzen wir die Aktion `getArgs :: IO [String]` aus dem Modul `System`:

```
main :: IO ()
main = do args ← getArgs
          if null args then error "Filepath␣missing!"
             else wc' (head args)
```

Die Eingabe des Dateinamens ist sicher noch nicht so robust, wie man sich das wünschen würde; für das Beispiel soll es genügen.

Es gibt im Modul `IO` der Standardbibliothek noch weitere Aktionen für Ein-Ausgabe. Damit können Dateien in verschiedenen *Eingabe-Modi* (*read*, *write*, *append* oder *readWrite*, Puffer-Modi (*ungepuffert*, *zeilenweise*, *blockweise*) oder *Such-Modi* (*absolut*, *relativ*, *rückwärts*) bearbeitet werden. Die Ein-Ausgabe wird über *Handles* verwaltet. Vergleiche [PJ+02] zu weiteren Details.

9.6 Kombination von Aktionen

Zusammen mit Fallunterscheidungen (**if** und **case**) und Rekursion unterstützen die Kombinatoren *bind* und *return* (bzw. **do**) schon einen imperativen Programmierstil.

Weil Aktionen aber auch nur Funktionen (mit speziellem Ergebnistyp) sind, können wir weitere Aktionskombinatoren definieren und so weitere Kontrollstrukturen besser definieren als in allen imperativen Programmiersprachen, die keine Funktionen höherer Ordnung unterstützen.

Mit dem vordefinierten Kombinator `sequence` kann man eine Liste von Aktionen nacheinander ausführen:

```
sequence :: [IO a] → IO [a]
sequence [] = return []
sequence (c:cs) = do x  ← c
                     xs ← sequence cs
                     return (x:xs)
```

Als Spezialfall für Typ Aktionen ohne Ergebniswert dient `sequence\_`:

```
sequence_ :: [IO ()] → IO ()
sequence_ [] = return ()
sequence_ (c:cs) = c ≫ sequence_ cs
```

Hier sind nur die Seiteneffekte der Aktionen interessant.

Damit können auch zwei *map*-Kombinatoren und ein *filter*-Kombinator für Aktionen definiert werden:

```
mapM :: (a → IO b) → [a] → IO [b]
mapM f = sequence ∘ map f

mapM_ :: (a → IO ()) → [a] → IO ()
mapM_ f = sequence_ ∘ map f

filterM :: (a → IO Bool) → [a] → IO [a]
filterM p []     = return []
filterM p (x:xs) = do b  ← p x
                      ys ← filterM p xs;
```

(*Filter* stammt aus dem Modul `Monad`, siehe weiter unten.)

Auch liebgewordene Kontrollstrukturen wie Schleifen können selbst definiert werden:

```
while :: IO Bool → IO () → IO ()
while cond act =
    do b ← cond
       if b then do act
                    while cond act
            else return ()

forever :: IO a → IO a
forever c = c ≫ forever c

for :: (a, a → Bool, a → a → a) → (a → IO ()) → IO ()
for (start, cond, incr) cmd =
  iter start where
    iter s = if cond s then cmd s ≫ iter (next s) else return ()
```

Dies wird wegen der mächtigeren Möglichkeit der Rekursion aber nur selten gebraucht.

9.7 Monaden

Schließlich wollen wir nicht die ganze bittere Wahrheit über die Klasse `IO` verschweigen: `IO` ist eine Instanz der *Typ-Konstruktorklasse* `Monad`. Im Gegensatz zu einer einfachen Typklasse ist eine Typ-Konstruktorklasse mit einem Typkonstruktor parameterisiert, im Falle der Klasse `Monad` ist dies eine Variable `m` für einen einstelligen Typkonstruktor. In Haskell gibt man deshalb die *Art* (engl. *kind*) von `m` mit $* \to *$ an, wobei jeder Stern für einen nicht funktionalen Typen steht.

Die Klasse `Monad` hat folgende Schnittstelle und Voreinstellungen:

```
class Monad m where
    return :: a → m a
    (>>=)  :: m a → (a → m b) → m b
    (>>)   :: m a → m b → m b
    fail   :: String → m a

    m >> k  =  m >>= \_ → k        -- default definitions
    fail s  = error s
```

Der Typkonstruktor `IO` ist eine *Instanz* von `Monad`; d. h., `IO a` ist ein Beispiel für die Typkonstruktor-Variable `m` $:: * \to *$. Generall steht der Typ "`m a`" für "Berechnungen" über Werten des Typs `a` – im Falle `IO` sind dies Ein-Ausgabe-Berechnungen.

Ein Monad ist eine Struktur aus der Kategorientheorie. Die Berechnungen eines Monaden bilden eine Halbgruppe bezüglich der Operation `(>@>)`, die *Kleisli-Komposition* genannt wird und etwas abstrakter ist als *bind*:

```
(>@>) :: Monad m ⇒ (a → m b) →
                    (b → m c) →
                    (a → m c)
(f >@> g) x = f x >>= g
```

Das heißt: Die Kleisli-Komposition ist *assoziativ* und *return* ist ihr *neutrales Element*:

```
return >@> f   =   f >@> return   =   f

f >@> (g >@> h)   =   (f >@> g) >@> h
```

Exkurs: Der Typkonstruktor [_] für Listen ist eine Instanz der Typkonstruktorklasse `Monad`.

```
instance Monad [] where
   return x = [x]
   xs >>= f = concat (map f xs)
   fail _ = []
```

Der Kombinator *bind* hat in dieser Instanz den Typ `(>>=) ::[a] →(a →[b]) →[b]`; es ist also ein Kombinator, der mehrere (reine) Ergebnisse einer Berechnung parallel (als Liste) verwaltet. In dieser Instanz treten keine Seiteneffekte auf.

Alternativ zu Listenumschreibungen mit mehreren Generatoren kann man **do** benutzen:

$$[(x,y) \mid x \leftarrow \ell_1, y \leftarrow \ell_2] = \begin{array}{l}\textbf{do}\ x \leftarrow \ell_1 \\ \quad\ y \leftarrow \ell_2 \\ \quad\ \mathit{return}\ (x,y)\end{array}$$

Man kann also beispielsweise die Zahlen von 1 bis 99 so berechnen:

```
is :: [Int]
is = do x ← [0 .. 9]
        y ← [0 .. 9]
        return (10 * x + y)
```

9.8 Haskells Eck

Haskell-Programme mit Aktionen können auch als *stand-alone*-Programme ausgeführt werden. Nehmen wir an, im Modul `Hello.hs` stünde folgendes Programm:

```
module Main where

main :: IO ()
main = putStrLn "Hello␣World!"
```

Dann könnten wir dieses Programm von der Kommandozeile starten mit

```
hof@humboldt:~> runhaskell Hello.hs
Hello World!
```

Wenn wir die Datei `Hello.hs` ausführbar (mit `chmod u+x Hello.hs`) machen und in ihre erste Zeile einfügen

```
#!/user/bin/env runhaskell
```

können wir das Programm sogar so aufrufen:

```
hof@humboldt:~> ./Hello.hs
Hello World!
```

In diesem Fall weichen wir also von der sonst üblichen Praxis ab, dass der "Vorname" einer Datei (`Hello`) und ihr Modulname (`Main`) gleich sind. Statt dessen könnten wir die Zeile "`module Main where`" gleich ganz weglassen.

9.9 Zusammenfassung

In Haskell ist jedes (*stand-alone*) Hauptprogramm (`main`) eine Aktion. Aktionen mit Seiteneffekten werden vom Laufzeitsystem interpretiert bzw. ausgeführt. Aktionen kann man mit (abstrakten) *Shell-Skripts* vergleichen, deren Texte *rein funktional* per Zeichenverkettung erstellt wurde. Seiteneffekte werden durch vordefinierte Aktionen bewirkt – und durch sonst nichts. Für den Nachweis von Korrektheit sollte man `IO` möglichst weit vom "eigentlichen" Programm abtrennen.

Aktionen sind jedenfalls besser als imperative (oder Skript-) Sprachen, weil sie eine strenge Typisierung, Polymorphie und Funktionen höherer Ordnung anbieten.

Es gibt noch weitere IO-Bibliotheken, die hier nicht vorgestellt wurden:

- Parallelverarbeitung durch Threads (`Control.Concurrent`) oder Prozesse (`System.Posix`)
- Graphische Benutzerschnittstellen (GUIs), etc.

(Dies ist Fassung 2.2 von 4. Februar 2010.)

10

Zustand

In Haskell sind alle Funktionen transparent

- Zustand – Inspektion und Veränderung
- Abstrakt: Monaden
- Beispiele: Knoten numerieren, `Maybe`, Listen, Parser, IO

10.1 Funktionen mit Seiteneffekten

- Funktionen `f :: a → b` sind *referentiell transparent*
- referentiell transparente *Inspektion* des Zustands:

```
type State = ...

f' :: a → State → b
```

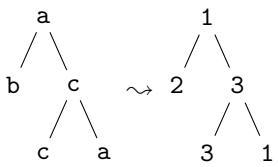
- Zustandsveränderungen verändern den Zustand der Welt: `State → State`
- referentiell transparente *Veränderung* des Zustands:

```
f'' :: a → State → (State, b)
```

- Zustandsveränderungen geschehen sequentiell
- Und wie sieht der Zustand aus? Zum Beispiel eine *Liste*

Beispiel 10.1 (Knoten in Bäumen ersetzen).

- `numTree ::Eq t ⇒ Tree t → Tree Int`
  ersetzt Knoteninhalte durch Zahlen
  (gleiche Knoteninhalte durch gleiche Zahlen)



- Zustand: eine Liste der bisher gefundenen Knoteninhalte
- Diese Liste wird durch den Baum gereicht und verändert
- *numberTree* ::Eq a ⇒*Tree* a →[a] → (*Tree* Int, [a]) transformiert den Baum und verändert den Zustand der Liste
- Numerierung nach Prä-Ordnung

Zustände von Hand transformieren

```
numberNode' :: Eq a ⇒ a → [a]  → (Int,[a])
numberNode' x t = case elemIndex x t of
                    Nothing → (length t, t++[x])
                    Just n  → (n,t)

numberTree' :: Eq a ⇒ Tree a → [a] → (Tree Int, [a])
numberTree'  Leaf t          = (Leaf, t)
numberTree' (Branch x l r) t = (Branch n l' r', t3)
            where (n , t1) = numberNode' x t
                  (l', t2) = numberTree' l t1
                  (r', t3) = numberTree' r t2

numTree' :: Eq t ⇒ Tree t → Tree Int
numTree' t = fst (numberTree' t [])
```

Hierbei werden sie Zustände explizit "durchgeschleift". Geht es auch anders? Ja!

- Zustandstyp

```
data ST s a = ST(s → (a,s))
```

- Zustand unverändert lassen

```
return :: a → ST s a
return x = ST (\s → (x,s))
```

- zwei Zustandsveränderungen verbinden

```
bind :: ST s a → (a → ST s b) → ST s b
(ST m) >>= f = ST $ \s → let (x,s1) = m s
                             ST f' = f x
                            (y,s2) = f' s1
                         in  (y,s2)
```

- abstrakte Baumveränderung

```
data ST s a = ST(s → (a,s))

numberNode :: Eq a ⇒ a → ST [a] Int
numberNode x  = ST (\t → case elemIndex x t of
                    Nothing → (length t, t++[x])
```

```
                    Just n  → (n,t))

numberTree :: Eq a ⇒ Tree a → ST [a] (Tree Int)
numberTree Leaf = return Leaf
numberTree (Branch x l r)
     = numberNode x  >>= \n →
       numberTree l >>= \l' →
       numberTree r >>= \r' →
       return (Branch n l' r')
```

- Zustände reisen *under cover*

10.1.1 Monaden

`return` und *`bind`* sind Klassenmethoden

- eine Typkonstruktorklasse (*`m`* hat einen Parameter)

```
class Monad m where
   (>>=)  :: forall a b. m a → (a → m b) → m b
   return :: a → m a

   (>>)   :: forall a b. m a → m b → m b
   m >> k = m >>= \_ → k

   fail   :: String → m a
   fail s = error s
```

- Eigenschaften von Monaden

```
return a >>= k  ==  k a
m >>= return  ==  m
m >>= (\x → k x >>= h)  ==  (m >>= k) >>= h
```

`ST s` ist eine Instanz von `Monad`

- ```
  instance Monad (ST s) where
  -- (>>=) :: forall a b. ST s a
  -- -> (a -> ST s b) -> ST s b
      (ST m) >>= f = ST (\s → let (x,s1) = m s
                                  ST f'  = f x
                              in f' s1)

  -- return :: a -> ST s a
      return x = ST (\s → (x,s))
  ```

syntaktischer Zucker für `return` und (≫=) die **do**-Notation
  ```

- die **do**-Notation für Monaden

$$\begin{array}{ll} \textbf{do}\ [P_1 \leftarrow]\, E_1 & \\ \quad\vdots & \\ \quad [P_n \leftarrow]\, E_n & \\ \quad \textbf{return}\ E_{n+1} & \end{array} \quad] \quad \equiv \quad [\quad \begin{array}{l} E_1 \gg [= \lambda P_1 \rightarrow] \\ \quad\vdots \\ E_n \gg [= \lambda P_n \rightarrow] \\ \textbf{return}\ E_{n+1} \end{array}$$

numberTree, syntaktisch verzuckert:

```
numberTree :: Eq a ⇒ Tree a → ST [a] (Tree Int)
numberTree Leaf = return Leaf
numberTree (Branch x t1 t2)
    = do n  ← numberNode x
         nt1 ← numberTree t1
         nt2 ← numberTree t2
         return (Branch n nt1 nt2)
```

- Zustände reisen immer noch *under cover*
- imperative Schreibweise

Weitere Instanzen von `Monad`

- `Maybe`
- Listen
- Parser
- Ein-Ausgabe

Die Instanz `Monad Maybe`

```
instance  Monad Maybe  where
-- (>>=) :: forall a b. Maybe a
-- -> (a -> Maybe b) -> Maybe b
    (Just x) >>= k   =  k x
    Nothing  >>= k   =  Nothing

-- return :: a -> m a
    return           =  Just

    fail s           =  Nothing
```

unify mit **do**

```
unify :: Type → Type → Maybe [(Int, Type)]
unify t1 t2 = case firstMatch t1 t2 of
  Nothing → Nothing
  Just [] → Just []
  Just s@((i, t) : _) → case unify (inst s t1) (inst s t2) of
    Just subst → Just $ (i, inst subst t) : subst
    Nothing → Nothing
```

```
unify' :: Type → Type → Maybe [(Int, Type)]
unify' t1 t2 = do s1 ← firstMatch t1 t2
                  if null s1 then return []
                     else do s2 ← unify' (inst s1 t1) (inst s1 t2)
                             return (head s1 : s2)
```

Die Instanz `Monad []`

```
instance  Monad []  where
-- (>>=) :: forall a b. [a] -> (a -> [b]) -> [b]
    m >>= k          = concat (map k m)

-- return :: a -> [a]
    return x         = [x]

    fail s           = []
```

- Listen mit (≫=) und **do**

```
l1 = [(x,y) | x ← [1,2,3], y ←['a','b']]

l2 = [1,2,3] >>= (\x → ['a','b'] >>= (\y → return (x,y)))

l3 = do {x ← [1,2,3]; y ←['a','b']; return (x,y)}
```

Parser, noch mal

- diesmal richtig
- Parsec (Daan Leeijen, Utrecht): *Text.Parsec.Combinator*

```
data Parse a b = Parser (State a → Reply a b)

data State a = State [a] Int  -- Position

data Reply a b = OkReply b (State a) Consumption
               | FailReply Consumption String  h

data Consumption = NoInputConsumed | Consumed
                 deriving (Eq, Ord)
```

Instanz `Monad (Parse a)`

```
instance Monad (Parse a) where
-- (>>=) :: Parse a b -> (b -> Parse a c) -> Parse a c
   Parser p1 >>= f = Parser ( \s → case p1 s of
     OkReply b t con1 →
       let Parser p2 = f b
       in case p2 t of
```

```
          OkReply c u con2 → OkReply c u (max con1 con2)
          FailReply con2 msg → FailReply (max con1 con2) msg
      FailReply con1 e → FailReply con1 e)

-- return :: b -> Parse a b
   return b =  Parser ( \ s → OkReply b s NoInputConsumed)

-- fail :: String -> Parse a b
   fail s = Parser ( \ _ → FailReply NoInputConsumed s)
```

Ausdrucks-Parser

```
data Expr = Expr InfixOp Expr Expr | Number Int deriving ...
data InfixOp = Div | Mult | Plus | Minus deriving ...

expr, term, factor :: CharParser Expr
expr = term >>= opExpr

opExpr, opTerm :: Expr → CharParser Expr  -- S
opExpr t =
     infixExpr Plus t "+" expr
 <|> infixExpr Minus t "-" expr
 <|> return t

term = factor >>= opTerm

opTerm f =
     infixExpr Mult f "*" term
 <|> infixExpr Div f "/" term
 <|> return f

factor =
      liftM Number (lexeme number)
  <|> (symbol "(" >> expr << symbol ")")

infixExpr :: InfixOp → Expr → String → CharParser Expr
                                     → CharParser Expr
infixExpr i e s p =  liftM (Expr i e) (symbol s >> p)
```

Ein- Ausgabe

- Veränderung der *Welt* (*vulgo* Systemzustand)

```
data IO a = ...  -- World -> (World,a)

instance Monad IO where
   (>>=)  = ...
   return = ...
```

```
fail s = ioError (userError s)
```

- Die Welt wird man nicht mehr los
- einmal `IO`, immer `IO`

Nächstes Mal

- Felder (`array`)
- Graphen

(Dies ist Fassung 2.0 von 4. Februar 2010.)

11

Feld und Graph

Nachdem wir im letzten Kapitel untersucht haben, wie imperativer Kontrollfluss beim funktionalen Programmieren realisiert werden kann (mit Zustands-Transformern und Monaden), wollen wir in diesem Kapitel Dattenstrukturen betrachten, die weithin als "*typisch imperativ*" gelten:

- Felder (*array*)
- Graphen

Auch bei den Datentypen können die imperativen Konzepte gut modelliert werden – sogar etwas allgemeiner und abstrakter – aber nur mit einem gewissen *Overhead* implementiert werden, was den Zeit- und Platzbedarf angeht.

11.1 Feld

Felder sind eine wichtige Datenstruktur in imperativen Sprachen – es war die erste Datenstruktur überhaupt, die es schon in Sprachen wie *Fortran* gabb, lange vor Strukturen ("**struct**", "**record**") oder gar rekursiven Strukturen. Das liegt auch daran, dass Felder sehr gut auf die Speicherstruktur herkömmlicher Hardware abgebildet werden können.

11.1.1 Felder in imperativen Sprachen

In der Sprache Ada können Felder so vereinbart werden:

x : **array** $(lwb..upb)$ **of** T

Die Werte des Feldes x können als zusammenhängende Speicherzellen allokiert werden. Dabei entsteht nur geringer *Overhead* für Speicherplatz, besonders wenn die Größen der Werte des Elementtyps T und die Grenzen lw, upb des Indexbereichs bekannt ("*statisch*") sind.

- Elemente werden zusammenhängend gespeichert
  (keine Zeiger, kein Speicher-*Overhead*))
- Zugriff $a[i]$ ist *random* in $\mathcal{O}(1)$
- Verändern $a[i] \leftarrow a[i] + 1$ ist *destruktiv* und kostenneutral

11.1.2 Felder in funktionalen Sprachen

- *functional arrays*
  Zugriff $\mathcal{O}(\log n)$, Verändern $\mathcal{O}(\log n)$, Zeiger-Overhead
- gekapselte imperative Felder
- gekapselte imperative Felder in Zustandsmonade
- *Was bleibt*: *Overhead* für Abstraktion

- ein abstrakter Datentyp
- spezielle tabellierte Abbildungen `Map a b`:
  - Der Definitionsbereich `a` lässt sich auf einen Bereich $[u..o]$ ganzer Zahlen abbilden.
  - In Haskell bedeutet das: Der Definitionsbereich muss *indizierbar* sein, also eine Instanz der vordefinierten Klasse `Ix`:

    ```
    class  Ord a ⇒ Ix a  where
        range       :: (a,a) → [a]
        index       :: (a,a) → a → Int
        inRange     :: (a,a) → a → Bool
        rangeSize   :: (a,a) → Int
    ```

  - Die folgenden vordefinierten Typen sind Instanzen von `Ix`: `Char`, `Int`, `Integer`, Tupel, `Bool`
  - Für Aufzählungstypen und Produkttypen kann eine Instanz abgeleitet werden ("**deriveing**").

- stark vereinfacht gegenüber Haskell-Feldern

```
arr :: (Int,Int) → a → Array a
(!) :: Array a → Int→ a
(//) :: Array a → (Int,a) → Array a
```

Implementierung von *functional arrays*

- Implementiert als annähernd balancierte Bäume
- Zugriff mit Division durch 2

```
data Array a = A (Int,Int) (Tree a) deriving (Show, Read)

data Tree a = Leaf | Branch a (Tree a) (Tree a) deriving (Show, Read)
```

```
arr (l,u) v = A (l,u) (nodes size)
    where size = if u-l+1 ≥ 0 then u-l+1 else  error ".."
          nodes n = if n == 0 then Leaf
                      else Branch v (nodes (n `div` 2))
                                    (nodes (n `div` 2))

A (l,u) t ! k = if l ≤ k && k ≤ u then t `at` (k-l+1)
                                  else error "array␣bounds␣error"
   where Leaf `at`          _ = error "wrong␣index"
         Branch v t1 t2 `at` k
          | k == 1    = v
          | even k    = t1 `at` (k `div` 2)
          | otherwise = t2 `at` (k `div` 2)

A (l,u) t // (k,v) = if l ≤ k && k ≤ u
                     then A (l,u) (upd t (k-l+1) v)
                     else error "array␣bounds␣error"
   where upd :: Tree a → Int→ a→ Tree a
         upd Leaf k v =
             if k== 1 then (Branch v Leaf Leaf)
             else error "Tree.upd:␣illegal␣index"
         upd (Branch w t1 t2) k v
            | k == 1    = Branch v t1 t2
            | even k    = Branch w (upd t1 (k `div` 2) v) t2
            | otherwise = Branch w t1 (upd t2 (k `div` 2) v)

data  (Ix a)    ⇒ Array a b = ... -- Abstract

array      :: (Ix a) ⇒ (a,a) → [(a,b)] → Array a b
(!)        :: (Ix a) ⇒ Array a b → a → b
bounds     :: (Ix a) ⇒ Array a b → (a,a)
indices    :: (Ix a) ⇒ Array a b → [a]
elems      :: (Ix a) ⇒ Array a b → [b]
assocs     :: (Ix a) ⇒ Array a b → [(a,b)]
accumArray :: (Ix a) ⇒ (b → c → b) → b → (a,a) → [(a,c)]
                                 → Array a b
(//)       :: (Ix a) ⇒ Array a b → [(a,b)] → Array a b
accum      :: (Ix a) ⇒ (b → c → b) → Array a b → [(a,c)]
                       → Array a b
```

- Instanz von `Functor` (*fmap*), `Eq`, `Ord`, `Show`, `Read`

Implementierung von `Array`

- Implementiert als imperative Felder

- Zugriff $\mathcal{O}(1)$ (mit *Overhead* wegen Indexprüfung)
- Verändern $\mathcal{O}(n)$ (!)

Beispiele

```
a' = array (1, 4) [(3, 'c'), (2, 'a'), (1, 'f'), (4, 'e')]

f n = array (0, n) [(i, i*i) | i ← [0..n]]

m = array ((1, 1), (2, 3))
          [((i, j), (i*j)) | i ← [1..2], j ← [1..3]]

fibs n = a
    where a = array (1,n) ((1, 1):(2, 1):
                [(i, a!(i-1) + a!(i-2)) | i ← [3..n]])
```

11.2 Graph

- endliche Menge von Knoten
- Kanten verbinden Knoten (gerichtet/ungerichtet, mit Gewicht)
- Knoten können beliebig viele Vorgänger haben
- Zyklen und unverbundenen Komponenten sind möglich

Schnittstelle

```
mkGraph :: (Ix n,Num w) ⇒ Bool → (n,n) → [(n,n,w)] → (Graph n w)
adjacent :: (Ix n,Num w) ⇒ (Graph n w) → n → [n]
nodes :: (Ix n,Num w) ⇒ (Graph n w) → [n]
edgesU :: (Ix n,Num w) ⇒ (Graph n w) → [(n,n,w)]
edgesD :: (Ix n,Num w) ⇒ (Graph n w) → [(n,n,w)]
edgeIn :: (Ix n,Num w) ⇒ (Graph n w) → (n,n) → Bool
weight :: (Ix n,Num w) ⇒ n → n → (Graph n w) → w
```

Graphen als Zeigerstrukturen

```
data Graph n w = Vertex n [((Graph n w),w)]

graphPTR = v1
     where
       v1 = Vertex 1 [(v2,12),(v3,34),(v5,78)]
       v2 = Vertex 2 [(v1,12),(v4,55),(v5,32)]
       v3 = Vertex 3 [(v1,34),(v4,61),(v5,44)]
       v4 = Vertex 4 [(v2,55),(v3,61),(v5,93)]
       v5 = Vertex 5 [(v1,78),(v2,32),(v3,44),(v4,93)]
```

- Probleme: unzusammenhängende Graphen, Zyklen

Graphen als Adjazenzlisten

```
type Graph n w = Array n [(n,w)]

graphAL = array (1,5) [(1,[(2,12),(3,34),(5,78)]),
                       (2,[(1,12),(4,55),(5,32)]),
                       (3,[(1,34),(4,61),(5,44)]),
                       (4,[(2,55),(3,61),(5,93)]),
                       (5,[(1,78),(2,32),(3,44),(4,93)])]
```

Operationen auf Graphen als Adjazenzlisten

```
mkGraph dir bnds es =
    accumArray (\xs x → x:xs) [] bnds
               ([(x1,(x2,w)) | (x1,x2,w) ← es] ++
                if dir then []
                else [(x2,(x1,w))|(x1,x2,w)←es,x1/=x2])

adjacent g v    = map fst (g!v)
nodes g         = indices g
edgeIn g (x,y)  = elem y (adjacent g x)
weight x y g    = head [ c | (a,c)←g!x , a==y]
edgesD g        = [(v1,v2,w) | v1← nodes g , (v2,w) ←g!v1]
edgesU g        = [(v1,v2,w) | v1← nodes g , (v2,w) ←g!v1 , v1 <
v2]
```

Graphen als Adjazenzmatritzen

```
type Graph n w = Array (n,n) (Maybe w)

graphAM = mkGraph True (1,5) [(1,2,10),(1,3,20),(2,4,30),(3,4,40),(4,5,50)]
```

Operationen auf Graphen als Adjazenzmatritzen

```
adjacent g v1 = [ v2 | v2 ←nodes g,(g!(v1,v2))/= Nothing]

nodes g       = range (l,u) where ((l,_),(u,_)) = bounds g

edgeIn g (x,y)= (g!(x,y)) /= Nothing

weight x y g  = w where (Just w) = g!(x,y)

edgesD g       = [(v1,v2,unwrap(g!(v1,v2)))
                       | v1 ←nodes g, v2 ← nodes g,
                         edgeIn g (v1,v2)]
```

```
    where unwrap (Just w) = w

edgesU g       = [(v1,v2,unwrap(g!(v1,v2)))
                  | v1 ←nodes g, v2 ← range (v1,u),
                    edgeIn g (v1,v2)]
    where (_,(u,_)) = bounds g
          unwrap (Just w) = w
```

Adjazenzlisten *versus* Adjazenzmatritzen

- Hängt von der *Dichte* des Graphen ab
- Adjazenzlisten sind beser für dünn besetzte Graphen
- Adjazenzmatritzen sind besser für dicht besetzte Graphen
- *Dünn besetzt*: $|E| < |V| \log |V|$

Einige Graphalgorithmen

- Tiefensuche
- Breitensuche
- topologisches Sortieren
- minimale aufspannende Bäume (Kruskal, Prim)

Topologisches Sortieren

- geordnete Graphen

Beispiel

```
g = mkGraph True (1,6) [(1,2,0),(1,3,0),(1,4,0),
                        (3,6,0),(5,4,0),(6,2,0),
                        (6,5,0)]
```

minimal aufspannende Bäume

Kruskals Algorithmus

```
kruskal :: (Num w, Ix n, Ord w) ⇒ Graph n w → [(w,n,n)]
kruskal g = kruskal' (fillPQ (edgesU g) emptyPQ)
                     (newTable [(x,x) | x← nodes g])
                     [] 1
   where n = length (nodes g)
         kruskal' pq t mst i
           | i==n    = mst
           | otherwise =
               let e@(_,x,y)    = frontPQ pq
                   pq'          = dePQ pq
```

```
          (updated,t') = unionFind (x,y) t
      in if updated
         then kruskal' pq' t'(e:mst) (i+1)
         else kruskal' pq' t  mst     i
```

minimal aufspannende Bäume

Prims Algorithmus

```
prim :: (Num w, Ix n, Ord w) ⇒ Graph n w → [(w,n,n)]
prim g = prim' [n] ns []
   where (n:ns) = nodes g
         es = edgesU g
         prim' t [] mst = mst
         prim' t r mst
           = let e@(c,u',v') = minimum [(c,u,v)|(u,v,c)←es,
                                        elem u t, elem v r]
             in prim' (v':t) (delete v' r) (e:mst)
```

Nächstes Mal

- Funktional, praktisch
- von Jasper van de Ven

(Dies ist Fassung 0.9 von 4. Februar 2010.)

12

Beweise und Typen

Dieses Kapitel ist zweigeteilt. Als erstes wenden wir uns dem Thema zu, wie wir Eigenschaften von funktionalen Programmen *beweisen* können. Aufwandsabschätzungen werden wir erst später betrachten. Es zeigt sich, dass der systematische Aufbau von Funktionen mit primitiver oder struktureller Rekursion dazu führt, dass man ihre Eigenschaften ebenso systematisch mit vollständiger, struktureller bzw. Fixpunkt-Induktion beweisen kann.

Als zweites beschäftigen wir uns damit, wie Typen in einer funktionalen Sprache behandelt werden, insbesondere, wie überprüft wird, ob ein Programm den Typregeln genügt. In modernen funktionalen Sprachen müssen die Typen von Funktionen in den allermeisten Fällen nicht angegeben werden, sondern können aus der Definition hergeleitet werden. Das bezeichnet man als *Typinferenz*.

12.1 Formalisierung und Beweis

Wie können wir verstehen, was eine Funktion tut? Auf diese Frage sind mehrere Antworten möglich:

- *Test am Rechner*: Wir können die Funktion für verschiedene Argumente durch Auswertung in `ghci` bestimmen lassen.
- *Test von Hand*: Wir können das Gleiche auch von Hand auf Papier tun, indem wir die Definition Schritt für Schritt anwenden. Das geht natürlich nur für einfachere Funktionen und sehr kleine Eingaben.
- *Beweis*: Wir können untersuchen, wie sich die Funktion im Allgemeinen verhält.

In diesem Kapitel werden wir versuchen, allgemeine Eigenschaften von Funktionen nachzuweisen.

Betrachten wir ein sehr einfaches Beispiel:

```
length []     = 0                                   (length.1)
length (x:xs) = 1 + length xs                       (length.2)
```

Dann können wir die Länge einer gegebenen Liste wie $[2,3,1]$ bestimmen:

$$\begin{aligned} \mathit{length}\ [2,3,1] &\leadsto 1 + \mathit{length}\ [3,1] \\ &\leadsto 1 + (1 + \mathit{length}\ [1]) \\ &\leadsto 1 + (1 + (1 + \mathit{length}\ [])) \\ &\leadsto 1 + (1 + (1 + 0)) \\ &\leadsto^3 3 \end{aligned}$$

Wir können die Gleichungen `(length.1)` und `(length.2)` auch als allgemeine Beschreibungen des Verhaltens von `length` begreifen:

- `(length.1)` definiert die Länge der leeren Liste.
- `(length.2)` sagt aus, dass die Länge einer Liste $(x : xs)$ gleich $1 + \mathit{length}\, xs$ ist, *für beliebige Werte* x und xs.

Die zweite Gleichung definiert eine allgemeine Eigenschaft von `length`, die für alle nicht leeren Listen gilt. Auf Basis der Gleichungen können wir schließen, dass

$$\mathit{length}[x] = 1 \qquad \texttt{(length.3)}$$

Wie geht das? Wir wissen, dass die Gleichung `(length.2)` für beliebige Listen xs gilt, also auch für $xs = []$. Also gilt:

$$\begin{aligned} \mathit{length}[x] &= \mathit{length}(x : []) && \text{Def. von } [x] \\ &= 1 + \mathit{length}[] && \text{wegen } \texttt{(length.2)} \\ &= 1 + 0 && \text{wegen } \texttt{(length.1)} \\ &= 1 \end{aligned}$$

Wir lernen hieraus, dass Funktionsdefinitionen auf mindestens zwei verschiedene Arten gelesen werden können:

- Wir können sie als Vorschriften nehmen, wie bestimmte Ergebnisse zu berechnen sind, wie $\mathit{length}\ [2,3,1]$.
- Wir können sie als allgemeine Beschreibungen des Verhaltens der Funktion verstehen.

Aus der allgemeinen Beschreibung können wir weitere Fakten herleiten. Einige – wie `(length.3)` – sind sehr einfach, andere können komplexere Zusammenhänge zwischen zwei oder mehr Funktionen beschreiben. Dazu später mehr.

Eine andere Sichtweise des "Beweises" von `(length.3)` ist, dass wir *symbolische Auswertung* betrieben haben: Wir haben die Definition angewendet, aber nicht auf konkrete Zahlen, sondern auf eine Variable x, einen Platzhalter für beliebige Zahlen. Symbolische Auswertung ist für Beweise sehr nützlich.

Abschließend stellen wir fest, dass die Gleichungen einer Funktionsdefinition "die Funktion selbst beschreiben", was Beweise sehr erleichtert. Man vergleiche nur einmal, wie man die oben gezeigten Eigenschaften für eine Implementierung der Längenfunktion in Sprachen wie JAVA oder C++ nachweisen könnte!

12.1.1 Definiertheit, Termination und Endlichkeit

Bevor wir uns weiter mit Beweisen beschäftigen, müssen über zwei Aspekte des Programmierens nachdenken, die wir bislang nur kurz gestreift haben.

Bei der Auswertung eines Ausdrucks kann zweierlei herauskommen:

- Die Auswertung hält an und liefert eine Antwort.
- Die Auswertung kann immer weiter gehen, ohne je anzuhalten und eine Antwort zu liefern.

Bei der Definition

```
fac:: Int → Int
fac 1 = 1
fac n = n * fac (n-1)
```

gibt es beide Arten von Auswertungen:

$$\begin{aligned} fac\ 2 &\rightsquigarrow 1 * fac\,1 \\ &\rightsquigarrow 1 * (1 * fac\,0) \\ &\rightsquigarrow 1 * (1 * 0) \\ &\rightsquigarrow 2 \end{aligned} \qquad \begin{aligned} fac(-2) &\rightsquigarrow (-2) * fac(-3) \\ &\rightsquigarrow (-2) * ((-3) * fac(-4)) \\ &\rightsquigarrow (-2) * ((-3) * (-4 * fac(-5))) \\ &\rightsquigarrow \ldots \end{aligned}$$

Wenn die Auswertung nicht terminiert, betrachten wir ihr Ergebnis als *undefiniert*, geschrieben "$\bot$". Wenn wir Beweise führen, müssen wir uns oft auf die Fälle beschränken, in denen die Auswertung definiert ist , weil andernfalls viele vertraute Eigenschaften nicht mehr gelten. Zum Beispiel erwarten wir, dass gilt:

$$0 * E \Longleftrightarrow 0$$

Das gilt aber nur, solange sie Auswertung von E definiert ist:

$$0 * fac(-2) \Rightarrow 0 * \bot \Rightarrow \bot \neq 0$$

In den meisten Fällen reicht es aus, nur definierte Fälle einer Auswertung zu betrachten – es sei denn, eine Funktion wäre undefiniert für Werte, die man eigentlich definieren wollte.

Der zweite Aspekt ist die Endlichkeit der Werte. In HASKELL werden Funktionen verzögert ausgewertet, so dass sehr wohl auch *unendliche Werte* definiert werden können, wie die Liste der natürlichen Zahlen

$$numbers = [1..] \rightsquigarrow [1, 2, 3, \ldots]$$

Auch partiell definierte Listen können definiert werden. Im Folgenden beschränken wir uns zunächst darauf, Eigenschaften nur für alle *endlichen* Werte zu zeigen, z.B. für alle endlichen Listen.

12.1.2 Rekursion und Induktion

Viele Funktionen sind mit *Fallunterscheidung* und *Rekursion* definiert. Die Struktur der Definition folgt dabei oft der Struktur des Arguments der Funktion f.

- Für mindestens einen Basisfall ist die Definition nicht rekursiv. Ist das Argument eine natürliche Zahl n, ist dies beispielsweise der Fall $n = 0$ oder $n = 1$. Ist das Argument eine Liste ℓ, könnte dies der Fall $\ell = []$ sein. Wenn das Argument ein Baum t ist, könnte dies der Fall $t = \mathit{Leaf}$ sein.
- Die rekursiven Fälle definieren den Wert von f für ein Argument, indem sie rekursiv Werte von f für Argumente benutzen, die im gewissen Sinne *kleiner* sind. So kann $f\,n$ für eine natürliche Zahl $n > 0$ mithilfe des Wertes von $f(n-1)$ bestimmt werden (bzw. für $f\,m$ für eine Zahle $m < n$). Der Wert $f(x : \ell)$ für eine nicht leere Liste kann mithilfe des Wertes $f(n-1)$ berechnet werden. Schließlich kann der Wert von $f(\mathit{Branch}\ t_1\,x\,t_2)$ für einen Knoten rekursiv mithilfe der Werte $f\,t_1$ und $f\,t_2$ definiert werden.

In der Rekursion wird die Aufgabe also von einem größeren auf ein kleinerer Argument reduziert. Bei mehreren Argumenten gilt dies analog. Siehe auch Tabelle 12.1.

Rekursion	*Verankerung*	*Schritt*
primitiv (über $\mathbb{N}$)	$f\,0 = \ldots$	$f\,n = \ldots f(n-1)\ldots$
über Listen	$f\,[] = \ldots$	$f(x : \ell) = \ldots f\,\ell\ldots$
über Bäumen	$f\ \mathit{Leaf} = \ldots$	$f(\mathit{Branch}\ t_1\,x\,t_2) = \ldots f\,t_1\ldots f\,t_2\ldots$

Tabelle 12.1. Schemata für rekursive Funktionsdefinitionen

Wenn man eine Eigenschaft P für beliebige Argumente x einer Funktion nachweisen will, tut man dies oft durch *Induktion*:

- Als *Induktionsanfang* zeigt man P für die Basisfälle.
- Die *Induktionsannahme* besagt, dass $P(a)$ für ein beliebiges Argument gelten möge.
- Im *Induktionsschritt* muss dann gezeigt werden, dass aus der Induktionsannahme $P(a)$ folgt, dass $P(a')$ für ein *größeres* Argument a' gilt.

Bei der Induktion wird also vom Kleineren aufs Größere geschlossen.

12.1.3 Beweis durch vollständige Induktion

Wenn eine Eigenschaft für beliebige natürliche Zahlen gezeigt werden soll, benutzt man *vollständige Induktion*. Zu zeigen ist also:

Für alle Zahlen $n \in \mathbb{N}$ gilt $P(n)$.

Der Beweis gliedert sich so:

- Als Induktionsanfang ist zu zeigen: $P(0)$ (oder $P(1)$).
- Im Induktionsschritt muss nachgewiesen werden, dass aus der Annahme, dass $P(n)$ für ein beliebiges $n \in \mathbb{N}$ schon gilt, auch $P(n+1)$ folgt.

Lemma 12.1. *Für beliebige Zahlen $n \in \mathbb{N}$ gilt*

$$\mathit{fac}\ n = \Pi_{i=1}^{n} i = 1 * \cdots * (n-1) * n \tag{12.1}$$

Beweis. (Durch vollständige Induktion)

- *Induktionsanfang*: $\mathit{fac}\ 1 = \Pi_{i=1}^{1} = 1$.
- *Induktionsannahme*: Für beliebige $n > 1$ gelte Gleichung 12.1.
- *Induktionsschritt*:

$$\begin{aligned}
\mathit{fac}(n+1) &= (n+1) * \mathit{fac}\ n && \text{Def. von } \mathit{fac}\\
&= (n+1) * \Pi_{i=1}^{n} i && \text{Ind.-Ann.}\\
&= (n+1) * 1 * \cdots * n && \text{Def. von } \Pi\\
&= 1 * \cdots * n * (n+1) && \text{Komm. von } (*)\\
&= \Pi_{i=1}^{n+1} i && \text{Def. von } \Pi
\end{aligned}$$

12.1.4 Beweis durch strukturelle Induktion

Bei zusammengesetzten Datentypen werden Beweise über den Aufbau der Werte geführt. Bei *Listen* ist zu zeigen:

Für alle Listen ℓ gilt $P(\ell)$

Der Beweis gliedert sich so:

- Als Induktionsanfang zeigen wir, dass $P([])$ gilt.
- Dann zeigen wir im Induktionsschritt, dass unter der Annahme $P(\ell)$ gelte für eine beliebige Liste, folgt dass $P(x : \ell)$ gilt, und zwar für beliebige x.

Als ein einfaches Beispiel betrachten wir die Verkettung und die Länge von Listen. Die Funktionen waren so definiert:

```
(++) :: [a]→ [a]→ [a]
[] ++ ys      = ys
(x:xs) ++ ys = x:(xs++ ys)

length :: [a] → Int
length []     = 0
length (x:xs) = 1 + length xs
```

Den folgenden Zusammenhang hatten wir in Abschnitt 3.1 schon behauptet.

Lemma 12.2. *Für beliebige Listen ℓ und ℓ' gilt:*

$$\mathit{length}(\ell +\!\!+ \ell') = \mathit{length}\,\ell + \mathit{length}\,\ell' \tag{12.2}$$

Beweis. Durch Induktion über ℓ. Als Induktionsanfang nehmen wir an, dass $\ell = []$. (Die Liste ℓ' ist beliebig.) Dann gilt:

$$\begin{aligned} \mathit{length}([] +\!\!+ \ell') &= \mathit{length}\,\ell' && \text{Def. von } +\!\!+ \\ &= 0 + \mathit{length}\,\ell' && \text{Def. von } + \\ &= \mathit{length}[] + \mathit{length}\,\ell' && \text{Def. von } \mathit{length} \end{aligned}$$

Im Induktionsschritt nehmen wir an, zwei beliebige Listen ℓ und ℓ' erfüllten Gleichung 12.2. Dann gilt für alle Elemente x:

$$\begin{aligned} \mathit{length}((x : \ell) +\!\!+ \ell') &= \mathit{length}(x : (\ell +\!\!+ \ell')) && \text{Def. von } +\!\!+ \\ &= 1 + \mathit{length}(\ell +\!\!+ \ell') && \text{Def. von } \mathit{length} \\ &= 1 + \mathit{length}\,\ell + \mathit{length}\,\ell' && \text{Induktionsannahme} \\ &= \mathit{length}(x : \ell) + \mathit{length}\,\ell' && \text{Def. von } \mathit{length} \end{aligned}$$

Ein weiteres Beispiel beweist einen Zusammenhang zwischen Verkettung und Umdrehen von Listen. Die Umkehr von Listen ist so definiert:

```
rev        :: [a] → [a]
rev []      = []
rev (x:xs) = rev xs ++ [x]
```

Lemma 12.3. *Für beliebige Listen ℓ und ℓ' gilt:*

$$\mathit{rev}(\ell +\!\!+ \ell') = \mathit{rev}\,\ell' +\!\!+ \mathit{rev}\,\ell \tag{12.3}$$

Beweis. Durch Induktion über ℓ. Wieder ist $\ell = []$ der Induktionsanfang:

$$\begin{aligned} \mathit{rev}([] +\!\!+ \ell') &= \mathit{rev}\,\ell' && \text{Def. von } +\!\!+ \\ &= \mathit{rev}\,\ell' +\!\!+ [] && \text{Lemma 12.4} \\ &= \mathit{rev}\,\ell' +\!\!+ \mathit{rev}[] && \text{Def. von } \mathit{rev} \end{aligned}$$

Die Induktionsannahme ist, dass Gleichung 12.3 für beliebige Listen ℓ und ℓ' gelte. Dann ist der Induktionsschritt:

$$\begin{aligned} \mathit{rev}((x : \ell) +\!\!+ \ell') &= \mathit{rev}(x : (\ell +\!\!+ \ell')) && \text{Def. von } +\!\!+ \\ &= \mathit{rev}(\ell +\!\!+ \ell') +\!\!+ [x] && \text{Def. von } \mathit{rev} \\ &= \mathit{rev}(\ell' +\!\!+ \mathit{rev}\,\ell) +\!\!+ [x] && \text{Induktionsannahme} \\ &= \mathit{rev}(\ell') +\!\!+ (\mathit{rev}\,\ell +\!\!+ [x]) && \text{Lemma 12.5} \\ &= \mathit{rev}(\ell') +\!\!+ (\mathit{rev}(x : \ell) && \text{Def. von } \mathit{rev} \end{aligned}$$

Eigenschaften von Funktionen auf Bäumen oder anderen rekursiven Datentypen können analog durch strukturelle Induktion über die Definition dieser Datentypen gezeigt werden.

Aufgabe

Zeigen Sie folgende Eigenschaften:

Lemma 12.4. $[]$ *ist das neutrale Element bezüglich der Listenverkettung* $(++)$*:*

$$\forall \ell \in [\alpha] : [] ++ \ell = \ell = \ell ++ []$$

Lemma 12.5. $(++)$ *ist assoziativ:*

$$\forall \ell_1, \ell_2, \ell_3 \in [\alpha] : \ell_1 ++ (\ell_1 ++ \ell_3) = (\ell_1 ++ \ell_2) ++ \ell_3$$

12.1.5 Fixpunktinduktion

Nicht bei allen Funktionen ist die Rekursion primitiv oder strukturell. Dann kann man Eigenschaften durch *Fixpunktinduktion* über die Gleichungen der Funktionsdefinition zeigen.

Wir nehmen an, die Funktion f sei mit einigen nicht rekursiven Gleichungen und mindestens einer rekursiven Gleichung definiert:

$$f\, x = \ldots f t_1 \ldots f t_n \ldots$$

Dann ist zu zeigen: Für alle Argumente x gilt $P(f\, x)$. Als *Induktionsverankerung* zeigen wir P für alle nichtrekursive Gleichungen von f.

Als *Induktionsannahme* setzen wir $P(f\, t_i)$ für alle t_i, $1 \leqslant i \leqslant n$ voraus, die *kleiner* sind als x. Im *Induktionsschritt* müssen wir dann zeigen, dass $P(f\, x)$ daraus folgt. Das heißt, wir zeigen, dass ein Rekursionsschritt P bewahrt. Das muss für jede rekursive Gleichung von f gemacht werden.

Als Beispiel zeigen wir die Korrektheit folgender Definition von *Quicksort*:

```
qsort       :: [Int] → [Int]
qsort []     = []
qsort [x]    = [x]
qsort (x:xs) = (qsort l) ++ [x] ++ (qsort r)
               where l = [ y | y ← xs, y≤ x ]
                     r = [ y | y ← xs, y > x ]
```

Jeder Sortieralgorithmus permutiert die Elemente einer Liste, so dass sie geordnet sind.

Lemma 12.6. *Für alle Listen* $\ell \in [Int]$ *gilt:*

$$Perm(\ell, qsort\ \ell) \wedge Sorted(qsort\ \ell)$$

Hierbei sind die Prädikate Perm und Sorted so definiert:

$$Perm([x_1, \ldots, x_n], [x_{i_1}, \ldots, x_{i_n}]) := \{i_1, \ldots, i_n\} = \{1, \ldots, n\}$$
$$Sorted([x_1, \ldots, x_n]) := \forall 1 \leqslant i < n \bullet x_i \leqslant x_{i+1}$$

Beweis. Durch Induktion über ℓ.

Als Induktionsverankerung betrachten wir die nicht rekursiven Gleichungen und sehen, dass offensichtlich gilt:

$$Perm([], qsort[]) \wedge Sorted(qsort[]) \text{ und } Perm([x], qsort[x]) \wedge Sorted(qsort[x])$$

Im Induktionsschritt betrachten wir die Prädikate *Perm* und *Sorted* getrennt.
1. (*Perm*) Die *Induktionsannahme* ist

$$Perm(l, qsort\ l) \wedge Perm(r, qsort\ r)$$

Aus den Definitionen von l und r folgt:

$$Perm(xs, l +\!\!+ r)$$

Nach Induktionsannahme gilt:

$$Perm(l +\!\!+ [x] +\!\!+ r, qsort\ l +\!\!+ [x] +\!\!+ \texttt{qsort}\ r)$$

wegen folgender Verträglichkeit von *Perm* mit `++`

$$\forall \ell_1, \ell_2, \ell_1', \ell_2' \in \texttt{[Int]}$$
$$\bullet Perm(\ell_1, \ell_1') \wedge Perm(\ell_2, \ell_2') \Rightarrow Perm(\ell_1 +\!\!+ \ell_2, \ell_1' +\!\!+ \ell_2')$$

gilt dann

$$Perm(xs, qsort\ l +\!\!+ qsort\ r)$$

wegen der Transitivität von *Perm*:

$$\forall \ell_1, \ell_2, \ell_3 \in \texttt{[Int]} \qquad \bullet\ Perm(\ell_1, \ell_2) \wedge Perm(\ell_2, \ell_3) \Rightarrow Perm(\ell_1, \ell_3)$$

2. (*Sorted*) Die Induktionsannahme ist:

$$Sorted(qsort\ l) \wedge Sorted(qsort\ r)$$

Dann gilt:

- $\forall y \in l : y \leqslant x$ (nach Definition von l).
- $\forall y \in qsort\ l : y \leqslant x$ (wegen $Perm(l, qsort\ l)$).
- Analog kann für r gezeigt werden: $\forall z \in qsort\ r : x < z$.
- Also gilt: $\forall y \in qsort\ l, z \in qsort\ r : y \leqslant x < z$
- Dann folgt $Sorted(qsort\ l +\!\!+ [x] +\!\!+ qsort\ r)$ aus der Induktionsannahme.

Fixpunktinduktion lässt sich auf wechselseitig rekursive Funktionen erweitern.

In Tabelle 12.2 sind die verschiedenen Arten von Induktion noch einmal gegenübergestellt.

Induktions-	vollständig (über $\mathbb{N}$)	strukturell über Listen	strukturell über Bäumen
-Anfang	$P(0), P(1)$	$P([])$	$P(Leaf)$
-Annahme	$P(n)$	$P(\ell)$	$P(t_1)$, $P(t_2)$
-Schritt	$P(n) \Rightarrow P(n+1)$	$P(\ell) \Rightarrow P(x:\ell)$	$P(t_1) \wedge P(t_2) \Rightarrow P(Branch\ t_1\, x\, t_2)$

Tabelle 12.2. Induktionsschemata für Beweise

12.2 Typinferenz

Alle modernen funktionalen Sprachen sind *getypt*. Programme in diesen Sprachen müssen so geschrieben sein, dass Funktionen nur auf Argumente angewendet werden, deren Typ mit dem der Funktion *verträglich* ist (engl. *compatible*). Die Typisierung ist *streng*: Das System von Typregeln kann nicht "überlistet" werden wie in manchen imperativen Sprachen (z.B. in C++). Die Typisierung ist *statisch*: (Fast) alle Typregeln werden schon bei der Analyse des Programms überprüft, *bevor* es ausgeführt wird.

Beim Programmieren haben wir praktisch *erfahren*, dass auch Haskell eine streng und statisch getypte Sprache ist. Dadurch ist es zwar nicht immer einfach, ein Funktionen so zu definieren, dass `ghci` sie akzeptiert. Wenn dies aber – endlich – gelungen ist, können wir sicher sein, dass unser Programm keine Typfehler mehr enthält. Das spart viele Tests, die in schwächer getypten Sprachen notwendig wären.

Auch wenn das Typsystem funktionaler Sprachen streng ist, so erlaubt es doch, Typen sehr allgemein und flexibel anzugeben. Typen sind *polymorph*: Sie können *Typvariablen* enthalten, die mit anderen Typen instantiiert werden können – auch mit polymorphen. Wenn Typvariablen mit *beliebigen* Typen instantiiert werden können, sprechen wir von *universeller Polymorphe*. In Haskell können polymorphe Typen noch präziser definiert werden. In einer Typdefinition kann angegeben werden, dass bestimmte Typvariablen nur mit Typen instantiiert werden dürfen, die bestimmte *Eigenschaften* haben. Dies nennt sich *eingeschränkte Polymorphie* (engl. *bounded polymorphism*). Die Eigenschaften betreffen das Vorhandensein von bestimmten Operationen auf den instantiierenden Typen, die mit der Zugehörigkeit zu *Typklassen* spezifiziert wird. Typen können zu einer *Instanz* einer Typklasse gemacht werden, indem die Operationen der Klasse für den Typen definiert werden; damit werden die Operationen also *überladen*.

Trotz dieser mächtigen Typkonzepte müssen die Typen von Funktionen in den allermeisten Fällen nicht angegeben werden, sondern können aus der Definition hergeleitet werden. Das bezeichnet man als *Typinferenz*. Aus methodischen Gründen ist es aber sinnvoll, die Typen von Funktionen *immer* explizit zu spezifizieren.

Wir gehen in drei Schritten vor: Zuerst betrachten wir nur monomorphe Typen, also ohne Typvariablen. Danach erweitern wir dies auf universelle Polymorphie und beschreiben schließlich, wie dies zu gebundener Polymorphie erweitert werden kann.

12.2.1 Monomorphe Typen

Typregeln müssen in einem funktionalen Programm an drei Stellen überprüft werden: In Funktionsdefinitionen (Gleichungen), in den Mustern auf deren linken Seiten, den Wächtern (engl. *guards*) und Ausdrücken auf deren rechten Seiten.

Dabei sind im Einzelnen folgende Regeln zu beachten.

1. Eine Gleichung für eine Funktion hat eine linke Seite, in der die Funktion auf k Muster angewendet wird, und $n \geqslant 0$ bewachte Ausdrücke als rechte Seite. ("Unbewachte" Gleichungen sind äquivalent zu "$otherwise = e_1$".)

$$\begin{array}{lll} f : t_1 & \to \cdots \to & t_k \to t \\ f : m_1 & \cdots & m_k \mid g_1 = e_1 \\ & & \vdots \\ & & \mid g_n = e_n \end{array}$$

   Dafür gelten die folgenden Typregeln:
   a) Alle Wächter g_i müssen den Typ `Bool` haben.
   b) Die Muster m_i müssen mit den Typen t_i verträglich sein.
   c) Die Ausdrücke e_i müssen den Typ t haben.
2. Muster sind Variablen aus der Menge $\{\_, x_1, \ldots\}$ oder Anwendungen $k\, m_1 \cdots m_n$ eines Konstruktors k auf $n \geqslant 0$ Muster.
   Ein Muster m ist verträglich mit einem Typ t, wenn es auf einige Elemente seiner Wertemenge passt.
   a) Eine Variable x_i und die anonyme Variable vertragen sich mit jedem Typ t. Eine Variable x_i gilt dann in der Gleichung als mit dem Typ t vereinbart. (Variablen werden immer implizit deklariert.)
   b) Ein Muster $k\, m_1 \cdots m_n$ verträgt sich mit t, wenn der Wertkonstruktor k den Typ $t_1 \to \cdots \to t_n \to s$ hat, alle Muster m_i verträglich mit t_i sind und s mit t verträglich ist. (Die Menge K der Wertkonstruktoren enthält die Listenkonstruktoren "[]" und "(:)" und die Tupelkonstruktoren "$(\,,\ldots,\,)$".)
3. Ausdrücke sind *Literale* für Zahlen (N), Zeichen (C) oder Zeichenketten (S), Namen von Konstanten (wobei Funktionen als Konstanten von funktionalen Typen $\sigma \to \tau$ aufgefasst werden) und Variablen, oder Anwendungen von Funktions-Ausdrücken auf Argument-Ausdrücke:

$$\begin{array}{lll} E ::= & N \mid C \mid S \\ \mid & c_1 \mid \cdots \mid c_k \\ \mid & x_1 \mid \cdots \\ \mid & E\, E \end{array}$$

In Ausdrücken gelten folgende Typregeln:

a) Literale N für Zahlen, C für Zeichen und S für Zeichenketten haben ihren immanenten Typ.
b) Konstanten c_i haben den Typ, der sich aus ihrer Definition ergibt.
c) Eine Funktionsanwendung $f\,a$ hat den Typ t, wenn der Funktionsausdruck f einen Funktionstyp $s \to t$ hat; der Typ s' des Argumentausdrucks a muss mit dem Argumenttyp s von f verträglich sein.

(In lokalen Definitionen "e **where** $d_1 \ldots d_m$" müssen für e die Regeln für Ausdrücke und für die d_i diejenigen für Definitionen gelten.)

Typüberprüfung wird oft mit *Inferenzregeln* beschrieben. Man geht aus von der Annahme, dass alle Typspezifikationen des Programms und der Bibliothek in einer Menge Δ von Definitionen der Form (f, t) enthalten sind. Mithilfe von Δ können dann die Typen in Ausdrücken mit zwei Regeln bestimmt werden:

- Jeder Name x hat den Typ, der in Δ eingetragen ist.
- Bei jeder Funktionsanwendung $f\,a$ muss der Ausdruck f einen Funktionstyp haben, dessen Argumenttyp mit dem von a verträglich ist; der Typ der Anwendung ist dann der Resultattyp von f.

In Inferenzregeln wird dies so formuliert.

$$\frac{(x,t) \in \Delta}{\Delta \vdash x : t} \qquad \frac{(c,t) \in \Delta}{\Delta \vdash c : t} \qquad \frac{\Delta \vdash f : s \to t, \Delta \vdash a : s}{\Delta \vdash f\,a : t}$$

Für die Funktionsgleichungen muss man dann nur noch sicherstellen, dass alle Wächter den Typ `Bool` haben und linke und rechte Seite der Gleichung den gleichen Typ haben.

Beispiel 12.1 (Typisierung von `sqSum`). Die Funktion `sqSum` berechnet das Quadrat der Summe seiner Argumente.

```
sqSum :: Double → Double → Double
sqSum a b = (^) ((+) a b) 2
```

(Wir schreiben die Operationen als normale Funktionsanwendungen.) Die Menge Δ soll (mindestens) folgende Definitionen enthalten:[1]

```
sqSum :: Double →Double → Double
(+)   :: Double →Double → Double
(^)   :: Double →Int → Double
2     :: Int
```

Mit den Inferenzregeln können die Typen der Teilausdrücke in der Definition bestimmt werden:

- `sqSum :: Double ->Double -> Double`
  ⇒ `a :: Double` und `sqSum a :: Double -> Double`.

[1] Die Typen von (+) und (^) sind in HASKELL allgemeiner.

- `sqSum a :: Double -> Double`
  ⇒ `b :: Double` und `sqSum a b :: Double`.
- `(+) :: Double -> Double -> Double`
  ⇒ `a :: Double` und `(+) a :: Double -> Double`.
- `(+) a :: Double -> Double`
  ⇒ `b :: Double` und `(+) a b :: Double`.
- `(^) :: Double -> Int -> Double`
  ⇒ `(+) a b :: Double` und `(^) ((+) a b) :: Int -> Double`.
- `(^) ((+) a b) :: Int -> Double`
  ⇒ `2 :: Int` und `(^) ((+) a b) 2 :: Double`.

Die Funktionsdefinition entspricht also den Typregeln, weil linke und rechte Seite beide den Typ `Double` haben.

Weil in diesem Beispiel angenommen wurde, dass alle Typen *monomorph* sind, ist die Überprüfung einfach zu erledigen, indem die Funktionsanwendungen *bottum-up* untersucht werden. Dabei muss für alle Teilausdrücke die Gleichheit von Typen überprüft werden.

12.2.2 Universell polymorphe Typen

Die Typen in einer funktionalen Sprache können *polymorph* sein, und also die Typen von Ausdrücken auch. Wir beschränken uns zunächst auf universelle Polymorphie. Dann kann die Verträglichkeit von zwei Typen t und s nicht mehr einfach durch Feststellen der Gleichheit bestimmt werden. Beide Typen können ja Typvariablen enthalten; die Typen sind verträglich, wenn diese Variablen durch Typen instantiiert werden können, so dass t und s gleich werden.

Eine derartige Überprüfung kann durch *Unifikation* der Typausdrücke erreicht werden. Allgemeine Unifikation ist eine Grundlage des logischen Programmieren. Hier betrachten wir nur die Unifikation von Typausdrücken.

Bei der Unifikation von zwei Typasdrücken t und s geht man zunächst davon aus, dass die Typvariablen in beiden Typausdrücken verschieden sind. Die Unifikation versucht, einen *Unifikator* σ zu finden, der allen Variablen Ausdrücke zuordnet so dass $t\sigma = s\sigma$. Der Unifikator wird so bestimmt (rekursiv über den Aufbau der Typen):

- Einer der Ausdrücke ist eine Typvariable x, sagen wir $t = x$. Dann definiert $x \mapsto s$ einen Unifikator für t und s.
- Andernfalls bestehen beide Typen aus einem Typkonstruktor, möglicherweise mit Typausdrücken als Parametern, sagen wir $t = k\ t_1 \ldots t_n$ und $s = k'\ s_1 \ldots s_n$. Dann muss gelten:
  - Die Typkonstruktoren sind gleich: $k = k'$.
  - Die entsprechenden Typparameter haben Unifikatoren σ_i so dass $t_i\sigma = s_i\sigma$.

  - Die Unifikatoren $\sigma_1, \ldots, \sigma_n$ lassen sich zu einem konsistenten Unifikator σ vereinigen.
- Andernfalls sind die Typausdrücke nicht unifizierbar.

Man kann zeigen, dass der so berechnete Unifikator – wenn er existiert – der *allgemeinste* ist. Er schränkt also die Polymorphie nur ein, wenn sonst ein Typfehler auftreten würde. Für die polymorphe Typinferenz braucht man die Typspezifikationen für die Funktionen nicht; es reicht, davon auszugehen, dass alle Namen von Konstanten, Funktionen und Variablen zunächst verschiedene allgemeine Typen haben, die später durch die Unifikation instantiiert werden können.

Beispiel 12.2. Wir betrachten die Längenfunktion mit folgenden Gleichungen:

```
length []     = 0
length (_:l) = (+) 1 (length l)
```

Die Definitionen sind zunächst:

$$\Delta = \left\{ \begin{array}{l} (\texttt{length}, \alpha), ([], [\beta]), ((:), \gamma \to [\gamma] \to [\gamma]), \\ (\texttt{0}, \texttt{Int}), (\texttt{1}, \texttt{Int}), ((+), \texttt{Int} \to \texttt{Int} \to \texttt{Int}), \end{array} \right\}$$

Wir bestimmen die Unifikatoren für beide Gleichungen:

- $[] : [\alpha] \Rightarrow \mathit{length}\ [\alpha] \to \delta$.
- $0 : \texttt{Int} \Rightarrow \delta = \texttt{Int}$ und damit $\mathit{length}\ [\alpha] \to \texttt{Int}$.
- $(:) : \gamma \to [\gamma] \to [\gamma] \Rightarrow l : [\gamma]$.
- $(+) : \texttt{Int} \to \texttt{Int} \to \texttt{Int} \wedge 1 : \texttt{Int} \Rightarrow (+)\ 1 : \texttt{Int} \to \texttt{Int}$.
- $(+)\ 1 : \texttt{Int} \to \texttt{Int} \Rightarrow \mathit{length}\ l : \texttt{Int}$.
- $\mathit{length}\ l : \texttt{Int} \Rightarrow \mathit{length} : [\gamma] \to \texttt{Int}$.

Der Typ von beiden Gleichungen ist also bis auf die Namen der Typvariablen, die irrelevant sind, gleich.

Hier noch drei weitere Beispiele:

1. Betrachten wir die Definition

   $$f(x, y) = (x, ['a'..y])$$

   Aus der linken Seite lassen sich die Typen $f : (\alpha, \beta) \to \gamma$ sowie $x : \alpha$ und $y : \beta$ herleiten. Die rechte Seite ist ein Tupel, dessen zweite Komponente eine Aufzählung von Zeichen ist. Deshalb ist der Unifikator σ definiert als $\beta \mapsto \texttt{Char}$ und $\gamma \mapsto (\alpha, [\texttt{Char}])$, und f hat den Typ $(\alpha, \texttt{Char}) \to (\alpha, [\texttt{Char}])$.
2. Nun untersuchen wir die Funktionsdefinition

   $$g(m, zs) = m + \mathit{length}\ zs$$

   Aus der linken Seite lassen sich wieder die Typen $g : (\alpha, \beta) \to \gamma$ sowie $m : \alpha$ und $zs : \beta$ herleiten. Auf der rechten Seite wird *length* auf zs angewendet; deshalb ist der Unifikator σ definiert als $\beta \mapsto [\delta]$ sowie $\gamma \mapsto \texttt{Int}$ und $\alpha \mapsto \texttt{Int}$. Damit hat g den Typ $(\texttt{Int}, [\delta]) \to \texttt{Int}$.

3. Nun betrachten wir die Komposition der beiden Funktionen:

   $h = g \circ f$

   wobei die Funktionskomposition (.) den Typ $(y \to z) \to (x \to y) \to x \to z$ hat. Dann muss der Resultattyp $(a, [\texttt{Char}])$ von f und der Argumenttyp $(\texttt{Int}, [b])$ von g zu $(\texttt{Int}, [\texttt{Char}])$ unifiziert wird. Die Funktion h hat also den Typ $(\texttt{Int}, [\texttt{Char}]) \to \texttt{Int}$.

12.2.3 Gebunden polymorphe Typen

In HASKELL gibt es nicht nur universell polymorphe Funktionen, sondern auch solche, deren Typparameter einen *Kontext* voraussetzen: die Zugehörigkeit zu einer Typklasse, und damit das Vorhandensein von bestimmten Operationen auf diesem Typ.

Wir wollen informell anhand eines Beispiels erläutern, wie bei der Typinferenz mit Kontexten umgegangen wird. Betrachten wir das `contains`-Prädikat:

```
contains []     _ = False
contains (x:xs) y = (x == y) || contains xs y
```

Der Typ von `contains` ist

$$Eq\,\alpha \Rightarrow [\alpha] \to \alpha \to \texttt{Bool}$$

weil `x` und `y` verglichen werden, sodass der Typ α vergleichbar sein muss. Nehmen wir an, wir wendeten die Funktion `contains` auf einen Ausdruck e vom Typ $Ord\,\beta \Rightarrow [[\beta]]$ an; e ist also eine Liste von Listen von Werten, die geordnet sein müssen. Ohne Berücksichtigung des Kontextes müsste `contains` also mit dem Typ

$$[[\beta]] \to [\beta] \to \texttt{Bool}$$

instantiiert werden, und die Funktionsanwendung `contains` e hätte also den Typ $[\beta] \to \texttt{Bool}$. Jetzt müssen wir noch die Kontexte unifizieren. Dies ergibt $(Eq\,[\beta], Ord\,\beta)$. Dieser Kontext muss nun überprüft und vereinfacht werden.

Kontexte dürfen sich nur auf Typvariable beziehen, weshalb wir die Anforderung $Eq\ [\beta]$ eliminieren müssen. Das kann nur durch Instanzen-Vereinbarungen geschehen. Da Listen mit "**deriving** $(Eq, \ldots)$" vordefiniert sind, gilt

instance $Eq\,\alpha \Rightarrow Eq\,[\alpha]$ **where** ...

können wir die Anforderung $Eq\,[\beta]$ vereinfachen zu $Eq\,\beta$. Der einfachere Kontext für `contains` ist also $(Eq\,\beta, Ord\,\beta)$. Das Eliminieren muss fortgesetzt werden, bis keine Instanz-Vereinbarungen mehr anwendbar sind. Wenn dann nicht alle Anforderungen auf Typvariable zurückgeführt werden konnten, gibt es eine Fehlermeldung. Wenn wir zum Beispiel geschrieben hätten "`contains [id]`", käme folgende Fehlermeldung, weil `id` eine Funktion ist und damit nicht Instanz von Eq:

```
ERROR: a → a is not an instance of the class "Eq"
```

In einem zweiten Schritt vereinfachen wir den Kontext weiter mithilfe der Klassen-Vereinbarungen. Die Klasse *Ord* ist so definiert:

class $Eq\,\alpha \Rightarrow Ord\,\alpha$ **where** ...

Also sind Instanzen von *Ord* automatisch auch welche von *Eq*, und es bleibt nur noch der Kontext

$Ord\,\beta$

übrig. Auch dieser Vorgang wird so lange wiederholt, bis keine Vereinfachungen mehr möglich sind. Die Funktion hat also den Typ:

$contains\,E : Ord\,\beta \Rightarrow [[\beta]] \to [\beta] \to \texttt{Bool}$

Aufgabe

Nehmen Sie an, die Funktion `sqSum` aus Beispiel 12.1 sei ohne Typspezifikation definiert worden. Leiten Sie ihren allgemeinsten Typ ab.

13

Aufwand

In diesem Kapitel wiederholen wir kurz einige Grundbegriffe der Komplexität von Programmen und gehen darauf ein, wie wir die Zeit- Und Speicherkomplexität von Funktionen bewerten. Wir wenden diese Maße auf einige Funktionen an und beschreiben einige Verfahren, mit denen Funktionen effizienter gemacht werden können. Ein wichtiges Verfahren ist die Überführung einer Funktionsdefinition in *endrekursive Form*. Außerdem gehen wir auf *Speicherlecks* und einige weniger effiziente Konstruktionen funktionaler Sprachen ein, wie überladene Funktionen und Listen.

13.1 Komplexität funktionaler Programme

13.1.1 Komplexitätsgrade

Die Grundlagen der Komplexitätsmaße – nämlich die $\mathcal{O}$- und Θ-Notation – sind sicherlich aus *Praktische Informatik 1 und 2* bekannt.

Wir sagen, eine (Komplexitäts-) Funktion f sei *vom Grad* g wenn es positive ganze Zahlen m und d gibt, so dass für alle $m \geqslant n$ gilt:

$$f\ n \leqslant d \cdot (g\ n)$$

Wir unterscheiden folgende Grade g

$$n^0 \ll \log n \ll n^1 \ll n \log n \ll n^2 \cdots \ll n^k \ll \cdots \ll 2^n \ll \cdots$$

Dann sagen wir, die Funktion f sei (höchstens) vom Grade $\mathcal{O}(g)$; ist die Funktion von keinem Grad $g' \ll g$, sagen wir sie sei *genau* vom Grad $\Theta(g)$.

13.1.2 Zählen

Bei vielen Aufwandsabschätzungen werden wir zählen. Wir betrachtren einige allgemeine Beispiele, die uns im Weiteren wieder begegnen werden:

1. *Wie oft müssen wir eine Liste in gleich lange Stücke teilen, bis die Stücke die Länge 1 haben?* Wenn die Liste n Elemente enthält, enthalten ihre Stücke nach der ersten Teilung $\frac{n}{2}$, nach der zweiten $\frac{n}{4}$ und nach der p-ten Teilung $\frac{n}{n^p}$ Elemente. Die Länge ist kleiner oder gleich 1, wenn

   $$2^p \geqslant n > 2^{p-1}$$

   Wenn wir den Logarithmus bilden, erhalten wir

   $$p \geqslant \log_2 n > p - 1$$

   Also ist der Aufwand für das Aufteilen einer Liste der Länge n gleich $\Theta(\log_2 n)$.
2. *Wieviele Knoten enthält ein balancierter Baum der Tiefe n?* Auf Tiefe 1 hat so ein Baum einen Knoten, auf Tiefe 2 hat er zwei Knoten, und auf Tiefe k hat er 2^{k-1} Knoten. In der Summe über alle b Tiefen ergibt das

   $$1 + 2 + 4 + \cdots + 2^{b-1} = 2^b - 1$$

   Die Größe eines balancierten Baumes ist also $\Theta(2^b)$ in Abhängigkeit seiner Tiefe b; umgekehrt hat ein balancierte Baum mit n Knoten also die Tiefe $\log_2 n$. Bei nicht balancierten Bäumen kann die Tiefe ein Extremfall linear von der Anzahl der Knoten abhängen.
3. Die Summe aller Zahlen von 1 bis n ist

   $$\sum_{i=1}^{n} i = 1 + 2 + \cdots + n = \frac{n}{2}(n+1)$$

   und hat damit die Größenordnung $\Theta(n^2)$, wächst also *quadratisch*.

13.1.3 Berechnungsaufwand für Funktionen

Wie können wir den Berechnungsaufwand einer Funktion *messen*? Wir können beispielsweise entsprechende Diagnosen unseres HASKELL-Systems benutzen. Gibt man im `ghci`-Interpreter den Befehl "`:set +s`" ein, so gibt er nach jeder Auswertung die Zeile aus:

(⟨x⟩ `seconds,` ⟨m⟩ `bytes)`

Diese Zahlen sagen leider nicht besonders viel, weil in diese Zeiten auch viele Aktionen der `ghci`-Laufzeitumgebung einfließen. Sonst ließe sich folgender Dialog schwer erklären:

```
Hugs.Base> 1
1
(0.04 secs, 2273592 bytes)
```

Leider gibt es keine direkte Möglichkeit, sich genau die Anzahl der Reduktionen einer Funktion anzeigen zu lasssen; das geht auch nicht für die von dieser Funktion angelegten Knoten. Allerdings kann man im *Debugger* Haltepunkte setzen und auch eine Geschichte der Programmausführung ausgeben lassen.

Wenn wir den Aufwand einer Funktion "symbolisch" abschätzen – also auf Papier durch Inspektion der Funktionsdefinitionen, bieten sich drei Maße an:

1. Den *Zeitaufwand* messen wir, indem wir die Funktionsanwendungen zählen, die für die Berechnung eines Ergebnisses gebraucht werden.
2. Als *Speicheraufwand* ermitteln wir vorwiegend den maximal benötigten Speicherplatz auf dem *Stapel* (*stack*). Den ermitteln wir, indem wir die maximale Größe des ausgewerteten Ausdrucks zählen.
3. Ein anderer Speicheraufwand ist der *totale Speicheraufwand*, bei dem man den in allen Schritten gebrauchten Speicherplatz aufsummiert. Dies ist die Anzahl der Zellen (Knoten), die `ghci` ausgibt und misst den Speicherverbrauch auf der *Halde* (*heap*).

Der Zeitaufwand wird immer als eine Funktion ermittelt, die von der Größe ihrer Eingabe abhängt. Oft ist es dabei interessant, den schlimmsten, den durchschnittlichen und den schlechtesten Fall zu unterscheiden. Einige Beispiele sollen dies erläutern:

1. Die Auswertung der *Fakultätsfunktion* mit der Definition

   ```
   fac 1 = 1
   fac n = n * fac (n-1)
   ```

   geschieht wie folgt:

   $$\begin{aligned} fac\, n &\rightsquigarrow n * fac(n-1) \\ &\rightsquigarrow \cdots \\ &\rightsquigarrow n * ((n-1) * \cdots * (2 * (1 * 1)) \cdots) \\ &\rightsquigarrow n * ((n-1) * \cdots * (2 * 1) \cdots) \\ &\rightsquigarrow n * ((n-1) * \cdots * 2 \cdots) \\ &\rightsquigarrow \cdots \\ &\rightsquigarrow n! \end{aligned}$$

   Die Berechnung benötigt $n+1$ Aufrufe von *fac* und n Multiplikationen, und ihr größter Ausdruck ist eine Multiplikation mit $n+1$ Operanden. Also ist sie linear ($\Theta(n^1)$).
2. *Einfügendes Sortieren* ist folgendermaßen definiert:

   ```
   iSort []     = []
   iSort (x:xs) = ins x (iSort xs)

   ins x []                 = [x]
   ins x (y:ys) | x≤ y      = x:y:ys
                | otherwise = y:ins x ys
   ```

Eine allgemeine Auswertung ist

$$\begin{aligned} iSort[a_1, a_2, \ldots, a_{n-1}, a_n] &\rightsquigarrow ins\, a_1 (iSort[a_2, \ldots, a_{n-1}, a_n]) \\ &\rightsquigarrow ins\, a_1 (ins\, a_2 (\cdots ((ins\, a_{n-1}\, ins\, a_n [])) \cdots)) \end{aligned}$$

Dann wird n mal *ins* ausgewertet. Eine allgemeine Auswertung hat die Form

$$ins\, a[a_1, a_2, \ldots, a_{n-1}, a_n]$$

Für den Aufwand kann man drei Fälle unterscheiden:
a) Im *besten* Fall, wenn $a \leqslant a_1$ ist, braucht man einen Schritt.
b) Im *schlechtesten* Fall, wenn $a \geqslant a_n$ ist, braucht man n Schritte.
c) Im Mittel braucht man $\frac{n}{2}$ Schritte.

Für das Sortieren als Ganzes heißt das:
a) Im *besten* Fall, braucht jedes *ins* einen Schritt, und *iSort* ist $\mathcal{O}(n^1)$.
b) Im *schlechtesten* Fall, werden die Einfüge-Operationen $1+2+\cdots+n-1$ Schritte brauchen, so dass das Sortieren $\mathcal{O}(n^2)$ Schritte braucht.
c) Im Mittel brauchen die Einfüge-Operationen $\frac{1}{1}+\frac{2}{2}+\cdots+\frac{n-1}{2}$ Schritte, was insgesamt auch $\mathcal{O}(n^2)$ Schritte ergibt.

Also hat *iSort* meistens quadratischen Aufwand, aber linearen Aufwand im Fall, wo die Liste schon annähernd sortiert ist.

13.2 Endrekursion

Bestimmte Formen der Rekursion brauchen Speicherplatz, weil erst bei Abbruch der Rekursion der in der Rekursion aufgebaute Ausdruck ausgewertet werden kann. Wir hatten das oben bei der Fakultätsfunktion gesehen. In ihrer Definition taucht der rekursive Ausdruck *geschachtelt* auf, in "$n * fac(n-1)$".

Dagegen nennen wir eine Funktion *endrekursiv*, wenn keiner ihrer rekursiven Aufrufe in einem geschachtelten Ausdruck steht. D.h., "über" dem rekursiven Aufruf stehen nur Fallunterscheidungen wie `if`, `case` oder Wächter (*guards*, "|"). So eine Rekursion entspricht Schleifen in imperativen Programmen. Sie kann mit Schleifen übersetzt werden.

Oft kann eine Funktion in endrekursive Form gebracht werden, so auch die Fakultät:

```
fac' :: Int→ Int
fac' n = fac0 n 1
         where  fac0 n a = if n == 0 then a
                           else fac0 (n-1) (n*a)
```

Die endrekursive Funktion *fac0* akkumuliert das Ergebnis in einem zweiten Argument `a`. Betrachten wir die Auswertung:

$$
\begin{aligned}
fac\, n &\rightsquigarrow fac0\,(n-1)1 \\
&\rightsquigarrow fac0\,(n-1)(n*1) \\
&\rightsquigarrow fac0\,(n-2)((n-1)*n*1) \\
&\rightsquigarrow \cdots \\
&\rightsquigarrow fac0\ 0(1*2*\cdots*(n-1)*n*1) \\
&\rightsquigarrow 1*2*\cdots*(n-1)*n*1 \\
&\rightsquigarrow 2*\cdots*(n-1)*n*1 \\
&\rightsquigarrow \cdots \\
&\rightsquigarrow n!
\end{aligned}
$$

Auch hier ist der maximale Ausdruck von der Größe $\mathcal{O}(n^1)$; dies liegt an der verzögerten Auswertung, denn erst nach der Anwendung von "*fac0* $0\cdots$" ist klar, dass der Wert von `a` gebraucht wird und ausgewertet werden muss. Um den Nutzen aus der endrekursiven Fassung ziehen, muss man das zweite Argument *strikt* auswerten lassen. Das geht mit der strikten Funktionsanwendung "`($!)`":[1]

```
(\$!) :: (a→ b)→ a→ b
f $! x = x `seq` f x
```

Diese Fassung der Fakultät lautet:

```
fac' :: Int→ Int
fac' n = fac0 n 1
         where  fac0 n a = if n == 0 then a
                           else fac0 (n-1) $!(n*a)
```

Betrachten der Auswertung zeigt, dass fac' einen konstanten Speicherbedarf hat.

$$
\begin{aligned}
fac\, n &\rightsquigarrow fac0\,(n-1)1 \\
&\rightsquigarrow fac0\,(n-1)(n) \\
&\rightsquigarrow fac0\,(n-2)(n^2-n) \\
&\rightsquigarrow \cdots \\
&\rightsquigarrow fac0\ 0(n!) \\
&\rightsquigarrow n!
\end{aligned}
$$

Die Funktion rev' zum Umdrehen einer Liste ist *nicht* endrekursiv:

```
rev' :: [a]→ [a]
rev' []     = []
rev' (x:xs) = rev' xs ++ [x]
```

[1] Die eingebaute Funktion `seq :: a-> b-> b` wertet zunächst das erste Argument aus.

In der Rekursion wird auch noch hinten an eine Liste angehängt. Deshalb ist der Speicheraufwand linear, und der Zeitaufwand sogar quadratisch.

Die endrekursive Fassung akkumuliert die Ergebnisliste in einem zweiten Argument; sie hat konstanten Speicheraufwand und auch nur linearen Zeitaufwand.

```
rev :: [a]→ [a]
rev xs = rev0 xs [] where
    rev0 []     ys = ys
    rev0 (x:xs) ys = rev0 xs (x:ys)
```

13.2.1 Überführung in Endrekursion

Eine Funktion $f' \colon S \to T$ kann immer in Endrekursion überführt werden, wenn ihre Definition folgende Form hat:

$$f'\, x = \textbf{if } B\, x \textbf{ then } H\, x \textbf{ else } \varphi(f'(K x))(E\, x)$$

Dabei ist $K \colon S \to S$ die Funktion, die das Argument von f in der Rekursion "verkleinert", $\varphi \colon T \to T \to T$ eine binäre Operation, in der der rekursive Aufruf von f geschachtelt ist, und $E\, x$ ein weiterer Bestandteil des geschachtelten Ausdrucks. Wenn φ assoziativ ist und $e : T$ als neutrales Element hat, sieht die die endrekursive Form $f \colon S \to T$ so aus:

$$f\, x = g\, x\, e \textbf{ where } g\, x\, y = \textbf{if } B\, x \textbf{ then } \varphi(H\, x)y \textbf{ else } g(K\, x)(\varphi(E\, x)y)$$

Wir wollen das an der Längenfunktion von Listen ausprobieren:

```
length' :: [a]→ Int
length' xs = if (null xs) then 0
             else 1+ length' (tail xs)
```

Zuordnung der Variablen:

$$\begin{array}{rlrl} K(x) & \mapsto \mathit{tail} & B(x) & \mapsto \texttt{null}\ x \\ E(x) & \mapsto 1 & H(x) & \mapsto 0 \\ \varphi(x,y) & \mapsto x + y & e & \mapsto 0 \end{array}$$

Es gilt: $\varphi(x, e) = x + 0 = x$ (0 ist neutrales Element.) Damit ergibt sich endrekursive Variante:

```
length :: [a]→ Int
length xs = len xs 0 where
  len xs y = if (null xs) then 0 -- was: 0+ 0
             else len (tail xs) (1+ y)
```

Im **then**-Teil nutzen wir aus, dass $\varphi(H\, x)y \equiv 0 + 0 = 0$ ist.

13.2.2 Endrekursion bei Aktionen

Auch bei Aktionen kann man diese Art der Rekursion entdecken bzw. sie herstellen.

Eine Aktion ist endrekursiv, wenn nach dem rekursiven Aufruf keine weiteren Aktionen folgen. Die Aktion `getLines'` ist nicht endrekursiv:

```
getLines' :: IO String
getLines' = do str← getLine
               if null str then return ""
                  else  do rest← getLines'
                           return (str++ rest)
```

Wieder wird die Struktur umgebaut, so dass das Ergebnis in einem Argument akkumuliert wird:

```
getLines :: IO String
getLines = getit "" where
   getit res = do str← getLine
                  if null str then return res
                     else getit (res++ str)
```

13.2.3 Fortgeschrittene Endrekursion

Die Ergebnisse einer Funktion können auch einen "Abschluss" (engl. *closure*) akkumuliert werden. Abschlüsse sind partiell instantiierte Funktionen.

Als Beispiel betrachten wir die Klasse `Show`:

```
class Show a where
   show :: a→ String
```

Nur `show` hätte quadratischen Aufwand ($O(n^2)$), weil Instanzen von `show` für zusammengesetzte Typen mit Komponenten $v_1, \ldots, v_n$ typischerweise als Verkettungen

$$show\ a_1 \mathbin{+\!\!+} \cdots \mathbin{+\!\!+} show\ a_n$$

definiert werden. Deshalb gibt es zusätzlich die Funktion `showsPrec`[2] , die die Zeichenkette erst aufbaut, wenn sie ausgewertet wird. Das hat linearen Aufwand ($O(n)$).

```
   showsPrec :: Int→ a→ String→ String
   show x    = showsPrec 0 x ""
```

Damit können wir zum Beispiel eine Instanz von `Show` für Mengen definieren, die als Listen dargestellt werden:

```
data Set a = Set [a] -- Mengen als Listen
```

[2] Der erste Parameter (vom Typ `Int`) sorgt für das richtige Setzen von Klammern.

```
instance Show a⇒ Show (Set a) where
  showsPrec i (Set elems) =
      \r→ r++ "{" ++ concat (intersperse ",␣"
                               (map show elems)) ++ "}"
```

Besser noch ist folgende Implementierung:

```
instance Show a⇒ Show (Set a) where
  showsPrec i (Set elems) = showElems elems where
    showElems []     = ("{}" ++)
    showElems (x:xs) = ('{' :) ∘ shows x ∘ showl xs
       where showl []     = ('}' :)
             showl (x:xs) = (',' :) ∘ shows x ∘ showl xs
```

13.3 Striktheit

Wir haben schon beim Beispiel der Fakultätsfunktion gesehen, dass strikte Auswertung den Platzaufwand für endrekursive Funktionen verringern kann. Dies gilt auch für Kombinatoren. Ein interessantes Beispiel dazu sind *Faltungen* (von Listen). Wir haben bisher meistens die *Rechtsfaltung* `foldr` benutzt:

```
foldr            :: (a → b → b) → b → [a] → b
foldr op z []     =  z
foldr op z (x:xs) =  op x (foldr op z xs)
```

Jedoch ist `foldr` ist nicht endrekursiv. Die Variante `foldl` faltet eine Liste von links und ist endrekursiv:

```
foldl :: (a → b → a) → a → [b] → a
foldl op z []      = z
foldl op z (x:xs)  = foldl op (op z x) xs

foldl' :: (a → b → a) → a → [b] → a
foldl' op z []      = z
foldl' op z (x:xs)  = foldl' op ((op $! z) $!x) xs
```

Wenn die Operation `op` assoziativ ist, kann man leicht die eine durch die andere Faltung ersetzen. Die Variante `foldl` ist nicht nur endrekursiv, sondern wertet die Operation `op` auch strikt aus.

Welche von den folgenden Versionen von `faculty` ist die effizienteste?

```
faculty, faculty', faculty'' :: Integer → Integer
faculty   n = foldr  (∗) 1 [1..n]
faculty'  n = foldl  (∗) 1 [1..n]
faculty'' n = foldl' (∗) 1 [1..n]
```

Alle drei Fassungen haben linearen Zeitaufwand, aber nur die letzte hat konstanten Speicheraufwand (ggü. linearem Speicheraufwand der anderen beiden Versionen). Das liegt daran, dass die zu faltende Operation ohnehin strikt ist.

Welche Fassung der vordefinierten Funktion `and` wäre die günstigste?

```
and, and', and'' :: [bool]→ Bool
and   = foldr  (&&) True
and'  = foldl  (&&) True
and'' = foldl' (&&) True
```

Die Konjunktion `(&&)` ist nur im ersten Argument strikt, so dass hier die erste Fassung am günstigsten ist, denn so muss nicht die ganze Liste durchlaufen werden, wie dies bei `foldl` der Fall wäre.

Dies sind die Vor- und Nachteile der verschiedenen Faltungen:

- `foldr` ist *nicht* endrekursiv, traversiert aber nicht immer die *ganze* Liste.
- `foldl` endrekursiv, traversiert aber immer die *ganze* Liste.
- `foldl'` hat konstanten Platzaufwand, traversiert aber immer die *ganze* Liste.

Wann sollte also welches `fold` genommen werden? Die Faustregel lautet so:

- Strikte Funktionen auf "endlichen" Listen sollten mit `foldl'` gefaltet werden.
- Wenn nicht immer die ganze Liste benötigt wird, sollte `foldr` benutzt werden, also wenn die binäre Operation "$\oplus$" nicht strikt in ihrem zweiten Argument ist.

13.4 Gemeinsame Teilausdrücke

In den allermeisten Implementierungen der allermeisten funktionalen Sprachen werden Ausdrücke nicht als Bäume (*Terme*), sondern als *Termgraphen* dargestellt. Damit soll vermieden werden, dass gleiche Teilausdrücke mehrmals dargestellt werden.

Betrachten wir die Funktion `square`:

```
square :: Int → Int
square x = x + x
```

Bei der Auswertung von *square* E führt dies dazu, dass ihr Argument E (bzw. der Termgraph, der E darstellt) bei der Anwendung der Definition nicht *kopiert* wird, sondern für die Argumente von "+" werden zwei Verweise auf ein und denselben Termgraphen eingefügt. Man sagt auf gut Deutsch, die beiden Auftreten von E würden "geshart" (von engl. *to share*, "sich in etwas teilen"). Wegen der referentiellen Transparenz ist dies semantisch unbedenklich. Das Argument E wird nie mehr als einmal ausgewertet.

Auch etwas subtilere Formen von gleichen Teilausdrücken werden von guten Implementierungen erkannt:

```
f :: Int→ Int
f x = (x+1)* (x+1)
```

Die Glasgower Implementierung (`ghc` bzw. `ghci`) wird den gemeinsamen Teilausdruck $x + 1$ erkennen; bei `ghci` ist das nicht so sicher.

Man kann *Sharing* von Teilausdrücken auch explizit erreichen, indem man mit **where** oder **let** Teilausdrücke benennt und ihre Namen an verschiedenen Stellen benutzt:

```
f' :: Int→ Int
f' x = y * y where y = x+1
```

Auch ein Teilmuster auf der linken Seite einer Funktionsdefinition kann auf ihrer rechten Seite wiederbenutzt werden, wenn man es explizit benennt:

```
isSorted :: Ord a ⇒ [a] → Bool
isSorted []  = True
isSorted [_] = True
isSorted (x:xs@(y:ys)) | x≤y      = True
                       | otherwise = isSorted xs
```

Das Muster "n@p" vergleicht ("*matcht*") das Muster p und bindet den Wert an den Namen n, damit er auf der rechten Seite der Gleichung benutzt werden kann. Schriebe man auf der rechten Seite statt `xs` den Ausdruck `y:ys`, hinge es von der Güte der Implementierung ab, ob dieser als gemeinsamer Teilausdruck erkannt würde. Im optimistischen Fall würde kein Speicherplatz auf der Halde benötigt, sonst linear viel.

13.5 Listen und Felder

Felder (engl. *array*) sind der wichtigste zusammengesetzte Datentyp imperativer Sprachen. Abstrakt gesehen sind sie *Abbildung* von einer Indexmenge in Elementwerte. Ihre Länge ist durch die Größe der Indexmenge festgelegt. In imperativen Sprachen werden Felder als zusammenhängende Speicherzellen angelegt. Deshalb hat sowohl der lesende Zugriff auf das i-te Element eines Feldes als auch das Ersetzen dieses Elementes einen konstanten Zeitaufwand. Bei Feldern wird das selektive Überschreiben ausgenutzt – dies ist einer der Gründe, weshalb sie so effizient sind.

Listen sind der wichtigste zusammengesetzte Datentyp funktionaler Sprachen. Abstrakt gesehen sind sie beliebig lange *Sequenzen* oder Wörter über Elementwerten. Ihre Länge ist beliebig und beliebig veränderbar. Listen werden als verkettete Knoten aus einem Kopf und einem Schwanz dargestellt. Deshalb hat der Zugriff auf das i-te Element einer Liste durchschnittlich linearen Zeitaufwand. Auch das Ersetzen eines Elementes hat durchschnittlich linearen Zeitaufwand; darüber hinaus hat es auch noch linearen Platzbedarf, weil wegen der referentiellen Transparenz alle vor dem ersetzten Element stehenden Kettenglieder kopiert werden müssen.

Auch wenn Listen prinzipiell ähnlich benutzt werden können wie Felder, muss man immer beachten, dass "beliebige" Zugriffe auf Listenelemente teuer sind. Wegen der referentiellen Transparenz können Felder in einer funktionalen Sprache nicht genau so realisiert werden wie etwa in JAVA. Wollte man Felder auch in HASKELL als zusammenhängende Speicherzellen darstellen, müsste für jede Änderung einer Zelle ein komplettes Feld neu angelegt werden, mit linearem Speicheraufwand auf der Halde! Deshalb muss man einen Kompromiss mit akzeptabler Zugriffszeit und akzeptablem Speicheraufwand bei Änderungen finden. Der Modul `Array` aus der HASKELL-Standardbibliothek stellt funktionale Felder mit folgender Basis-Schnittstelle zur Verfügung:

```
data Ix a⇒ Array a b  -- abstract
array     :: (Ix a) ⇒ (a,a) → [(a,b)] → Array a b
listArray :: (Ix a) ⇒ (a,a) → [b] → Array a b
(!)       :: (Ix a) ⇒ Array a b → a → b
(//)      :: (Ix a) ⇒ Array a b → [(a,b)] → Array a b
```

Als Indexbereich können Wertepaare geeigneter Typen angegeben werden, die zur Klasse `Ix` gehören müssen. Die Basisdatentypen `Int`, `Integer`, `Char`, `Bool`, Aufzählungstypen sind Instanzen von `Ix`; außerdem auch alle Tupel über Indextypen, womit mehrdimensionale Felder realisiert werden können.

13.5.1 Einfache funktionale Felder

Wir wollen exemplarisch einen einfachen Modul für funktionale Felder implementieren. Dabei werden die Felder durch annähernd balancierte binäre Bäume dargestellt, die eine Indexmenge $\{1, \ldots, n\}$ haben. Der Zugriffspfad auf ein Element wird im Index kodiert, wie in Abbildung 13.1 illustriert wird.

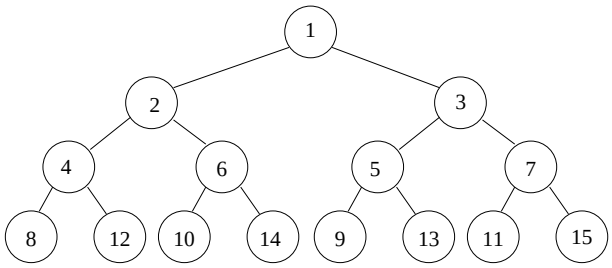


Abb. 13.1. Ein funktionales Feld als balancierter Baum

Die Schnittstelle des Moduls sieht so aus:

```
module FArray(
  Array, -- abstract
```

```
  (!),   -- :: Array a-> Int-> Maybe a,
  upd,   -- :: Array a-> Int-> a-> Array a,
  remv,  -- :: Array a-> Int-> Array a
  ) where
```

Der Basisdatentyp ist ein binärer Baum:

```
data Tree a = Leaf | Node a (Tree a) (Tree a)
              deriving (Show, Read)
type Array a = Tree a
```

Die Darstellung verlangt als *Invariante:*, dass die Indexmenge des Feldes zusammenhängt.

Der lesende Zugriff `(!)` durchläuft den Baum:

```
(!) :: Tree a → Int→ Maybe a
Leaf ! _           = Nothing
(Node v t1 t2) ! k
   | k == 1    = Just v
   | even k    = t1 ! (k `div` 2)
   | otherwise = t2 ! (k `div` 2)
```

Das selektive Verändern eines Elementes geschieht analog. In dieser vereinfachten Variante des Moduls kann das Feld immer nur um ein Element erweitert werden.

```
upd :: Tree a → Int→ a→ Tree a
upd Leaf k v =
  if k== 1 then (Node v Leaf Leaf)
  else error "Tree.upd:␣illegal␣index"
upd (Node w t1 t2) k v
  | k == 1    = Node v t1 t2
  | even k    = Node w (upd t1 (k `div` 2) v) t2
  | otherwise = Node w t1 (upd t2 (k `div` 2) v)
```

Mit `remv` kann immer nur ein kompletter Teilbaum entfernt werden.

```
remv :: Tree a→ Int→ Tree a
remv Leaf _  = Leaf
remv (Node w t1 t2) k
  | k == 1    = Leaf
  | even k    = Node w (remv t1 (k `div` 2)) t2
  | otherwise = Node w t1 (remv t2 (k `div` 2))
```

Bei `remv` wird die Invariante *nicht geprüft.* Diesem Mangel kann einfach abgeholfen werden, wenn der Typ wie folgt erweitert wird:

```
type Array a = (Tree a, Int)
```

In Tabelle 13.1 stellen wir die Eigenschaften von Listen, imperativen und funktionalen Feldern noch einmal zusammen.

Eigenschaft	Listen	Felder	
		funktional	imperativ
Konzept	Sequenzen / Wörter	Abbildungen	
abstrakte Definition	α^*	$Ix \to \alpha$	
Zugriff	$\ell!!i$	$a!i$	a(i)
Zeitaufwand	$\mathcal{O}(n)$	$\mathcal{O}(\log n)$	$\mathcal{O}(1)$
Ändern		$a//[(i,e)]$	a(i) = e
Zeit-/Speicheraufwand	$\mathcal{O}(n)$	$\mathcal{O}(\log n)$	$\mathcal{O}(1)$

Tabelle 13.1. Eigenschaften von Listen und Feldern

(Dies ist Fassung 1.10.10 von 4. Februar 2010.)

14

Rückblick und Ausblick

In diesem Kapitel wollen wir noch einmal rückblickend das Wesentliche der vorangegangenen Kapitel zusammenfassen, auf mögliche Vertiefungen des hierin behandelten Stoffes hinweisen und einen Ausblick in die Zukunft des Programmierens mit deklarativen, regelbasierten Sprachen wagen.

14.1 Logik-Programmieren

Das Wesen der Logik-Programmierung lässt sich so zusammenfassen:

- Ein logisches Programm definiert ein "Universum", indem es Fakten und Regeln dafür angibt. In dem Universum gilt das – und nur das – was die Regeln definieren (*closed world assumption*).
- Fakten und Regeln definieren *Prädikate* über *Termen*; das sind Konstruktorausdrucke, die baumartige Werte mit Variablen ("Löchern") darstellen, die in dem definierten Universum existieren.
- Die Ausführung eines logischen Programms beantwortet eine *Anfrage*, in der nachgefragt wird, ob ein oder mehrere Prädikate für bestimmte Parameter-Terme gelten. Wenn die Terme Variablen enthalten, wird durch *Resolution* mit den Regeln und *Unifikation* der darin enthaltenenen atomaren Formeln ein Beispiel für Belegungen dieser Variablen angegeben, für die die Prädikate gelten.
- Durch *backtracking* können auch weitere Beispiele für Variablenbelegungen aufgezählt werden.

Nur in Logik-Programmen kann man auf diese Weise mit Termen und Variablen rechnen und mittels *backtracking* Ergebnisse aufzählen lassen.

14.1.1 Einsatzfelder des Logik-Programmierens

Die Eigenheiten des Logik-Programmierens legen Anwedneungen nahe, in denen nichtdeterministisch mit Prädikaten und Relationen gerechnet wird. Einige sollen hier erwähnt werden:

- Jede *Datenbank* definiert Daten und Realationen darauf und kann insofern mit den Fakten und Regeln einer logischen Programmiersprache modelliert werden. Diese ANalogie ist schon die Terminologie des Logik-Programmierens verankert (*Datenbasis*, *Anfragen*). Wegen der Möglichkeit des logischen Schließens gilt das auch für *deduktive Datenbanken*.
- Der Nichtdeterminismus er möglicht es auch, *wissensbasierte Systeme*, und andere Systeme der künstliche Intelligenz zu realisieren.
- Außerdem eignen sich logische Sprachen auch dazu, die *Analyse natürlicher Sprachen* zu implementieren, die ja auch ein Teilgebiet der künstlichen Intelligenz ist. Dies war auch eine wichtige Motivation für den Entwurf von PROLOG durch Alain Colmerauer [Col78].
- Die Fakten und Regeln eines logischen Programms sind Terme über logischen Konnektoren. Insofern lassen sie sich als Objekte eines logischen Programms darstellen, mit dem man dann logische Programme selbst transformieren kann. In anderen Worten, logische Sprachen erlauben es, *Objektsprache* und *Metasprache* zu verschmelzen. In PROLOG gibt es dazu die vordefinierten Prädikate `asserta/1` `assertz/1` und `retract/1`, mit denen als Terme konstruierte Regeln und Fakten zur Datenbasis hinzugefügt bzw. aus ihr entfernt werden können. (Weil in PROLOG die Reihenfolge der Regeln immens wichtig ist, gibt es zwei Möglichkeiten des Hinzufügens: `asserta` fügt vorne und `assertz` hinten an die Datenbasis an.)

In [NM95] werden diese Einsatzfelder genauer beschrieben.

14.1.2 Bemerkungen zu Prolog

PROLOG ist nicht nur die älteste, sondern auch immer noch die mit Abstand am meisten verbreitete logische Programmiersprache.

Eine Stärke von PROLOG ist sicher die *Einfachheit* der Sprache: Es gibt wenig mehr zu lernen als die Syntax der Prädikatenlogik (mit anderen Konnektoren) und eine Bibliothek von vordefinierten Prädikaten.

Gerade im Vergleich zu "klassischen Programmiersprachen" wie HASKELL und auch JAVA fallen aber auch einige Schwächen auf:

- *Typisierung* fehlt vollständig. Jeder geklammerte Ausdruck über Atomen und Variablen ist ein Term. Deshalb hat man keinerlei Sicherheit vor Tippfehlern: "`willi`" und "`willy`" sind einfach zwei verschiedene Atome, auch wenn es sich bei einem von Beiden vielleicht um einen Schreibfehler handelt. Dementsprechend sind Listen *heterogen*: Sie können durcheinander Zahlen, Zeichen und andere Listen enthalten.
- Alle Berechnungen in PROLOG müssen als Prädikate definiert werden, auch wenn in einigen Fällen "einfach nur" Funktionen, also deterministische Prädikate, bei denen die Parameter immer als Eingaben (instanbziierte Terme) oder immer als Resultate (uninstanziierte Variablen) benutzt werden. Alle Funktoren sind reine Daten-Konstruktoren.

- Programme, die sich selbst mit `assert` und `retract` modifizieren können, sind zwar sehr mächtig, aber zu mächtig, als das man über ihr Verhalten allgemeine Eigenschaften nachweisen könnte.
- *Backtracking* ist ineffizient, auch wenn dies für bestimmte Anwendungen so wichtig sein mag, dass man dies gerne in Kauf nimmt – weil einfacher programmiert werden kann.

14.1.3 Logisches Programmieren nach Prolog

An mehreren Stellen ist PROLOG weiterentwickelt worden, um einige der oben genannten Schwächen zu beseitigen. Hier sollen nur einige genannt werden.

- Die Sprache MERCURY, entwickelt in den den Neunzigern and der University of Melbourne [HCSJ96], führt eine Typisierung für Terme ein. Außerdem können *Modes* für Prädikate angegeben werden. Damit kann man angeben, welche Parameter eines Prädikates Eingabewerte sind, die bei Aufrufen bereits instantiiert sind, und welche Resultate sind, für die typischerweise uninstantiierte Variablen angegeben werden, deren Belegung das "Ergebnis" des Prädikats liefern soll.
- Die Sprache CURRY, entwickelt an der CAU Kiel [Cur06], kombiniert funktionales und logisches Programmieren, wobei der funktionale Anteil stark an Haskell erinnert. Es gibt also polymorphe Datentypen und Funktionen höherer Ordnung. Zusätzlich können auch nichtdeterministische Prädikate definiert und ähnlich wie in PROLOG Anfragen gestellt werden.
- Die Sprache OZ und die dazu gehörende Entwicklungsumgebung M**oz**art wurden am DFKI in Saarbrücken entwickelt [Moz06]. Diese Sprache kombiniert noch mehr Programmierstile: logisches, funktionales, objektoriuentiertes und nebenläufiges Programmieren, im Grunde also *alle* bekannten Stile. In der Sprache OZ können auch *constraints* behandelt werden, also mit Gleichheiten und Ungleichheiten auf Zahlöen gerechnet werden. In OZ können alle Programmierstile benutzt und kombiniert werden, wie im Lehrbuch [VRH04] vorgeführt wird.

14.2 Funktionales Programmieren

Der Kern des funktionalen Programmierens sieht so aus:

- Ein funktionales Programm definiert *mathematische Funktionen* (im allgemeinen: partielle), die Einagbewerten eindeutige Resultate zuordnen.
- Die Funktionen werden mit *rekursive Gleichungen* spezifiziert.
- Bei der Ausführung eines funktionalen Programms wird ein Ausdruck *ausgewertet*, indem die Gleichungen auf ihn angewendet werden, solange das möglich ist.

- Die einfache Struktur macht es leicht, *Beweise* über Termination und Korrektheit funktionaler Programme zu führen, durch strukturelle und Fixpunktinduktion.

Dies ist allen funktionalen Programmiersprachen der Fall. Die meisten modernen funktionale Sprachen, also auch Haskell, bieten zusätzlich folgende Konzepte:

- Werte werden als *algebraische Datentypen* konstruiert, die *parametrisiert* sein können.
- Alle Ausdrücke und Funktionsdefinitionen sind *statisch getypt*, so dass die Wohlgeformtheit eines Programms bei der Übersetzung festgestellt werden kann.
- Funktionen können *polymorph* definiert werden, so dass sie auf einer Vielzahl von ähnlichen Typen aus gleiche Weise arbeiten.
- Funktionen sind Werte erster Klasse, die als Parameter und Ergebnisse anderer Funktionen benutzt werden dürfen. Mit solchen *Funktionen höherer Ordnung* können sehr mächtige Kombinatoren definiert werden, die benutzerdefinierte Kontrollstrukturen realisieren.
- Abstrakte Datentypen und Module werden unterstützt.

Haskell und einige andere Sprachen haben noch einige weitergehenden Eigenschaften:

- Die *verzögerte Auswertung* erlaubt den Umgang mit unendliche Datenstrukturen und öffnet damit neue Wege, Funktionen zu definieren.
- *Überladene Funktionen*, d.h., die Benutzung eines Namens für mehrere, verschiedene Funktionsdefinitionen, kann in Haskell durch Typklassen realisiert werden (auch für Literale wie Zahlen).
- Zustandsbehaftete Berechnungen, die beispielsweise auf die unterliegende Plattform zugreifen, werden gekapselt; in Haskell geschieht das durch den abstrakten Datentyp `IO`.

Alle diese Eigenschaften haben wir kennengelernt. Sie werden vom Standard von Haskell unterstützt.

14.2.1 Haskell

Obwohl vieles in diesem Text auf die Sprache Haskell zugeschnitten war, haben die Zeit und der Platz nicht gereicht, alle Konzepte von Haskell zu behandeln:

- Die Felder (**array**) aus zustandsorientierten Sprachen speichern Elemente innerhalb eines zusammenhängenden Indexbereiches in einem ebenfalls zusammenhängenden Speicherbereich ab. Der lesende und schreibende Zugriff auf ein Feldelement hat konstanten Aufwand. Deshalb ist diese Datenstruktur in imperativen und objektorientierten Sprachen so wichtig

und beliebt. Die Effizienz beruht aber entscheidend auf der Tatsache, das schreibender Zugriff auf Felder selektiv ist.
Das wiederspricht dem Prinzip der *referentiellen Transparenz*. Deshalb können Felder nicht einfach auf funktionale Sprachen übertragen werden. In HASKELL sind Felder *endliche Abbildungen*, die z.B. als balancierte Bäume dargestellt werden können.

-

  Fortgeschrittene Konzepte von HASKELL

- Felder (`array`) siehe Kapitel "Effizienz"
- Nebenläufigkeit (*concurrent* HASKELL)
  - *Threads* in Haskell:
    - `forkIO :: IO () -> IO ThreadID`
    - `killThread :: ThreadID -> IO ()`
  - Zusätzliche Primitive zur Synchronisation
  - Erleichtert Programmierung *reaktiver Systeme*
- Schnittstelle zu C

  Grafische Benutzungsschnittstellen für HASKELL

- *HTk*
  - Verkapselung von Tcl/Tk in Haskell
  - Nebenläufig mit *Events*
  - Entwickelt an der AG BKB (Dissertation E. Karlsen)
  - Mächtig, abstrakte Schnittstelle, mittelprächtige Grafik
- *GTk+HS*
  - Verkapselung von GTk+ in Haskell
  - Zustandsbasiert mit call-backs
  - Entwickelt an der UNSW (M. Chakravarty)
  - Neueres Toolkit, ästethischer, nicht ganz so mächtig

  Bewertung von HASKELL

- Stärken
  - Abstraktion durch
    - Polymorphie und Typsystem
    - algebraische Datentypen
    - Funktionen höherer Ordnung
  - Flexible Syntax
  - Ausgereifter Compiler
  - Bibliothek

- Schwächen
  - Komplexität
  - Dokumentation
  - Noch viel im Fluß

  Andere funktionale Sprachen

- Lisp (Common Lisp, Scheme)
  - dynamische Typisierung, Seiteneffekte
- ML [Pau91] (Edinburgh, Caml, O'Caml)
  - Polymorphie, Module, kontrollierte Seiteneffekte
- Miranda (Kent, Turner) [Tur86]
  - Mutter von Haskell
- Clean (RU Nijmegen) [BvEvLP87]
  - Stiefschwester von Haskell
  - effiziente Implementierung, grafische Benutzungsschnittstelle

Einsatzfelder funktionaler Sprachen

- Common Lisp
  - künstliche Intelligenz (fragt Diedrich Wolter)
- Erlang (Firma Ericsson)
  - Telekommunikation
- ML
  - Theorembeweiser
- Haskell
  - Casl-Entwicklungsumgebung

Grafische Benutzerschnittstellen

- *HTk*
  - Verkapselung von Tcl/Tk in Haskell
  - Nebenläufig mit *Events*
  - Entwickelt an der AG BKB (Dissertation E. Karlsen)
  - Mächtig, abstrakte Schnittstelle, mittelprächtige Grafik
- *GTk+HS*
  - Verkapselung von GTk+ in Haskell
  - Zustandsbasiert mit call-backs
  - Entwickelt an der UNSW (M. Chakravarty)
  - Neueres Toolkit, ästethischer, nicht ganz so mächtig

Warum funktionale Programmierung lernen?

- Abstraktion
  - Denken in Algorithmen, nicht in Programmiersprachen
- FP konzentriert sich auf *wesentlichen* Elemente moderner Programmierung:
  - Typisierung und Spezifikation
  - Datenabstraktion
  - Modularisierung und Dekomposition
- Blick über den Tellerrand — Blick in die Zukunft

– Studium ≠ Programmierkurs— was kommt in 10 Jahren?

Hat es sich gelohnt?

(Dies ist Fassung 1.12.01 von 4. Februar 2010.)

A

Haskell for Fun

In diesem Kapitel versuchen wir die Sprache HASKELL systematisch zu beschreiben. Wir konzentrieren uns dabei auf das Wesentliche (für diese Lehrveranstaltung) und vernachlässigen manche Besonderheiten. Dies geschieht in der Hoffnung, dass dies besser verständlich ist als die vollständige Beschreibung [PJ+02], die in allen Fragen konsultiert werden sollte, die dieser Anhang offen oder unklar lässt.

A.0.1 Konventionen für die Beschreibung der Syntax

In den folgenden Abschnitten benutzen wir folgende Konventionen für die Beschreibung von Programmtexten:

- "*Programmteil* ::= *Form*" definiert die textuelle Form eines Programmteils.
- Senkrechte Striche "|" trennen alternative Formen eines Programms.
- Von eckigen Klammern "$[\cdots]$" umschlossene Programmteile sind *optional*, d.h., sie dürfen fehlen.
- Ausdrücke wie "$\mathit{Import}_1; \ldots;\ \mathit{Import}_k$" beschreiben Aufzählungen, wobei die Indizes k, m, n, l die Anzahl möglicher Elemente in Folgen angeben; wenn nichts anderes gefordert wird, gilt dabei $k, m, n, l \geqslant 0$; für den Fall $k = 0$ kann das Trennzeichen (in diesem Beispiel ";") fehlen.
- $\mathcal{V}$ ist eine Menge von Bezeichnern für *Variablen* wie x, y und f.
- $\mathcal{O}$ ist eine Menge von Bezeichnern für *Operatoren* wie $\oplus$, $\otimes$ usw.
- $\mathcal{Y}$ ist eine Menge von *Typvariablen* wie α, β usw.
- $\mathcal{K}$ ist eine Menge von (*Wert-*) *Konstruktoren* wie K.
- $\mathcal{T}$ ist eine Menge von *Typkonstruktoren* wie T.
- $\mathcal{C}$ ist eine Menge von Klassenbezeichnern wie C.
- X ist irgendein Bezeichner für eine Variable, einen Wert- oder Typkonstuktor oder eine Klasse.
- $\mathcal{M}$ ist eine Menge von *Modulbezeichnern* wie M.
- $\mathcal{X} \cap \mathcal{Y} \neq \emptyset$.

- $\mathcal{T} \cap \mathcal{C} = \emptyset$.
- $\mathcal{K} \cap (\mathcal{T} \cup \mathcal{C}) \cap \mathcal{M} \neq \emptyset$.

A.1 Programme und Module

Ein *Programm* besteht aus einem Modul und all den Modulen, die dieses Modul direkt oder indirekt importiert. Die Module des Vorspanns (*Prelude*) beschreiben die vordefinierten Datentypen mit ihren Funktionen und werden implizit von jedem Modul importiert.

Jedes *Modul* namens M wird in eine Datei mit Namen M`.hs` oder M`.lhs` geschrieben. Es hat folgende Form:

$$\begin{array}{l}[\ \textbf{module}\ M\ [\ (X_1, \ldots, X_m)\ ]\ \textbf{where}\ ] \\ \quad \{\mathit{Import}_1; \ldots;\ \ \mathit{Import}_k;\ \ \mathit{Top\text{-}Declaration}_1; \ldots;\ \ \mathit{Top\text{-}Declaration}_n\}\end{array}$$

Die *Modulüberschrift* (*header*) definiert seinen Namen M und die vom ihm exportierten Vereinbarungen; sie kann entfallen, wenn der Modul nicht von anderen importiert werden soll. Der *Rumpf* (*body*) des Moduls kann anderer Module importieren und eine Menge von *Vereinbarungen* (*declaration*) enthalten, von denen diejenigen exportiert werden, deren Namen in seiner Exportschnittstelle ($\mathit{Export}_1, \ldots, \mathit{Export}_m$) genannt werden.

Die *Exporte* $X_1, \ldots, X_m$ bezeichnen Variablen, Tykkonstruktoren und Klassen, die in den Vereinbarungen des Moduls eingeführt oder aus anderen Modulen importiert wurden. Ein Typkonstruktor T eines Datentyps (nicht eines Typsynonyms) wird ohne die von T definierten Wertkonstruktoren exportiert. Die von einer Typklasse C spezifizierten Variablen werden ebenfalls nicht mit exportiert.

A.1.1 Importe

Ein *Import* hat folgende Form

$$\textbf{import}\ \ [\ \textbf{qualified}\ ]\ M\ [\ \textbf{as}\ L]\ \big[[\ \textbf{hiding}\ ](X_1, \ldots, X_n)\big]$$

und macht alle von M exportierten Namen von Vereinbarungen sichtbar (*visible*). Mit "$(X_1, \ldots, X_n)$" werden nur die exportierten Bezeichner $X_1, \ldots, X_n$ importiert, mit "**hiding** $(X_1, \ldots, X_n)$" alle exportierten Bezeichner außer $X_1, \ldots, X_n$.

Analog zum Export wird ein Typkonstruktor ohne die für ihn definierten Wertkonstruktoren importiert, und eine Typklasse C ohne die für sie Spezifizioerten Variablen. [1]

Mit dem Zusatz "**as** L" wird M lokal in L umbenannt, und der Zusatz "**qualified**" erzwingt, dass jeder importierte Namen X_i qualifiziert benutzt wird, d.h. als "$M.X_i$" bzw. "$L.X_i$".

[1] Ein Operator $\oplus$ muss mit der Schreibweise "$(\oplus)$" zu einem Bezeichner gemacht werden, um exportiert oder importiert werden zu können.

A.1.2 Vereinbarungen

Eine *Vereinbarung* definiert eines der folgenden Dinge:

1. ein Typsynonym
2. einen Datentyp
3. eine Typklasse
4. eine Instanz einer Typklasse
5. den Wert einer Variablen
6. die Typsignatur von Variablen
7. die Priorität und Assoziativität von Operatoren

Die Vereinbarungen 1-4 können nur global im Rumpf eines Moduls getroffen werden (*top declaration*); die Arten 5–7 können auch lokal in den Vereinbarungen von Klassen, Instanzen und Werten vorkommen.

Globale Vereinbarungen werden im nächsten Abschnitt beschrieben; die anderen in Abschnitt A.3.

A.2 Typen und Klassen

Mit Typsynonymen und algebraischen Datenbtypen werden Wertemengen eines Programms vereinbart.

Typklassen spezifizieren Anforderungen an Typen, und Instanziierungen definieren, wie Datentypen die Anforderungen einer Typklasse erfüllen. Typklassen und Instanziierungen werden hier beschrieben, weil sie in der HASKELL-Bibliothek oft benutzt werden; in einfacheren Programmen werden sie kaum gebraucht .

A.2.1 Typsynonyme

Eine *Typsynonymdefinition*

$$\textbf{type}\ T\ \alpha_1 \cdots \alpha_k = t$$

führt den Bezeichner T als Abkürzung (mit $k \geqslant 0$ voneinander verschiedenen Typvariablen $\alpha_1, \ldots, \alpha_k$) für den Typausdruck t ein. Bei jeder Benutzung $T\ t_1 \cdots t_k$ des Typsononyms mit Typausdrücken t_i $(1 \leqslant i \leqslant k)$ ist es eine Abkürzung für den Typausdruck $t[\alpha_1/t_1 \cdots \alpha_k/t_k]$.[2]

[2] Hier bezeichnet $t[\alpha_i/t_i]$ die Substitution von allen Auftreten von α_i in t durch t_i.

A.2.2 Datentypen

Eine *algebraische Datentyp-Definition*

$$\begin{aligned}\textbf{data}\ [(C_1\,\beta_1,\cdots,C_k\,\beta_h)\Rightarrow] \\ T\,\alpha_1\cdots\alpha_k &= K_1\,t_{1,1}\ldots t_{1,m_1} \\ &\vdots \\ &\mid K_n\,t_{i,1}\ldots t_{i,m_n}\ [\,\textbf{deriving}\ (C_1,\ldots,C_l)\,]\end{aligned}$$

definiert einen neuen algebraischen Datentyp T mit $k \geqslant 0$ verschiedenen Typvariablen und $n \geqslant 1$ Wertkonstruktoren $K_i\colon t_{i,1} \to \cdots \to t_{i,m_i} \to T\,\alpha_1\cdots\alpha_k$, die jeweils m_i Komponenten vom Typ $t_{i,j}$ haben ($1 \leqslant j \leqslant m_i$).

T ist ein *Aufzählungstyp* wenn $m_i = 0$ für $1 \leqslant i \leqslant n$. (Dann sollte auch $k = h = 0$ gelten.) T ist ein *Produkttyp* wenn $n = 1$ gilt. Sonst ist T ein *Summentyp*.

Ein Typkonstruktor T *benutzt* einen Typkonstruktor T', wenn T' in einem der Typausdrücke $t_{i,j}$ benutzt wird, oder wenn ein Typkonstruktor T'', der in irgendeinem $t_{i,j}$ auftaucht, T' benutzt. Ein Typkonstruktor T ist *rekursiv*, wenn er sich selbst benutzt. Rekursive Typen sollten Summentypen sein.

Mit dem Kontext $(C_1\,\beta_1,\cdots,C_k\,\beta_h)$ kann verlangt werden, dass die Typvariablen $\beta_j \in \{\alpha_1,\ldots,\alpha_k\}$ Instanzen der Typklassen C_j sind ($1 \leqslant j \leqslant h$), siehe Abschnitt A.2.5.

Ist der Zusatz "**deriving** $(C_1,\ldots,C_l)$" angegeben, werden automatisch die Operationen abgeleitet, die T zu einer Instanz der Typklassen $\{C_1,\ldots,C_l\} \subseteq \{$`Eq`, `Ord`, `Enum`, `Bounded`, `Show`, `Read`, $\}$ machen.[3]

A.2.3 Typklassen

Die Vereinbarung einer *Typklasse* hat die Form

$$\begin{aligned}&\textbf{class}\ [\,(C_1\,\alpha,\cdots,C_k\,\alpha)\Rightarrow\,]\,C\,\alpha \\ &\quad[\ \textbf{where}\ \{\mathit{Declaration}_1;\ldots;\mathit{Declaration}_n\}\,]\end{aligned}$$

Sie führt die Typlasse C ein und spezifiziert Operationen, die für alle Instanzen von C verfügbar sein müssen. Diese Spezifikation muss die Sigantur der Operationen festlegen und kann darüber hinaus auch für einige Operationen definieren, wie sie aus anderen Operationen der Klasse abgeleitet werden können.

[3] Instanzen von `Enum` können nur für Aufzählungstypen hergeleitet werden; Instanzen für `Bounded` können nur für Aufzählungstypen und Produkttypen hergeleitet werden.

A.2.4 Instanzen von Typklassen

Eine Instanziierung hat die Form

instance $[(C_1\ \alpha_1, \cdots, C_k\ \alpha_k) \Rightarrow]\ C\ t$
$[\ \textbf{where}\ \{Declaration_1; \ldots; Declaration_n\}\ ]$

Sie macht den Typ t zu einer Instanz der Klasse C, wenn die im Typausdruck t enthaltenen Typvariablen α_i Instanzen der Klassen C_i sind.

A.2.5 Kontexte

A.2.6 Typausdrücke

Typausdrücke ist entweder eine Typvariable, oder die Anwendung eines *Typkonstruktors* auf $k \geqslant 0$ Typausdrücke.

$$\frac{\alpha \in \mathcal{Y}}{\alpha \in \mathbb{T}} \quad \frac{T \in \mathcal{T}_k,\ t_1, \ldots, t_k \in \mathbb{T},\ k \geqslant 0}{T\ t_1 \cdots t_k \in \mathbb{T}} \quad \frac{t \in \mathbb{T}}{(t) \in \mathbb{T}}$$

$$\frac{}{() \in \mathcal{T}_0} \quad \frac{}{([\,]) \in \mathcal{T}_1} \quad \frac{}{(\rightarrow) \in \mathcal{T}_2} \quad \frac{k \geqslant 1}{(\underbrace{,\ldots,}_{k}) \in \mathcal{T}_{k+1}}$$

Der eingebaute Typkonstruktor `(->)` für Funktionsräume kann infix benutzt werden, und die vordefinierten Typkonstruktoren `([])` für Listen und `(,...,)` für Tupel können um ihre Typparameter herum geschrieben werden.

$$t_1 \rightarrow t_2 \rightarrow t_3 \equiv t_1 \rightarrow (t_2 \rightarrow t_3) \quad t_1 \rightarrow t_2 \equiv (\rightarrow)\ t_1\ t_2$$

$$[t] \equiv ([\,])\ t \quad (t_1, \ldots, t_k) \equiv (,\ldots,)\ t_1\ \cdots\ t_k$$

Tabelle A.1 gibt den Vorrang und die Assoziativität der vordefinierten Operatoren an.

A.3 Variablen

In Haskell werden alle Namen, die Werte bezeichnen, seien dies konstante Daten, Funktionen oder infix geschriebene Operationen, *Variablen* genannt. Dies kommt daher, dass diese Namen frei vom Programmierer gewählt werden können. Ist ein Name allerdings vereinbart, ist seine Bedeutung unveränderlich, solange die Vereinbarung gilt – ganz anders als Variablen in anderen Programmiersprachen.

Funktionen, Signaturen von Funktionen und die Priorität und Assoziativität von Infix-Operatoren können nicht nur global in Modulen vereinbart werden, sondern auch in den Rümpfen von Klassen und Instanziierungen und lokal in Funktionsvereinbarungen.

$\mathcal{O}_i^a$	Assoziativität		
Vorrang	links	nicht	rechts
0			$, $!, ‘seq‘
1	>>, >>=		
2			||
3			&&
4		==, / =, <, <=, >, >=, ‘elem‘, ‘notElem‘	
5			:, ++
6	+, −		
7	*, /, ‘div‘, ‘mod‘, ‘rem‘, ‘quot‘		
8			^, ^^, **
9	!!		

Tabelle A.1. Vorrang und Assoziativität von vordefinierten Operatoren

A.3.1 Funktionen

Eine Funktionsdefinition kann verschiedene Formen haben.[4] Häufig wird eine Funktion f mit mehreren Gleichungen folgender Form definiert:

$$f\ P_1 \cdots P_n = E \\ \qquad [\textbf{where}\ \{\mathit{Declaration}_1; \ldots; \mathit{Declaration}_n\}]$$

In der Gleichung wird der Fall betrachtet, in dem die aktuellen Parameter von f nach dem Mustern $P_1, \ldots, P_n$ aufgebaut sind. Mit **where** können Hilfsdefinitionen eingeführt werden, die auf der rechten Seite der Gleichung, im Ausdruck E, benutzt werden können.

Eine Gleichung kann auch *bedingt* sein:

$$\begin{array}{rl} f\ P_1 \cdots P_n & \mid \quad G_1 \quad = E_1 \\ & \mid \quad G_2 \quad = E_2 \\ & \qquad \vdots \\ & \mid \quad G_k \quad = E_k \\ & \mid \mathit{otherwise} = E_{k+1} \\ & \quad [\textbf{where}\ \{\mathit{Declaration}_1; \ldots; \mathit{Declaration}_n\}] \end{array}$$

In diesem Fall wird zusätzlich überprüft, of die Parameter der Funktion f einen der Wächter (*guards*), gegeben als Boole'sche Ausdrücke $G_1, \ldots, G_k$,

[4] Definitionen von Werten, die keine Funktionen sind, werden nicht gesondert behandelt. Für ihre Definition gilt $n = 0$.

erfüllen. Es wird der erste Ausdruck E_i ausgewählt, dessen Wächter G_i den Wert `True` ergibt.

Besonders in lokalen Definitionen (nach **where** oder **let**) kann eine weitere Form der Funktionsgleichung praktisch sein;

$$P = E$$
$$\quad [\textbf{where } \{\mathit{Declaration}_1; \ldots; \mathit{Declaration}_n\}]$$

Hier bildet ein Muster P die linke Seite. Wenn der Ausdruck E immer Werte der Form p liefert, kann so das Ergebnis der Hilfsdeinition in Komponenten zerlegt werden. Beispielsweise werden mit der Definition "$(x, y) = E$" gleich auf die Elemente x und y eines Paares (x, y) selektiert.

A.3.2 Signaturen

Eine Signaturspezifikation definiert den Typ von Variablen.

$$x_1, \ldots, x_n :: [\ \mathit{Context} \Rightarrow]\ \mathit{Type}$$

A.3.3 Priorität und Assoziativität von Operatoren

Eine sogenannte *"Fixity"-Vereinbarung* hat die Form

$$(\textbf{infixl} \mid \textbf{infixr} \mid \textbf{infix})\ [k]\ \oplus_1, \ldots, \oplus_n$$

Sie definiert, dass die Operatoren $\oplus_1, \ldots, \oplus_n$ infix mit dem *Vorrang* (*priority*) $0 \leqslant k \leqslant 9$ benutzt werden können und entweder *linksassoziativ* (**infixl**) oder *rechtsassoziativ* (**infixr**) oder *nicht assoziativ* (**infix**) ist.

Wir teilen die Ausdrücke $\mathbb{E}$ in Ausdrücke der Prioritäten $i = 1 \ldots 10$ ein.

$$\frac{E, E' \in \mathbb{E}_{i+1}, \oplus \in \mathcal{O}^i_n}{E \oplus E' \in \mathbb{E}_i} \quad \frac{E \in \mathbb{E}_i, E \in \mathbb{E}_{i+1}, \oplus \in \mathcal{O}^i_l}{E \oplus E' \in \mathbb{E}_i} \quad \frac{E \in \mathbb{E}_{i+1}, E \in \mathbb{E}_i, \oplus \in \mathcal{O}^i_r}{E \oplus E' \in \mathbb{E}_i} \quad \frac{E \in \mathbb{E}_{i+1}}{E \in \mathbb{E}_i}$$

A.4 Ausdrücke

Ausdrücke berechnen Werte durch geschachtelte Anwendung von Funktionen auf Argumente.

$$\frac{x \in \mathcal{V}}{x \in \mathbb{E}} \quad \frac{K \in \mathcal{K}}{K \in \mathbb{E}} \quad \frac{l \in \mathbb{L}}{l \in \mathbb{E}} \quad \frac{F, A \in \mathbb{E}}{F\ A \in \mathbb{E}} \quad \frac{E \in \mathbb{E}}{(E) \in \mathbb{E}}$$

$$\frac{P_1, \ldots P_n \in \mathbb{P},\ E \in \mathbb{E}}{\lambda\, P_1 \cdots P_n \to E \in \mathbb{E}} \quad \frac{D_1, \ldots D_n \in \mathbb{D},\ E \in \mathbb{E}}{\textbf{let } \{D_1; \ldots; D_n \textbf{ in } E \in \mathbb{E}}$$

$$\frac{E \in \mathbb{E}, P_1, \ldots, P_k \in \mathbb{P}, E_1, \ldots, E_k \in \mathbb{E}}{\textbf{case } E \textbf{ of } \{P_1 \to E_1; \ldots; P_k \to E_k\} \in \mathbb{E}} \quad \frac{E, E_1, E_2 \in \mathbb{E}}{\textbf{if } E \textbf{ then } E_1 \textbf{ else } E_2 \in \mathbb{E}}$$

A.4.1 Muster

Muster (*pattern*) sind eingeschränkte Ausdrücke über (Wert-) Konstruktoren.

$$\frac{x \in \mathcal{V}}{x \in \mathbb{P}} \quad \frac{}{\_ \in \mathbb{P}} \quad \frac{l \in \mathbb{L}}{l \in \mathbb{P}} \quad \frac{K \in \mathcal{K},\, P_1, \ldots, P_k \in \mathbb{P}}{K\ P_1\ \cdots\ P_k \in \mathbb{P}} \quad \frac{P \in \mathbb{P}}{(P) \in \mathbb{P}} \quad \frac{x \in \mathcal{V},\, P \in \mathbb{P}}{x@P \in \mathbb{P}}$$

Das Muster "x @ P" passt auf dieselben Werte wie das Muster P, bindet diesen Wert aber zusätzlich an die Variable x.

Muster von Listen und Tupeln können eine besondere Schreibweise haben.

$$[e_1, \ldots, e_k] \equiv e_1 : \cdots : e_k : [\,]$$
$$(e_1, \ldots, e_k) \equiv (, \ldots,)\ e_1\ \cdots\ e_k$$

Allgemeine Vereinfachungen

Diese Beschreibung vereinfacht einige Programmteile und lässt manche Konzepte ganz aus. Hier sollen sie der Vollständigkeit halber genannt werden.

- Die Infixform von Funktionen und Wert-Konstruktoren wird meistens nicht gesondert beschrieben.
- An vielen Stellen können bei geklammerten Listen $(E_1, \ldots, E_k)$ für den Fall $k = 1$ die Klammern weggelassen werden.

Feinere Exporte

Die Wertkonstruktoren eines algebraischen Datentypkonstruktors T können mit der Schreibweise "$T(..)$" eportiert werden. Schreibt man "$T(K_1, \ldots, K_n)$", werden nur die genannten Konstruktoren exportiert.

Auch die Variablen einer Typklasse C können mit der Schreibweise "$C(..)$" eportiert werden. Mit "$C(x_1, \ldots, x_n)$" werden nur die genannten Variablen exportiert.

Schreibt man "**module** M", werden alle Dinge eines Moduls exportiert. Ist M ein importiertes Modul, sind dies alle von M exportierten Dinge; ist M der Name des gerade definierten Moduls, sind dies alle in ihm vereinbarten Dinge, jedoch *ohne* die der von ihm importierten Module.

Feinere Importe

Auch beim Import können alle Wertkonstruktoren eines algebraischen Datentypkonstruktors T mit der Schreibweise "$T(..)$" imortiert werden, und ausgewählte Wertkonstruktoren mit "$T(K_1, \ldots, K_n)$"; auch alle Variablen einer Typklasse C können mit "$C(..)$" imortiert werden, bzw. ausgewählte Variablen mit "$C(x_1, \ldots, x_n)$".

Feinere Datentypen

Konstruktoren mit *Feldnamen*

strikte Konstruktoren

Konstruktor-Operatoren

Konstruktor-Klassen

— qtycls (tyvar atype1 ... atypen) (n¿=1)

Neue Typen (newtype)

newtype [*Context* $\Rightarrow$] $T\ \alpha_1 \cdots \alpha_k = K\ t_1 \ldots t_m$ [**deriving** $(C_1, ..., C_n)$]

Numerische Default-Typen

default $(T_1, \ldots, T_n)(n >= 0)$

Vereinfachungen der syntaktischen Beschreibung:

- Einsetzen von selten gebrauchten Regeln
- Prioritäten in Ausdrücken usw.
- Optionale Qualifizierung von Namen

A.5 Die Struktur des Programmtextes

Programme sind Texte, also Zeichenketten, die in Dateien abgespeichert werden können. Wie gewöhnliche Texte bestehen Programme aus Wörtern, Satzzeichen und Zwischenräumen. Diese Textbausteine werden *Lexeme* genannt. Die Lexeme von HASKELL sind:

- Wörter sind einerseits *Namen*, die für Dinge im Programm eingeführt werden (siehe Abschnitt A.5.1), andererseits *Literale*, die Werte von Zahlen, Zeichen und Zeichenketten angeben (siehe Abschnitt A.5.2).
- Als Satzzeichen dienen neun *Spezialzeichen* `(),;[]‘{}` und die *reservierten Namen* aus Abschnitt A.5.1.
- Zwischenraum (*whitespace*) besteht aus den *Layoutzeichen* Leerzeichen, Tabulator (horizontal und vertikal), Zeilenende (*line feed*), Wagenrücklauf (*carriage return*) und Seitenende (*form feed*). Mehr zur Rolle von Layout in HASKELL steht in Abschnitt A.5.4. Auch *Kommentare*, die das Programm dokumentieren, sind nur “Zwischenraum”, weil sie für die Bedeutung des Programms nicht ändern. (siehe Abschnitt A.5.3).

A.5.1 Namen

Viele Dinge in einem Programm können benannt werden. In HASKELL sind das Module, Klassen, Typen, (Wert-) Konstruktoren und Werte. Es gibt zwei Arten von Namen, mit jeweils zwei Unterarten:

- *Bezeichner* beginnen mit einem Buchstaben und können weitere Buchstaben, Ziffern, Unterstriche "_" und Striche "'" enthalten. (Allgemeiner als in der Mathematik üblich dürfen die Striche nicht nur am Ende eines Bezeichners auftreten; "`a''b'c`" ist also ein gültiger Bezeichner.) Es gibt zwei Arten von Bezeichnern:
  - *Konstruktorbezeichner* beginnen mit einem Großbuchstaben. Sie werden als Namen für *Wertkonstruktoren*, *Typkonstruktoren*, *Typklassen* und *Module* verwendet.
  - *Variablenbezeichner* beginnen mit einem Kleinbuchstaben. Sie benennen Werte oder Platzhalter – für Typen in Typdefinitionen bzw. für Werte in Wertvereinbarungen und Ausdrücken.
    Folgende 21 Variablenbezeichner sind als *Schlüsselwörter* reserviert:

    ```
    case    class     data    default  deriving  do
    else    if        import  in       infix     infixl
    infixr  instance  let     module   newtype   of
    then    type      where   _
    ```

    Sie können nicht als normale Variablenbezeichner verwendet werden.
- *Symbole* bestehen aus den folgenden 20 Sonderzeichen:

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

  Symbole bezeichnen zweistellige Funktionen, die in Infixschreibweise benutzt werden. Sie werden auch *Operatoren* genannt.
  - Symbole, die mit dem Doppelpunkt ":" beginnen, bezeichnen Wert-Konstruktor-Operationen.[5]
  - Symbole, die mit einem anderen Zeichen als ":" beginnen, bezeichnen Operationen.
  - Die folgenden 11 Symbole sind reserviert:

    ```
    ..  :  ::  =  \  |  <-  ->  @  ~  =>
    ```

    Viele weitere Operatoren sind in der Bibliothek vordefiniert.

Qualifikation. Namen können an manchen Stellen mit dem Namen des Moduls *qualifiziert* werden, der sie einführt (oder importiert). So bezeichnet "$M.x$" eine Variable x aus dem Modul M. Dies dient dazu, gleiche Namen, die in verschiedenen Modulen eingeführt wurden, zu unterscheiden.

[5] Ein einzelner Doppelpunkt ist reserviert für den Konstruktor (nicht-leerer) Listen, dem wichtigsten Typen in HASKELL.

Symbole als Bezeichner und umgekehrt. An manchen Stellen möchte man einen Operator wie "+" als einen Bezeichner verwenden, z. B. in der Exportliste eines Moduls. Dann muss man es in Klammern einschließen: "`(+)`".[6] (*section*). Umgekehrt kann der Wunsch aufkommen, den Bezeichner einer zweistelligen Funktion, wie die ganzzahlige Division `div`, als Infix-Operator zu schreiben. Dies kann man tun, wenn man den Funktionsbezeichner in "abfallende Hochkommata" (`‘`, *accent grave*) einfasst: `12 ‘div‘ 4`.

Mehrfache Benutzung von Konstruktorbezeichnern. Konstruktorbezeichner können im selben Modul gleichzeitig verschiedene Dinge benennen: In einem Modul `Stack` könne ein Datentyp `Stack` mit einem Wertkonstruktor `Stack` eingeführt werden. Der Kontext macht bei jeder Benutzung des Namens klar, ob das Modul, der Datentyp oder der Wertkonstruktor gemeint ist.

A.5.2 Literale

Literale bezeichnen die Werte vordefinierter Typen:

- *Numerische Literale* bezeichnen ganze oder gebrochene *Zahlen*; sie bestehen aus Ziffern und können einen Dezimalpunkt und einen mit "`e`" oder "`E`" eingeleiteten Exponentteil haben.
- Zeichen-Literale bezeichnen Werte des Typs `Char`; sie werden in einzelne Hochkommata "`’`" eingeschlossen. Dazu gibt es noch Möglichkeiten, Zeichen als oktale oder hexadezimale Zahlen anzugeben.
- Zeichenketten-Literale bezeichnen Werte des Typs `String`; sie werden in doppelte Anführungszeichen "`"`" eingeschlossen.

A.5.3 Kommentare

Kommentare sollen das Programm dokumentieren soll, werden aber ansonsten überlesen.

In Haskell gibt es zwei Arten von Kommentaren:

- *zeilenweise* Kommentare stehen zwischen von `--` und dem nächsten Zeilenende.
- mehrzeilige Kommentare werden zwischen `{-` und `-}` eingeschlossen. Dies Kommentare können auch geschachtelt werden.

Ein Beispiel zeigt beide Formen:

```
{- Fakultaetsfunktion fuer ganze Zahlen
   (c) 2009 Berthold Hoffmann          -}
fac :: Int → Int                      -- Signatur
fac n | n ≤  0   = error "fac:␣n≤0"
      | n == 1   = 1
      | otherwise = n * fac (n-1)
```

[6] In Haskell heißt so etwas ein Abschnitt (*section*).

Daneben gibt es noch eine alternative, *literarische* Sicht auf Programme, im Sinne des von Donald E. Knuth propagierten *Literate Programming* [Knu84]:[7] In *Literate Haskell* (normalerweise in Dateien mit der Endung `.lhs` statt `.hs` versehen) werden nicht die Kommentaranteile markiert, sondern umgekehrt der *Programmtext*. Alles andere ist Literatur. (*Ähm*, Kommentar.)

Hier gibt es wiederum zwei Möglichkeiten. Man kann alle Programmzeilen mit `>` in der ersten Spalte kennzeichnen:

```
Fakultaetsfunktion fuer ganze Zahlen
(c) 2009 Berthold Hoffmann

>fac :: Int → Int                        -- Signatur
>fac n | n <  0    = error "fac:␣n<0"
>      | n == 0    = 1
>      | otherwise = n * fac (n-1)
```

Das Beispiel zeigt: Die mit `--` eingeleiteten Zeilenkommentare gehen trotzdem noch.

Oder die Programmstücke werden zwischen `\begin{code}` und `\end\{code}`, wobei `\end{code}` in der ersten Spalte beginnen muss:

```
Fakultaetsfunktion fuer ganze Zahlen
(c) 2009 Berthold Hoffmann
\begin{code}
fac' :: Int → Int -- Signatur
fac' n | n <  0    = error "fac:␣n<0"
       | n == 0    = 1
       | otherwise = n * fac' (n-1)
\end{code}
```

Das ist besonders praktisch zur Benutzung mit LaTeX, weil man dann die Dokumentation eines Programms mit LaTeX setzen kann und *die gleiche Quelldatei* auch übersetzen kann. Dazu muss man `ghci` beispielsweise nur mit der Option `+l` überreden, auch Dateien mit der Endung `.tex` als *literate Haskell* zu verstehen.[8]

[7] Dort heißt es auf Seite 97: *I must confess that there may also be a bit of malice in my choice of a title. During the 1970s I was coerced like everybody else into adopting the ideas of structured programming, because I couldn't bear to be found guilty of writing* unstructured *programs. Now I have a chance to get even. By coining the phrase "literate programming," I am imposing a moral commitment on everyone who hears the term; surely nobody wants to admit writing an* illiterate *program.*

[8] Der vorliegende Text ist so formatiert, dass seine Kapitel literarische Skripte sind – nicht jedoch dieser Anhang!

A.5.4 Layout

in einem HASKELL-Programm können an vielen Stellen Layoutzeichen eingefügt werden, aber im Gegensatz zu anderen Sprachen (auch JAVA) ist das Layout nicht immer egal.

In HASKELL gilt die Abseitsregel (*offside rule*). Für eine Funktionsdefinition

$$\mathtt{f}\,\mathtt{x}_1\,\mathtt{x}_2\ldots\mathtt{x}_n = \mathtt{E}$$

gilt: Alles, was gegenüber `f` in den folgenden Zeilen eingerückt ist, gehört noch zur Definition von `f`. Erst mit der nächsten Zeile, die in der gleichen Spalte wie `f` beginnt, fängt eine neue Definition an.

```
f x = hier faengts an
  und hier gehts weiter
      immer weiter
g y z = und hier faengt was neues an
```

Das gilt auch bei verschachtelten Definitionen.

Diese Regel ist vielleicht zunächst gewöhnungsbedürftig. Sie ist trotzdem sinnvoll, weil man so durch Einrückungen gruppieren und sequenzialisieren kann, wozu in anderen Sprachen wie JAVA Klammerungen mit "{" und "}" und Trennzeichen ";" benötigt werden. (In HASKELL können diese Zeichen in genau der gleichen Funktion benutzt werden, wenn sonst die Abseitsregel verletzt wäre; wir werden das aber kaum brauchen. Allerdings erscheinen sie manchmal in Fehlermeldungen, wenn die Abseitsregel verletzt wurde.)

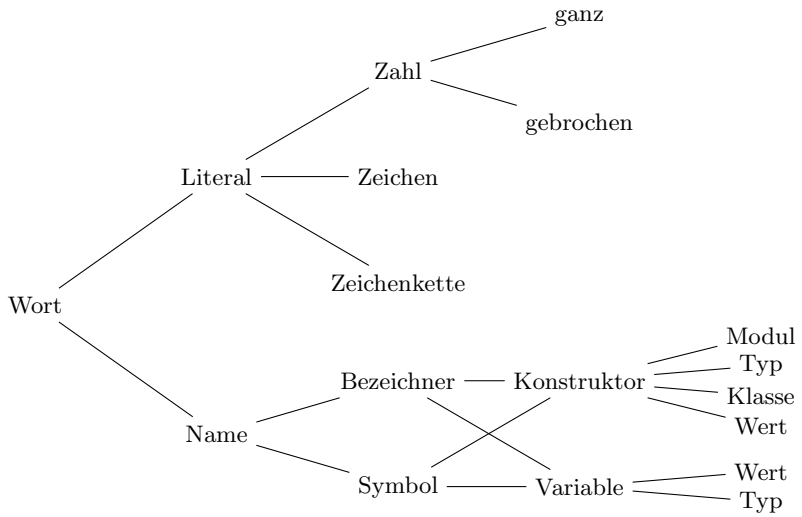


Abb. A.1. Namen und Literale in HASKELL

A.6 Lexikon

Ein HASKELL-Programm ist eine Folge von Zeichen. Wir berücksichtigen hier nur den ACSII-Zeichensatz. Für Programme, die in *Unicode* verfasst werden, sind die vergrößern sich die Mengen der Zeichen, die als Zwischenräume, (Klein- und Groß-) Buchstaben, Ziffern und Symbole benutzt werden können.

A.6.1 Namen

Namen bezeichnen die Dinge (*entities*) eines Programms, wie Typen, Funktionen, Werte usw. Einige Namen sind für einen festgelegten Zweck als sogenannte "Schlüsselwörter" *reserviert*; andere bezeichnen Dinge, die in der Bibliothek *vordefiniert* sind und in jedem Programm benutzt werden können. Abgesehen von den Schlüsselwörtern kann jeder Name in einer Vereinbarung (*declaration*) eines Programm frei als Bezeichnung für ein Ding eingeführt werden. Dann sagt man, der Name sei an das Ding *gebunden*. Die Bindung eines Namens an ein Ding kann frühere Bindungen des Namens an andere Dinge unsichtbar machen (*überdecken*). Namen können auch gleichzeitig mehrere Dinge bezeichnen; solche Namen heißen *überladen* (*overloaded*). Dann muss bei jeder Benutzung des Namens aus dem Kontext ersichtlich ist, welche der Dinge hier bezeichnet werden soll.

Namen können mit dem Namen des Moduls *qualifiziert* werden, in dem sie definiert wurden: $M.n$ bezeichnet dann das Ding, die im Modul M an den Namen n gebunden wurde. Damit können Namenskonflikte vermieden werden, wenn Module die gleichen Namen binden.

An ihrer äußeren Form unterscheidet HASKELL zwei verschiedene Arten von Namen mit zwei bzw. drei Unterarten, die an sechs verschiedene Arten von Dinge gebunden werden können:

1. Ein *Bezeichner* (*identifier*, kurz *id*) beginnt mit einem Buchstaben, kann weitere Buchstaben, Ziffern und Unterstriche ("`_`") enthalten und mit Apostrophs "`'`" beendet werden.
   a) *Variablen-Bezeichner* beginnen mit einem Kleinbuchstaben; sie können (Wert-) Variablen und Typvariablen bezeichnen.
   b) *Konstruktor-Bezeichner* beginnen mit einem Großbuchstaben; sie können (Wert-) Konstruktoren, Typkonstruktoren, Klassen und Module bezeichnen.
2. *Symbole* bestehen aus den Sonderzeichen

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

   a) Die Symbole "`->`" "`[]`" und "`(,...,)`" (mit $k > 0$ Kommata) bezeichnen die vordefinierten Typ-Konstruktoren für Funktionsräume, Listen und Tupel (mit $k + 1$ Komponenten), die in Typausdrücken in Infix- bzw. Mixfix-Notation benutzt werden können (siehe Seite 216).

b) Symbole, die *nicht* mit einem Doppelpunkt beginnen, können zweistellige Funktionen bezeichnen, die als Infix-Operationen benutzt werden sollen.
c) Symbole, die mit einem Doppelpunkt ":" beginnen, können zweistellige (Wert-) Konstruktoren bezeichnen, die als Infix-Konstruktor-Operationen benutzt werden sollen.

A.6.2 Literale

...

B

Syntaktischer Zucker in Haskell

Funktionsdefinitionen

Bedingte Gleichungen entsprechen bedingten Ausdrücken.

$$
\begin{array}{rlll}
f\ p \mid & C_1 & = E_1 & \\
\mid & C_2 & = E_2 & \\
& \vdots & & \equiv \\
\mid & C_k & = E_k & \\
\mid & \mathit{otherwise} & = E_{k+1} &
\end{array}
\qquad
\begin{array}{ll}
f\ p = & \textbf{if} \quad C_1 \ \textbf{then}\ E_1 \\
& \textbf{else if}\ C_2 \ \textbf{then}\ E_2 \\
& \vdots \\
& \textbf{else if}\ C_k \ \textbf{then}\ E_k \\
& \textbf{else}\ E_{k+1}
\end{array}
$$

Mehrere Gleichungen mit Mustern entsprechen einer Gleichung mit einer Fallunterscheidung.

$$
\begin{array}{l}
f\ p_1 = E_1 \\
\quad \vdots \\
f\ p_k = E_k
\end{array}
\quad \equiv \quad
\begin{array}{ll}
f\ x = & \textbf{case}\ x\ \textbf{of} \\
& \quad p_1 \to E_1 \\
& \quad \vdots \\
& \quad p_k \to E_k
\end{array}
$$

Elementweise Definitionen von Funktionen entsprechen einer Funktionsdefinition mit einem Lambda-Ausdruck.

$$f\ x = E \quad \equiv \quad f = \lambda x \to E$$

Ausdrücke

Bedingte Ausdrücke sind Fallunterscheidungen über Wahrheitswerten.

$$
\begin{array}{ll}
\textbf{if} & C \\
\textbf{then} & E_1 \\
\textbf{else} & E_2
\end{array}
\quad \equiv \quad
\begin{array}{l}
\textbf{case}\ C\ \textbf{of} \\
\quad \mathrm{True} \to E_1 \\
\quad \mathrm{False} \to E_2
\end{array}
$$

Hilfsdefinitionen mit **let** und **where** entsprechen einander.

$$\begin{array}{lll} \textbf{let} & D_1 & \\ & \vdots & \equiv \\ & D_k & \\ \textbf{in} & E & \end{array} \qquad \begin{array}{ll} E & \\ \textbf{where} & D_1 \\ & \vdots \\ & D_k \end{array}$$

Operatoren können wie Funktionen benutzt werden und umgekehrt

$$\begin{array}{rcl} E_1 \otimes E_2 & \equiv & (\otimes)\ E_1\ E_2 \\ E_1\ `f`\ E_2 & \equiv & f\ E_1\ E_2 \end{array}$$

Listenumschreibungen sind Kombinationen der Funktionen *map* und *filter*.

$$[T \mid x \leftarrow G, C] \quad \equiv \quad map\ (\backslash x \rightarrow T)\ (filter\ (\backslash x \rightarrow C)\ G)$$

Listenausdrücke mit Ellipsen entsprechen vordefinierten Aufzählungsfunktionen.

$$\begin{array}{rcl} [u..] & \equiv & from\ u \\ [u..o] & \equiv & fromTo\ u\ o \\ [u, n..] & \equiv & fromBy\ u\ (n - u) \\ [u, n..o] & \equiv & fromByTo\ u\ (n - u)\ o \end{array}$$

Listenausdrücke sind geschachtelte Aufrufe des Listenkonstruktors.

$$[e_1, ..e_k] \quad \equiv \quad e_1\ : \cdots : e_k\ : [\,]$$

Zeichenkettenliterale sind Listen von Einzelzeichen.

$$['z_1', ..'z_k'] \quad \equiv \quad \text{ß}_1\ \cdots\ z_k\text{"}$$

Typ-Ausdrücke

Die Typkonstruktoren für Funnktionsräume, Listen und Tupel können infix bzw. mixfix benutzt werden.

$$\begin{array}{rcl} t \rightarrow s & \equiv & (\rightarrow)\ t\ s \\ [t] & \equiv & ([\,])\ t \\ (t_1, \ldots, t_k) & \equiv & ((, \ldots,)\ t_1\ \ldots\ t_2 \end{array}$$

C

TEXnischer Zucker in Haskell

In den Programmschnipseln dieses Buches werden mathematische Sonderzeichen und griechische Buchstaben für einige der vordefinierten Operationssymbole bzw. für Typvariablen verwendet.

Die Sonderzeichen werden übrigens mit dem LATEX-Paket `lstlisting` eingefügt, genauso wie die besondere Schreibweisen für Schlüsselwörter, Kommentare und lokale Varaiblen von Funktionsdefinitionen. In den TEX-Dateien nur Schnipsel steht also ganz "normaler" HASKELL-Text, der mit dem Filter "`unlit`" als lauffähiges HASKELL-Programm aus den Texten extrahiert werden kann.

D

Das Glasgower Haskell-System

D.0.3 Code in TEX

- Umgebung `code`
- Auskommentieren von Programmteilen, die nicht im formatierten Text erscheinen sollen.

D.0.4 Das LATEX-Paket `lstlisting`

E

Glossar

Bedingung Ein Ausdruck der einen Wahrheitswert liefert.
Currying
Datentyp Allgemein eine Wertemenge mit Operationen darauf
    Abstrakter _
    Algebraischer _
partielle Parametrisierung :
Typvariable (α, β, ...) Platzhalter für einen Typausdruck.
Variable (genauer: *Wertvariable*): getypter Platzhalter für einen (unveränderlichen) Wert
Wert : Element eines Datentyps

(Dies ist Fassung A.A.001 von 4. Februar 2010.)

Literaturverzeichnis

[Arm96] Joe Armstrong. Erlang – a survey of the language and its industrial application. In *Proc. 9'th Exhibition and Symposium on Industrial Applications of Prolog (*INAP'96*)*, Hino, Tokyo Japan, 1996.

[AS93] Harold Abelson and Gerald J. Sussman. *Structure and Interpretation of Computer Programs*. MIT Press, Cambridge, Massachusetts, 2. edition, 1993.

[AVL62] G. M. Adelson-Velskii and E. M. Landis. An algortihm for the organisation of information. *Soviet Math. Doklady (English translation)*, 3:1259–1263, 1962.

[Bac78] J. Backus. Can Programming be Liberated from the van-Neumann Style? *Comm. ACM*, 21(8):613–641, 1978.

[BE00] Jon Barwise and John Echtemendy. *Language, Proof, and Logic*. CSLI Press, Center for the Study of Language and Information, Stanford, CA, 2000.

[BvEvLP87] T. Brus, M.C.J.D. van Eekelen, M.O. van Leer, and M.J. Plasmeijer. Clean: A Language for Functional Graph Rewriting. In *Functional Programming Languages and Computer Architecture*, number 274 in Lecture Notes in Computer Science, pages 364–384. Springer, 1987.

[CF58] Haskell B. Curry and Robert Feys. *Combinatory Logic*, volume 1. North-Holland, Amsterdam, 1958.

[Chu41] Alonzo Church. *The Calculi of Lambda Conversion*. Princeton University Press, Princeton, New Jersey, 1941.

[CK04] Manuel M. T. Chakravarty and Gabriele C. Keller. *Einführung in die Programmierung mit* HASKELL. Pearson Studium, München, 2004.

[CM84] W.F. Clocksin and C.S. Melish. *Programming in Prolog*. Springer, 2. edition, 1984.

[Col78] Alain Colmerauer. Matamorphosis grammars. In L. Bolc, editor, *Natural Language Communication with Computers*, number 63 in Lecture Notes in Computer Science, pages 133–189. Springer, 1978.

[Cur06] Christian-Albrecht-Universität Kiel. CURRY – *An Integrated Functional Logic Language (Version 0.8.2)*, 2006. Link: `www.informatik.uni-kiel.de/~curry`.

[EMC+01] Hartmut Ehrig, Bernd Mahr, Felix Cornelius, Martin Große-Rhode, and Philip Zeitz. *Mathematisch-strukturelle Grundlagen der Informatik*. Springer-Verlag, 2 edition, 2001.

[Fre97] Gottlob Frege. *Begriffsschrift, eine der arithmetischen nachgebildete Formelsprache des reinen Denkens*. Olms: Hildesheim, Halle a. d. Saale, 1897. Nachdruck 1998.

[Fuc05] Norbert E. Fuchs. *Logische Programmierung*, chapter D 6, pages 613–632. In Rechenberg and Pomberger [RP05], 4. edition, 2005.

[HCSJ96] F. Henderson, T. Conway, Z. Somogyi, and D. Jeffery. The Mercury language reference manual. Technical Report 96/10, Department of Computer Science, University of Melbourne, Melbourne, Australia, 1996.

[Hud00] Paul Hudak. *The* Haskell *School of Expression: Learning Functional Programming through Multimedia*. Cambridge University Press, Cambridge;New York;Melbourne;Madrid, 2000.

[Hut07] Graham Hutton. *Programming in Haskell*. Cambridge University Press, 2007.

[Knu84] Donald E. Knuth. Literate programming. *Comput. J.*, 27(2):97–111, 1984.

[Lan66] P. J. Landin. The next 700 programming languages. *Comm. of the ACM*, 9(3):157–166, 1966.

[McC60] John McCarthy. Recursive functions of symbolic expressions and their computations by machine (part 1). *Comm. of the ACM*, 3(4):184–195, 1960.

[Moz06] The mozart programming environment. `www.mozart-oz.org`, 2006.

[MS06] Till Mossakowski and Lutz Schröder. Logik für Informatiker. `www.informatik.uni-bremen.de/~lschrode/teaching/Logic_e.htm`, Winter 2005/2006. Vorlesung, Universität Bremen.

[NM95] Ulf Nilsson and Jan Małuszýnski. *Logic, Programming, and* Prolog. John Wiley and Sons, Chichester, 2. edition, 1995. `www.ida.liu.se/~ulfni/lpp`.

[Ode05] Martin Odersky. *Funktionale Programmierung*, chapter D 5, pages 599–612. In Rechenberg and Pomberger [RP05], 4. edition, 2005.

[OGS08] Bryan O'Sullivan, John Goerzen, and Donald Stewart. *Real World Haskell*. O'Reilly, Sebastopol, CA, 2008.

[Pau91] Lawrence C. Paulson. *ML for the Working Programmer*. Cambridge University Press, 1991.

[Pep03] Peter Pepper. *Funktionale Programmierung in OPAL, ML, HASKELL und GOFER*. Springer, Berlin;Heidelberg;New York, 2 edition, 2003.

[PJ+02] Simon Peyton Jones et al. Haskell 98 language and library: The revised report, December 2002. Available via `http//www.haskell.org`.

[RL99] Fethi Rabhi and Guy Lapalme. *Algorithms – A Functional Programming Approach*. Addison-Wesley, Reading, Massachusetts, 1999.

[RP05] Peter Rechenberg and Gustav Pomberger, editors. *Informatik-Handbuch*. Hanser, 4. edition, 2005.

[Sch24] Moses Schönfinkel. Über die Bausteine der mathematischen Logik. *Mathematische Annalen*, 92:305, 1924.

[Sch00] Uwe Schöning. *Logik für Informatiker*. Spektrum Wissenschaftsverlag, Heidelberg Berlin, 5. edition, 2000.

[Smo08] Gert Smolka. *Programmierung – eine Einführung in die Informatik mit Standard ML*. Oldenbourg Wissenschaftsverlag, München, 2008.

[Tho99] Simon Thompson. *The Craft of Functional Programming*. International Computer Science Series. Addison Wesley, Reading, Massachusetts, 2 edition, 1999.

[Tur86] David .A. Turner. An Overview of Miranda. *SIGPLAN Notices*, 23(12):159–168, 1986.

[VRH04] Peter Van Roy and Seif Haridi. *Concepts, Techniques, and Models of Computer Programming*. MIT Press, 2004.

(Dies ist Fassung 1.2.02 von 4. Februar 2010.)