Grundlagen der Funktionalen Programmierung

Jürgen Giesl

Wintersemester 2002/2003

Lehr- und Forschungsgebiet Informatik II

RWTH Aachen

Inhaltsverzeichnis

0 Einleitung **4**

1 Einführung in die Sprache HASKELL **10**

1.1 Grundlegende Sprachkonstrukte 10
1.1.1 Deklarationen 10
1.1.2 Ausdrücke 22
1.1.3 Patterns 26
1.1.4 Typen 29
1.2 Funktionen höherer Ordnung 39
1.3 Programmieren mit Lazy Evaluation 50
1.4 Monaden 55
1.4.1 Ein- und Ausgabe mit Monaden 55
1.4.2 Programmieren mit Monaden 60

2 Semantik funktionaler Programme **70**

2.1 Vollständige Ordnungen und Fixpunkte 70
2.1.1 Partiell definierte Werte 71
2.1.2 Monotone und stetige Funktionen 74
2.1.3 Fixpunkte 83
2.2 Denotationelle Semantik von HASKELL 86
2.2.1 Konstruktion von Domains 87
2.2.2 Semantik einfacher HASKELL-Programme 92
2.2.3 Semantik komplexer HASKELL-Programme 98

3 Der Lambda-Kalkül **113**

3.1 Syntax des Lambda-Kalküls 114
3.2 Reduktionsregeln des Lambda-Kalküls 116
3.3 Reduzierung von HASKELL auf den Lambda-Kalkül 119
3.4 Der reine Lambda-Kalkül 126

4 Typüberprüfung und -inferenz **129**

4.1 Typschemata und Typannahmen 129
4.2 Der Typinferenzalgorithmus 131
4.2.1 Typinferenz bei Variablen und Konstanten 132
4.2.2 Typinferenz bei Lambda-Abstraktionen 132

4.2.3 Typinferenz bei Applikationen 136
4.2.4 Der gesamte Typinferenzalgorithmus 138
4.3 Typinferenz bei HASKELL-Programmen 140

Kapitel 0

Einleitung

Für Informatiker ist die Kenntnis verschiedener Familien von Programmiersprachen aus mehreren Gründen nötig:

- Die Vertrautheit mit unterschiedlichen Konzepten von Programmiersprachen ermöglicht es, eigene Ideen bei der Entwicklung von Software besser auszudrücken.
- Das Hintergrundwissen über verschiedene Programmiersprachen ist nötig, um in konkreten Projekten jeweils die am besten geeignete Sprache auszuwählen.
- Wenn man bereits einige konzeptionell verschiedene Programmiersprachen erlernt hat, ist es sehr leicht, sich später weitere Programmiersprachen schnell anzueignen.
- Es ist auch die Aufgabe von Informatikern, neue Programmiersprachen zu entwerfen. Dies kann nur auf der Grundlage der bereits entwickelten Sprachen geschehen.

Generell unterscheiden wir grundsätzlich zwischen *imperativen* und *deklarativen* Programmiersprachen *deklarative Programmiersprache* (wobei sich deklarative Sprachen weiter in funktionale und logische Sprachen unterteilen). In imperativen Sprachen setzen sich die Programme aus einer Folge von nacheinander ausgeführten Anweisungen zusammen, die die Werte der Variablen im Speicher verändern. Die meisten der heute verwendeten Programmiersprachen beruhen auf diesem Prinzip, das auch einen direkten Zusammenhang zu der klassischen Rechnerarchitektur besitzt, die auf John von Neumann zurückgeht.

In der deklarativen Programmierung bestehen die Programme hingegen aus einer Spezifikation dessen, *was* berechnet werden soll. Die Festlegung, *wie* die Berechnung genau verlaufen soll, wird dem Interpreter bzw. dem Compiler überlassen. Deklarative Programmiersprachen sind daher problemorientiert statt maschinenorientiert.

Einerseits sind die verschiedenen Programmiersprachen alle "gleichmächtig", d.h., jedes Programm lässt sich prinzipiell in jeder der üblicherweise verwendeten Sprachen schreiben. Andererseits sind die Sprachen aber unterschiedlich gut für verschiedene Anwendungsbereiche geeignet. So werden imperative Sprachen wie C beispielsweise für schnelle maschinennahe Programmierung eingesetzt, da dort der Programmierer direkt die Verwaltung des Speichers übernehmen kann (und muss). In anderen Programmiersprachen wird diese Aufgabe automatisch (vom Compiler) durchgeführt. Dies erlaubt eine schnellere Programmentwicklung, die weniger fehleranfällig ist. Andererseits sind die dabei entstehenden Programme meist auch weniger effizient (d.h., sie benötigen mehr Zeit und Speicherplatz). Funktionale Sprachen werden vor allem für die Prototypentwicklung, im Telekommunikationsbereich, aber auch für Multimediaanwendungen verwendet.

Das folgende Beispiel soll den Unterschied zwischen imperativen und funktionalen Programmiersprachen verdeutlichen. Wir benutzen hierfür zwei typische Sprachen, nämlich JAVA und HASKELL. Zur Illustration dieser Programmierparadigmen betrachten wir den Algorithmus zur Berechnung der Länge einer Liste. Die Eingabe des Algorithmus ist also eine Liste wie z.B. [15, 70, 36] und die Ausgabe des Algorithmus soll die Länge dieser Liste sein (in unserem Beispiel also 3). Ein imperativer Algorithmus zur Lösung dieser Aufgabe lässt sich leicht angeben.

```
class Element {

  Data    value;
  Element next;
}
```

```
class List {
  Element head;

  static int len (List l) {
    int n = 0;

    while (l.head != null) {
      l.head = l.head.next;
      n = n + 1;
    }
    return n;
  }
}
```

Anhand dieses Programms kann man die folgenden Beobachtungen machen:

- Das Programm besteht aus einzelnen Anweisungen, die nacheinander abgearbeitet werden. Hierzu existieren verschiedene Kontrollstrukturen (Bedingungen, Schleifen, etc.), um den Programmablauf zu steuern.

- Die Abarbeitung einer Anweisung ändert die Werte der Variablen im Speicher. Jede Anweisung kann daher Seiteneffekte auslösen. Beispielsweise ändert sich im Verlauf der Ausführung von `len` sowohl der Wert von `n` als auch von `l`. Letzteres hat auch außerhalb der Methode `len` Auswirkungen. Bei der Berechnung von `len(l)` wird auch der Wert des Objekts, auf das `l` zeigt, geändert. Man erhält also nicht nur einen `int`-Wert als Ergebnis, sondern als *Seiteneffekt* wird `l` dabei verändert (zum Schluss ist `l.head = null`), d.h., die Liste wird beim Berechnen der Länge geleert.

- Der Programmierer muss sich Gedanken über die Realisierung und die Speicherverwaltung bei nicht-primitiven Datentypen (wie Listen) machen. Beispielsweise ist der obige Seiteneffekt evtl. nicht gewollt. Um dies zu vermeiden, muss der Programmierer aber erwünschte und unerwünschte Seiteneffekte voraussehen. Die Seiteneffekte und die explizite Speicherverwaltung in imperativen Programmen führen daher oft zu schwer lokalisierbaren Fehlern.

Nachdem wir die Konzepte des imperativen Programmierens illustriert haben, kommen wir nun zu deklarativen Programmiersprachen, bei denen das Programm beschreibt, was berechnet werden soll, aber die genaue Festlegung, wie die Berechnung verlaufen soll, dem Compiler oder Interpreter überlässt. Unser Ziel ist wieder, die Länge einer Liste zu berechnen. Die folgende Beschreibung macht deutlich, was die Länge `len` einer Liste `l` bedeutet:

(A) Falls die Liste `l` leer ist, so ist `len(l) = 0`.

(B) Falls die Liste `l` nicht leer ist und "`xs`" die Liste `l` ohne ihr erstes Element ist, so ist `len(l) = 1 + len(xs)`.

Wir schreiben im folgenden `x:xs` für die Liste, die aus der Liste `xs` entsteht, indem man das Element (bzw. den Wert) `x` vorne einfügt. Wir haben also `15:[70,36] = [15,70,36]`. Jede nicht-leere Liste kann man daher als `x:xs` darstellen, wobei `x` das erste Element der Liste ist und `xs` die verbleibende Liste ohne ihr erstes Element. Wenn wir die leere Liste mit `[]` bezeichnen, so haben wir `15:70:36:[] = [15,70,36]` (wobei das Einfügen mit "`:`" von rechts nach links abgearbeitet wird, d.h., "`:`" assoziiert nach rechts).

Nun lässt sich die obige Spezifikation (Beschreibung) der Funktion `len` direkt in ein funktionales Programm übersetzen. Die Implementierung in der funktionalen Programmiersprache HASKELL lautet wie folgt:

```
len :: [data] -> Int
len []     = 0
len (x:xs) = 1 + len xs
```

Ein funktionales Programm ist eine Menge von Deklarationen. Eine Deklaration bindet eine Variable (wie `len`) an einen Wert (wie die Funktion, die die Länge von Listen berechnet). Die Zeile `len :: [data] -> Int` ist eine *Typdeklaration*, die besagt, dass `len` eine Liste als Eingabe erwartet und eine ganze Zahl als Ausgabe berechnet. Hierbei bezeichnet `data` den Typ der Elemente der Liste. Die Datenstruktur der Listen ist in HASKELL vordefiniert (aber natürlich gibt es die Möglichkeit, weitere Datenstrukturen selbst zu definieren).

Es folgen die definierenden Gleichungen von `len`. Die erste Gleichung gibt an, was das Resultat der Funktion `len` ist, falls `len` auf die leere Liste `[]` angewendet wird. (In HASKELL sind Klammern um das Argument einer Funktion nicht nötig, sie können jedoch auch geschrieben werden. Man könnte also für die erste definierende Gleichung auch `len([]) = 0` und für die zweite Gleichung `len(x:xs) = 1 + len(xs)` schreiben.) Die zweite Gleichung ist anwendbar, wenn das Argument von `len` eine nicht-leere Liste ist. Das Argument hat dann die Form `x:xs`. In diesem Fall wird nun `1 + len xs` berechnet. Man erkennt, dass `len` *rekursiv* definiert ist, d.h., zur Berechnung von `len(x:xs)` muss wiederum len (von einem anderen Argument, nämlich `xs`) berechnet werden.

Die Ausführung eines funktionalen Programms besteht in der Auswertung eines Ausdrucks mit Hilfe dieser Funktionen. In unserem Beispiel wird lediglich die Funktion `len` definiert. Um die Arbeitsweise des obigen Algorithmus zu veranschaulichen, betrachten wir die Berechnung von `len [15,70,36]`. Bei Ausführung des Algorithmus wird zunächst überprüft, ob das Argument die leere Liste ist (d.h., ob das Argument `[]` in der ersten definierenden Gleichung auf das aktuelle Argument `[15,70,36]` passt). Diesen Vorgang bezeichnet man als *Pattern Matching*. Da dies nicht der Fall ist, versucht man nun, die zweite definierende Gleichung anzuwenden. Dies ist möglich, wobei in unserem Beispiel das erste Listenelement `x` der Zahl `15` entspricht und die Restliste `xs` der Liste `[70,36]`. Um `len [15,70,36]` zu berechnen, muss man also `1 + len [70,36]` bestimmen. Da das neue Argument `[70,36]` wieder eine nicht-leere Liste mit dem `x`-Wert `70` und dem `xs`-Wert `[36]` ist, führt dies zu einem erneuten Aufruf von `len` mit dem Argument `[36]`. Schließlich ergibt sich `1 + 1 + 1 + 0` = `3`.

Die wichtigsten Eigenschaften der funktionalen Programmiersprache HASKELL lassen sich wie folgt zusammenfassen:

- **Keine Schleifen**
  In (rein) funktionalen Sprachen gibt es keine Kontrollstrukturen wie Schleifen, sondern stattdessen wird nur Rekursion verwendet.

- **Polymorphes Typsystem**
  Funktionale Programmiersprachen erlauben es üblicherweise, dass eine Funktion wie `len` für Listen von Elementen beliebiger Typen verwendet werden kann. Man bezeichnet dies als *parametrischen* Polymorphismus. Für die Variable `data` im Typ von `len` kann also ein beliebiger Typ von Elementen eingesetzt werden (`data` ist eine *Typvariable*), d.h., `len` arbeitet unabhängig vom Typ der Elemente. Trotzdem garantiert das Typsystem, dass Daten nicht falsch interpretiert werden.

  Durch parametrischen Polymorphismus ergibt sich eine Reduzierung des Programmieraufwands, da entsprechende Funktionen nicht immer wieder neu geschrieben werden müssen.

- **Keine Seiteneffekte**
  Die Reihenfolge der Berechnungen beeinflusst das Ergebnis des Programms nicht. Insbesondere haben Programme keine Seiteneffekte, d.h., der Wert der Parameter ändert sich nicht bei der Ausführung einer Funktion. Das Ergebnis einer Funktion hängt also nur von den Argumenten der Funktion ab und wird eine Funktion mehrmals auf dieselben Argumente angewendet, so ist das Ergebnis immer dasselbe. Dieses Verhalten bezeichnet man als *referentielle Transparenz*.

- **Automatische Speicherverwaltung**
  Explizite Zeigermanipulation sowie Anforderung oder Freigabe des Speicherplatzes werden nicht vom Programmierer durchgeführt.

- **Gleichberechtigte Behandlung von Funktionen als Datenobjekte**
  Funktionen werden als gleichberechtigte Datenobjekte behandelt. Insbesondere können Funktionen also auch Argumente oder Ergebnisse anderer Funktionen sein. Solche Funktionen heißen Funktionen höherer Ordnung.

- **Lazy Evaluation** (Verzögerte Auswertung)
  Zur Auswertung eines Funktionsaufrufs werden nur diejenigen Teile der Argumente ausgewertet, die notwendig für die Berechnung des Ergebnisses sind. (Diese Auswertungsstrategie wird allerdings nicht in allen funktionalen Sprachen verwendet. Beispielsweise arbeiten Sprachen wie ML oder LISP und SCHEME mit sogenannter strik-

ter Auswertungsstrategie, bei der alle Argumente einer Funktion ausgewertet werden müssen, bevor die Funktion selbst angewendet werden kann.)

Insgesamt ergeben sich folgende wichtige Vorteile der funktionalen Programmierung:

- Die Programme sind kürzer, klarer und besser zu warten.
- Bei der Programmierung geschehen weniger Fehler, so dass zuverlässigere Programme entstehen. Funktionale Sprachen haben üblicherweise auch eine klare mathematische Basis und sind besser zur Verifikation geeignet als imperative Programmiersprachen.
- Die Programmentwicklung ist schneller und einfacher. (Dies liegt auch daran, dass der Programmierer sich wesentlich weniger mit der Speicherorganisation befassen muss als in imperativen Sprachen). Beispielsweise hat die Firma *Ericsson* nach Einführung der funktionalen Sprache ERLANG eine um den Faktor 10 – 25 schnellere Programmentwicklungszeit gemessen. Die entstehenden Programme waren um den Faktor 2 – 10 kürzer als zuvor. Insbesondere zeigt dies, dass funktionale Sprachen ideal zur Prototypentwicklung sind. Bei sehr zeitkritischen (Realzeit-)Anwendungen sind aus Effizienzgründen maschinennahe Sprachen oft geeigneter.
- Funktionale Programme sind oft besser wiederzuverwenden und modularer strukturiert.

Der Aufbau der Vorlesung ist wie folgt: In Kapitel 1 wird eine Einführung in die funktionale Programmiersprache HASKELL gegeben. Hierdurch werden die Möglichkeiten funktionaler Sprachen deutlich und man gewinnt einen Eindruck von einer realen funktionalen Programmiersprache.

In den folgenden Kapiteln gehen wir auf die Konzepte und Techniken ein, die funktionalen Programmiersprachen zugrunde liegen, wobei wir uns dabei wieder auf HASKELL beziehen. Im Kapitel 2 zeigen wir, wie man die Semantik solcher Sprachen definieren kann. Hierunter versteht man eine formale Festlegung, welche Bedeutung ein funktionales Programm bzw. ein Ausdruck in einem funktionalen Programm hat (d.h., welches Ergebnis dadurch berechnet wird). Solch eine Festlegung ist nötig, um die Korrektheit von Programmen bestimmen zu können und um zu definieren, was die Konstrukte der Programmiersprache bedeuten. Insbesondere ist es also die Grundlage für jede Implementierung der Sprache, da nur dadurch festgelegt werden kann, wie Interpreter oder Compiler arbeiten sollen.

Anschließend führen wir in Kapitel 3 den sogenannten Lambda-Kalkül ein. Dies ist die Grundsprache, die allen funktionalen Programmiersprachen zugrunde liegt. Diese Programmiersprachen sind lediglich lesbarere Versionen des Lambda-Kalküls. Wir zeigen daher, wie HASKELL auf den Lambda-Kalkül zurückgeführt werden kann. Der Lambda-Kalkül stellt insbesondere eine Möglichkeit dar, um funktionale Programme zu implementieren und auf bestimmte Korrektheitseigenschaften zu überprüfen.

Hierzu stellen wir in Kapitel 4 ein Verfahren vor, das untersucht, ob ein Programm (bzw. der entsprechende Ausdruck im Lambda-Kalkül) korrekt getypt ist. Mit anderen Worten, wir überprüfen, ob Funktionen immer nur auf Argumente der richtigen Sorte angewendet werden. Diese Überprüfung wird in Interpretern oder Compilern als erster Schritt vor der Ausführung eines funktionalen Programms durchgeführt. Wie erwähnt, besitzt HASKELL ein polymorphes Typsystem. Dadurch kann man z.B. die Berechnung der Länge von Listen von Zahlen, von Listen von Zeichen, von Listen von Listen etc. mit ein- und derselben Funktion `len` realisieren, was ein hohes Maß an Wiederverwendbarkeit erlaubt. Andererseits wird aus diesem Grund die Typprüfung nicht-trivial.

Ich danke meinen Mitarbeitern Dipl.-Inform. René Thiemann und Dipl.-Inform. Darius Dlugosz für ihre konstruktiven Kommentare und Vorschläge beim Korrekturlesen des Skripts.

Kapitel 1

Einführung in die funktionale Programmiersprache HASKELL

In diesem Kapitel geben wir eine Einführung in die Sprache HASKELL. Hierbei werden wir die Syntax der Sprache vorstellen und die Bedeutung der Sprachkonstrukte informell erklären. Eine formale Definition der Semantik der Sprache folgt in Kapitel 2. Für weitere Beschreibungen der Sprache HASKELL sei auf [Thi94, Bir98, PJH98, Tho99, HPF00, Hud00, PJ00, Pep02] verwiesen. Weitere Informationen zu HASKELL findet man auf der HASKELL-Homepage (`http://www.haskell.org`). Der HASKELL-Interpreter HUGS ist auf der Seite `http://www.haskell.org/hugs` erhältlich.

Wir stellen zunächst in Abschnitt 1.1 die grundlegenden Sprachkonstrukte von HASKELL vor. Anschließend gehen wir auf funktionale Programmiertechniken ein. Hierzu betrachten wir in Abschnitt 1.2 Funktionen höherer Ordnung, d.h. Funktionen, die wiederum Funktionen verarbeiten. In Abschnitt 1.3 zeigen wir, wie man mit Lazy Evaluation programmiert und dabei unendliche Datenobjekte verwenden kann. Schließlich gehen wir in Abschnitt 1.4 auf das Konzept der Monaden ein, die insbesondere zur Ein- und Ausgabe in HASKELL verwendet werden.

1.1 Grundlegende Sprachkonstrukte

In diesem Abschnitt führen wir die grundlegenden Sprachkonstrukte von HASKELL (Deklarationen, Ausdrücke, Patterns und Typen) ein.

1.1.1 Deklarationen

Ein Programm in HASKELL ist eine Folge von Deklarationen. Die Deklarationen müssen linksbündig untereinander stehen. Der Grund dafür wird später klar, wenn wir lokale Deklarationen betrachten, die eingerückt (bzw. in der gleichen Zeile) stehen.

Eine *Deklaration* ist (im einfachsten Fall) eine Beschreibung einer Funktion. Funktionen sind gekennzeichnet durch ein Funktionssymbol (den Namen der Funktion), den Definitionsbereich, den Wertebereich des Resultats und eine Abbildungsvorschrift. Den Definitionsbereich und den Wertebereich legt man in einer sogenannten *Typdeklaration* fest und die Abbildungsvorschrift wird in einer *Funktionsdeklaration* beschrieben. Die Syntax von Deklarationen ist daher durch folgende kontextfreie Grammatik gegeben.

$$\underline{\mathsf{decl}} \;\rightarrow\; \underline{\mathsf{typedecl}} \mid \underline{\mathsf{fundecl}}$$

Im folgenden werden wir Nichtterminalsymbole immer durch Unterstreichung kenntlich machen. Außerdem werden wir mit einer Teilmenge von HASKELL beginnen und die Grammatikregeln sukzessive erweitern. (Einige in der HASKELL-Syntax erlaubte Programme werden wir nicht berücksichtigen — die komplette Grammatik für HASKELL-Programme findet sich in [PJH98].)

Als *Kommentare* werden in HASKELL Texte betrachtet, die zwischen `{-` und `-}` eingeschlossen sind sowie jeglicher Text zwischen `--` und dem Zeilenende.

Typdeklarationen

Als Funktionssymbole dienen in HASKELL Variablenbezeichner. Für eine Funktion zur Quadrierung von Zahlen kann z.B. der Name `square` verwendet werden. Eine Deklaration bindet eine Variable (wie `square`) an einen Wert (wie die Funktion, die Zahlen quadriert). Dann kann man die folgende Typdeklaration für `square` angeben.

```
square :: Int -> Int
```

Das erste `Int` beschreibt den Definitionsbereich und das zweite `Int` beschreibt den Wertebereich von `square`. Der Typ `Int` ist dabei in HASKELL vordefiniert. Die Deklaration $\underline{\mathsf{var}}$ `::` $\underline{\mathsf{type}}$ bedeutet, dass die Variable $\underline{\mathsf{var}}$ den Typ $\underline{\mathsf{type}}$ hat. Mit Hilfe von "`->`" wird ein Funktionstyp definiert (d.h., `Int -> Int` ist der Typ der Funktionen, die ganze Zahlen in ganze Zahlen abbilden). Als weiteres Beispiel beschreibt `[Int]` den Typ der Listen von ganzen Zahlen. Im allgemeinen existiert zu jedem Typ `a` der Typ `[a]` der Listen mit Elementen vom Typ `a`.

Man erhält die folgende Grammatikregel für die Syntax von Typdeklarationen. Hierbei legt eine Typdeklaration den Typ von einer oder mehreren Variablen fest.

$$\underline{\mathsf{typedecl}} \;\rightarrow\; \underline{\mathsf{var}}_1, \ldots, \underline{\mathsf{var}}_n \; :: \underline{\mathsf{type}}, \;\text{ wobei } n \geq 1$$

Typdeklarationen müssen nicht mit angegeben werden. Sie werden dann durch den Interpreter oder Compiler automatisch berechnet. Allerdings sind Typdeklarationen vorteilhaft für die Verständlichkeit von Programmen und sollten daher normalerweise verwendet werden. Diese Deklarationen werden dann vom Interpreter oder Compiler überprüft.

Variablenbezeichner $\underline{\mathsf{var}}$ sind beliebige Folgen von Buchstaben und Zahlen (Strings), die mit einem Kleinbuchstaben beginnen (wie z.B. `square`).

Funktionsdeklarationen

Nach der Typdeklaration folgen die definierenden Gleichungen, d.h. die Abbildungsvorschrift. Beispielsweise könnte die Funktionsdeklaration für `square` wie folgt lauten.

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

Die linke Seite einer definierenden Gleichung besteht aus dem Namen der Funktion und der Beschreibung des Arguments und die rechte Seite definiert das Ergebnis der Funktion. Hierbei müssen die Typen der Argumente und der Ergebnisse natürlich zum Typ der Funktion "passen" (d.h., `square` darf sowohl als Argument wie als Ergebnis nur Ausdrücke vom Typ `Int` bekommen). Arithmetische Grundoperationen wie `+`, `*`, `-`, etc. sowie Vergleichsoperationen wie `==` (für die Gleichheit), `>=`, etc. sind in HASKELL vordefiniert. Ebenso ist auch die Datenstruktur `Bool` mit den Werten `True` und `False` und den Funktionen `not`, `&&` und `||` vordefiniert. Zur Definition solcher häufig verwendeter Funktionen dienen Bibliotheken. Die oben erwähnten Funktionen sind in einer Standardbibliothek (dem sogenannten "Prelude") definiert, das bei jedem Start von HASKELL geladen wird. Allgemein werden Funktionsdeklarationen wie folgt aufgebaut.

$$\begin{array}{lcl} \underline{\mathsf{fundecl}} & \to & \underline{\mathsf{funlhs}}\ \underline{\mathsf{rhs}} \\ \underline{\mathsf{funlhs}} & \to & \underline{\mathsf{var}}\ \underline{\mathsf{pat}} \\ \underline{\mathsf{rhs}} & \to & =\ \underline{\mathsf{exp}} \end{array}$$

Hierbei steht $\underline{\mathsf{var}}$ für den Funktionsnamen (wie `square`) und $\underline{\mathsf{pat}}$ für das Argument auf der linken Seite der definierenden Gleichung (wie z.B. `x`). Wie solche Argumente im allgemeinen Fall aussehen dürfen, wird in Abschnitt 1.1.3 erläutert. Die rechte Seite einer definierenden Gleichung ist ein beliebiger Ausdruck $\underline{\mathsf{exp}}$ (wie z.B. `x * x`).

Ausführung eines funktionalen Programms

Die Ausführung eines Programms besteht aus der Auswertung von Ausdrücken. Dies geschieht ähnlich wie bei einem Taschenrechner: Der Benutzer gibt (bei einem Interpreter) einen Ausdruck ein und der Rechner wertet ihn aus. Gibt man beispielsweise `42` ein, so wird auch als Ergebnis `42` zurückgegeben. Gibt man `6 * 7` ein, so wird ebenfalls `42` zurückgegeben, denn die Operation `*` ist vordefiniert. Aber auf dieselbe Art und Weise werden auch die benutzerdefinierten Deklarationen für die Auswertung verwendet. Bei der Eingabe von `square 11` erhält man also das Ergebnis `121` und dasselbe Resultat ergibt sich bei der Eingabe von `square (12 - 1)`. Die Bindungspriorität der Funktionsanwendung ist hierbei am höchsten, d.h., bei der Eingabe von `square 12 - 1` erhält man `143`.

Die Auswertung eines Ausdrucks erfolgt durch *Termersetzung* in zwei Schritten:

(1) Der Computer sucht einen Teilausdruck, der mit der linken Seite einer definierenden Gleichung übereinstimmt, wobei hierbei die Variablen der linken Seite durch geeignete Ausdrücke ersetzt werden müssen. Solch einen Teilausdruck bezeichnet man als *Redex* (für "reducible expression").

(2) Der Redex wird durch die rechte Seite der definierenden Gleichung ersetzt, wobei die Variablen in der rechten Seite genauso wie in (1) belegt werden müssen.

Diese Auswertungsschritte werden solange wiederholt, bis kein Ersetzungsschritt mehr möglich ist.

In Abb. 1.1 sind alle Möglichkeiten zur Auswertung des Ausdrucks `square (12 - 1)` dargestellt. Jeder Pfad durch das Diagramm entspricht einer möglichen Folge von Auswertungsschritten. Eine *Auswertungsstrategie* ist ein Algorithmus zur Auswahl des nächsten Redex.

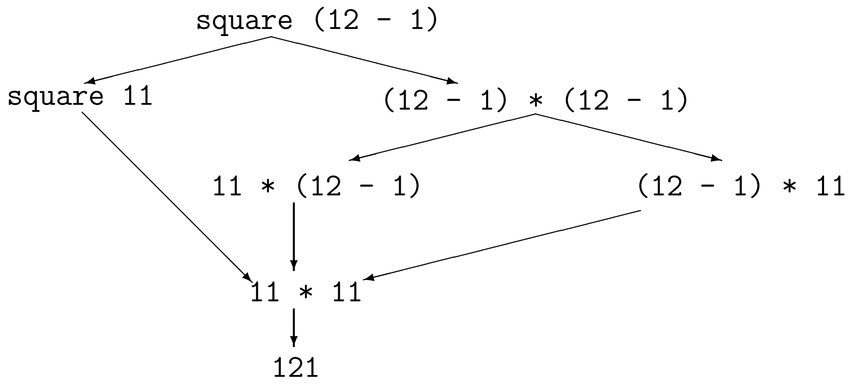

Abbildung 1.1: Auswertung eines Ausdrucks

Insbesondere unterscheiden wir zwischen *strikter* und *nicht-strikter* Auswertung. Bei strikter Auswertung wird stets der am weitesten innen links im Ausdruck vorkommende Redex gewählt. Dies entspricht dem linkesten Pfad durch das Diagramm in Abb. 1.1. Diese Strategie bezeichnet man auch als *leftmost innermost* oder *call-by-value* Strategie oder als *eager evaluation*.

Bei der nicht-strikten Auswertung wird der am weitesten außen links im Ausdruck auftretende Redex gewählt. Die Argumente von Funktionen sind nun in der Regel unausgewertete Ausdrücke. Dies entspricht dem mittleren Pfad durch das Diagramm in Abb. 1.1. Diese Strategie wird auch als *leftmost outermost* oder *call-by-name* Strategie bezeichnet.

Beide Strategien haben Vor- und Nachteile. Bei der nicht-strikten Auswertung werden nur die Teilausdrücke ausgewertet, deren Wert zum Endergebnis beiträgt, was bei der strikten Auswertung nicht der Fall ist. Andererseits muss die nicht-strikte Strategie manchmal denselben Wert mehrfach auswerten, obwohl dies in der strikten Strategie nicht nötig ist (dies geschieht hier mit dem Teilausdruck `12 - 1`).

HASKELL verfolgt das Prinzip der sogenannten *Lazy Evaluation* (verzögerte Auswertung), das beide Vorteile zu kombinieren versucht. Hierbei wird die nicht-strikte Auswertung verfolgt, jedoch werden doppelte Teilausdrücke nicht doppelt ausgewertet, wenn sie aus dem gleichen Ursprungsterm entstanden sind. Im obigen Beispiel würde der Teilterm $12 - 1$ z.B. durch einen Zeiger auf die gleiche Speicherzelle realisiert werden und damit nur einmal ausgewertet werden.

Beim Vergleich der Auswertungsstrategien erhält man das folgende wichtige Resultat: Wenn irgendeine Auswertungsstrategie terminiert, so terminiert auch die nicht-strikte Auswertung (aber nicht unbedingt die strikte Auswertung). Außerdem gilt für alle Strategien: Wenn die Berechnung endet, dann ist das Ergebnis unabhängig von der Strategie gleich. Die Strategien haben also nur Einfluss auf das Terminierungsverhalten, aber nicht auf das Ergebnis. Als Beispiel betrachten wir die folgenden Funktionen.

```
three :: Int -> Int        non_term :: Int -> Int
three x = 3                non_term x = non_term (x+1)
```

Die Auswertung der Funktion `non_term` terminiert für kein Argument. Die strikte Auswertung des Ausdrucks `three (non_term 0)` würde daher ebenfalls nicht terminieren. In

HASKELL wird dieser Ausdruck hingegen zum Ergebnis `3` ausgewertet. Weitere Vorteile der nicht-strikten Strategie werden wir später in Abschnitt 1.3 kennen lernen.

Bedingte definierende Gleichungen

Natürlich will man auch mehrstellige Funktionen und bedingte definierende Gleichungen verwenden. Hierzu betrachten wir eine Funktion `maxi` mit folgender Typdeklaration.

```
maxi :: (Int, Int) -> Int
```

Hierbei bezeichnet `(Int, Int)` das kartesische Produkt der Typen `Int` und `Int` (dies entspricht also der mathematischen Notation `Int` × `Int`). `(Int, Int) -> Int` ist demnach der Typ der Funktionen, die Paare von ganzen Zahlen auf ganze Zahlen abbilden. Die Funktionsdeklaration von `maxi` lautet wie folgt.

```
maxi(x, y) | x >= y     = x
           | otherwise  = y
```

Der Ausdruck auf der rechten Seite einer definierenden Gleichung kann also durch eine Bedingung (d.h. einen Ausdruck vom Typ `Bool`) eingeschränkt werden. Zur Auswertung verwendet man dann die erste Gleichung, deren Bedingung erfüllt ist (die Fallunterscheidung in den Gleichungen muss aber nicht vollständig sein). Der Ausdruck `otherwise` ist eine vordefinierte Funktion, die immer `True` liefert. Also muss die Grammatikregel für die Bildung von rechten Seiten $\underline{\mathsf{rhs}}$ definierender Gleichungen nun wie folgt geändert werden:

$$\begin{array}{lll} \underline{\mathsf{rhs}} & \rightarrow & = \underline{\mathsf{exp}} \mid \underline{\mathsf{condrhs}}_1 \ldots \ \underline{\mathsf{condrhs}}_n, \quad \text{wobei } n \geq 1 \\ \underline{\mathsf{condrhs}} & \rightarrow & \mid \underline{\mathsf{exp}} = \underline{\mathsf{exp}} \end{array}$$

Currying

Um die Anzahl der Klammern in Ausdrücken zu reduzieren (und damit die Lesbarkeit zu verbessern), ersetzt man oftmals Tupel von Argumenten durch eine Folge von Argumenten. Diese Technik ist nach dem Logiker *Haskell B. Curry* benannt, dessen Vorname bereits für den Namen der Programmiersprache HASKELL benutzt wurde. Betrachten wir zur Illustration zunächst eine konventionelle Definition der Funktion `plus`.

```
plus :: (Int, Int) -> Int
plus (x, y) = x + y
```

Stattdessen könnte man nun folgende Definition verwenden:

```
plus :: Int -> (Int -> Int)
plus x y = x + y
```

Eine Überführung der ersten Definition von `plus` in die zweite bezeichnet man als *Currying*. Für den Typ `Int -> (Int -> Int)` könnte man auch einfacher `Int -> Int -> Int` schreiben, denn wir benutzen die Konvention, dass der Funktionsraumkonstruktor `->` nach rechts assoziiert. Die Funktionsanwendung hingegen assoziiert nach links, d.h., der Ausdruck `plus 2 3` steht für `(plus 2) 3`.

Jetzt bekommt `plus` nacheinander zwei Argumente. Genauer ist `plus` nun eine Funktion, die eine ganze Zahl `x` als Eingabe erhält. Das Ergebnis ist dann die Funktion `plus x`. Dies ist eine Funktion von `Int` nach `Int`, wobei `(plus x) y` die Addition von `x` und `y` berechnet.

Solche Funktionen können also auch mit nur einem Argument aufgerufen werden (dies bezeichnet man auch als *partielle Anwendung*). Die Funktion `plus 1` ist z.B. die Nachfolgerfunktion, die Zahlen um `1` erhöht und `plus 0` ist die Identitätsfunktion auf ganzen Zahlen. Diese Möglichkeit der Anwendung auf eine geringere Zahl an Argumenten ist (neben der Klammerersparnis) der zweite Vorteil des Currying. Insgesamt ändert sich also die Grammatikregel für linke Seiten definierender Gleichungen wie folgt:

$$\underline{\mathsf{funlhs}} \;\rightarrow\; \underline{\mathsf{var}}\; \underline{\mathsf{pat}}_1 \ldots \underline{\mathsf{pat}}_n, \quad \text{wobei } n \geq 1$$

Funktionsdefinition durch Pattern Matching

Die Argumente auf der linken Seite einer definierenden Gleichung müssen im allgemeinen keine Variablen sein, sondern sie dürfen beliebige *Patterns* (Muster) sein, die als Muster für den erwarteten Wert dienen. Betrachten wir hierzu die Funktion `und`, die die Konjunktion boolescher Werte berechnet.

```
und :: Bool -> Bool -> Bool
und True  y = y
und False y = False
```

Insbesondere haben wir also jetzt mehrere Funktionsdeklarationen (d.h. definierende Gleichungen) für dasselbe Funktionssymbol.

Hierbei sind `True` und `False` vordefinierte Datenkonstruktoren des Datentyps `Bool`, d.h., sie dienen zum Aufbau der Objekte dieses Datentyps. Konstruktoren beginnen in HASKELL immer mit Großbuchstaben.

Um bei einem Funktionsaufruf `und` $\underline{\mathsf{exp}}_1$ $\underline{\mathsf{exp}}_2$ festzustellen, welche definierende Gleichung anzuwenden ist, testet man der Reihe nach von oben nach unten, welche Patterns zu den aktuellen Argumenten $\underline{\mathsf{exp}}_1$ und $\underline{\mathsf{exp}}_2$ passen (Matching). Die Frage ist also, ob es eine Substitution gibt, die die Variablen der Patterns durch konkrete Ausdrücke ersetzt, so dass dadurch die instantiierten Patterns mit $\underline{\mathsf{exp}}_1$ und $\underline{\mathsf{exp}}_2$ übereinstimmen. In diesem Fall sagt man auch, dass der Pattern $\underline{\mathsf{pat}}_i$ (auf) den Ausdruck $\underline{\mathsf{exp}}_i$ "matcht". Dann wird der Gesamtausdruck zu der entsprechend instantiierten rechten Seite ausgewertet. Beispielsweise wird also `und True True` zu `True` ausgewertet, da bei der Substitution [`y/True`] die Patterns `True` und `y` der ersten definierenden Gleichung mit den aktuellen Argumenten `True` und `True` übereinstimmen.

Da Patterns von oben nach unten ausgewertet werden, ist die Definition von `und` äquivalent zur folgenden alternativen Deklaration.

```
und :: Bool -> Bool -> Bool
und True y = y
und x    y = False
```

Wenn wir eine Funktion

```
unclear :: Int -> Bool
unclear x = not (unclear x)
```

haben, deren Auswertung nicht terminiert, so terminiert die Auswertung von `und False (unclear 0)` dennoch, denn um das Pattern Matching durchzuführen, muss `unclear 0` nicht ausgewertet werden. Hingegen terminieren `und (unclear 0) False` oder `und True (unclear 0)` nicht.

In der Funktion `und` gelingt das Pattern Matching, weil ein Wert vom Typ `Bool` nur mit den Datenkonstruktoren `True` oder `False` gebildet werden kann. Boolesche Werte werden also anhand der folgenden Regel konstruiert.[1]

$$\underline{\texttt{Bool}} \;\rightarrow\; \texttt{True} \mid \texttt{False}$$

Pattern Matching ist jedoch auch bei anderen Datentypen möglich. Um zu zeigen, wie man Pattern Matching bei Listen verwenden kann, betrachten wir wieder den Algorithmus `len`.

```
len :: [data]  -> Int
len []       = 0
len (x : xs) = 1 + len xs
```

Die vordefinierte Datenstruktur der Listen hat die Datenkonstruktoren `[]` und `:`, so dass Listen wie folgt gebildet werden:

$$\underline{\texttt{[data]}} \;\rightarrow\; \texttt{[]} \mid \underline{\texttt{data}} : \underline{\texttt{[data]}}$$

Hierbei steht `[]` für die leere Liste und der (Infix-)Konstruktor "`:`" dient zum Aufbau von nicht-leeren Listen. Wie erwähnt steht der Ausdruck `x:xs` für die Liste `xs`, in der vorne das Element `x` eingefügt wurde. Das Element `x` hat hierbei einen Typ `data` und `xs` ist eine Liste von Elementen des Typs `data`. (Die Grammatik gibt also an, wie Listen vom Typ `[data]` gebildet werden.)

Bei der Auswertung von `len [15,70,36]` wird zunächst die Listenkurzschreibweise aufgelöst. Das Argument von `len` ist also `15:(70:(36:[]))`. Nun wird Pattern Matching beginnend mit der ersten definierenden Gleichung durchgeführt. Der erste Datenkonstruktor `[]` passt nicht auf den Konstruktor "`:`", mit dem das aktuelle Argument gebildet ist. Aber der Pattern der zweiten definierenden Gleichung passt auf diesen Wert, wobei die Substitution [`x/15, xs/70:(36:[])`] verwendet wird. Also wertet dieser Ausdruck im ersten Schritt zu `1 + len (70:(36:[]))` aus, etc.

Analog dazu könnte man auch folgenden Algorithmus definieren:

```
second :: [Int]  -> Int
second []          = 0
second (x : [])    = 0
second (x : y : xs) = y
```

[1] Die Grammatikregeln für <u>`Bool`</u>, <u>`[data]`</u> und <u>`Int`</u> dienen hier nur zur Illustration des Pattern Matchings und sind nicht Teil der HASKELL-Sprachdefinition.

Man darf auch die Listenkurzschreibweise in diesen Patterns verwenden und die zweite Gleichung durch `second [x] = 0` ersetzen.

Pattern Matching ist auch für `Int` möglich, wie das folgende Beispiel zur Berechnung der Fakultät zeigt.

```
fac :: Int  -> Int
fac 0     = 1
fac (x+1) = (x+1) * fac x
```

Natürliche Zahlen werden beim Pattern Matching so behandelt, also ob sie die Form `0` oder `0 + 1 + ...+ 1` hätten. Für nicht-negative ganze Zahlen erhalten wir also die folgende Regel.

$$\underline{\texttt{Int}} \;\rightarrow\; 0 \,|\, \underline{\texttt{Int}} + 1$$

Bei der Berechnung von `fac 2` wird also zunächst `2` als `0 + 1 + 1` geschrieben. Wenn man `x` durch `0 + 1` instantiiert, ist daher die zweite definierende Gleichung von `fac` anwendbar und man erhält `(0 + 1 + 1) * (fac (0 + 1)) = 2 * (fac (0 + 1))`. Nun muss `x` mit `0` instantiiert werden, um die Gleichung erneut anzuwenden. Dies ergibt `2 * ((0 + 1) * (fac 0)) = 2 * (1 * (fac 0))`. Die Anwendung der ersten Gleichung führt nun zu `2 * (1 * 1) = 2 * 1 = 2`.

Schließlich sei noch erwähnt, dass in HASKELL keine Vollständigkeit der definierenden Gleichungen gefordert ist. Die gegebene Definition von `fac` ist beispielsweise bei negativen Zahlen undefiniert. Das könnte man allerdings beheben, indem man eine weitere dritte Gleichung

$$\texttt{fac}\,\texttt{x} = 1$$

ergänzt.

Beim Pattern Matching mit ganzen Zahlen muss man statt `x + 1 + 1` jeweils `x + 2` schreiben (etc.). So lautet der Algorithmus zur Berechnung der abgerundeten Hälfte wie folgt:

```
half :: Int  -> Int
half 0     = 0
half 1     = 0
half (x+2) = 1 + half x
```

Patterndeklarationen

Nicht nur Funktionen, sondern auch andere Werte können in Deklarationen festgelegt werden:

```
pin :: Float
pin = 3.14159

suc :: Int -> Int
suc = plus 1

x0, y0 :: Int
```

```
(x0, y0) = (1,2)

x1, y1 :: Int
[x1,y1] = [1,2]

x2 :: Int
y2 :: [Int]
x2:y2 = [1,2]
```

Hierbei ist `Float` der vordefinierte Typ für Gleitkommazahlen.

Im allgemeinen darf einem beliebigem *Pattern* ein Ausdruck zugewiesen werden. Im einfachsten Fall ist ein Pattern eine Variable. Sonst ist es ein Ausdruck wie z.B. `(x0, y0)`, so dass bei einer Zuweisung eines Werts wie `(1,2)` an diesen Ausdruck eindeutig festliegt, welche Werte den einzelnen Variablenbezeichnern zugewiesen werden. Eine Patternbindung darf für jeden Bezeichner nur einmal vorkommen (wohingegen Funktionsbindungen mehrfach — mit verschiedenen Pattern für die Argumente — auftreten dürfen).

Wir erweitern also die Möglichkeiten für Deklarationen $\underline{\mathsf{decl}}$ nun um Patterndeklarationen wie folgt:

$$\begin{array}{rcl} \underline{\mathsf{decl}} & \rightarrow & \underline{\mathsf{typedecl}} \mid \underline{\mathsf{fundecl}} \mid \underline{\mathsf{patdecl}} \\ \underline{\mathsf{patdecl}} & \rightarrow & \underline{\mathsf{pat}}\ \underline{\mathsf{rhs}} \end{array}$$

Lokale Deklarationen

Lokale Deklarationen werden verwendet, um innerhalb einer Deklaration einen weiteren lokalen Deklarationsblock zu erstellen. In jeder rechten Seite einer Funktions- oder Patterndeklaration kann man dazu nach dem Schlüsselwort `where` eine Folge von lokalen Deklarationen angeben, die sich nur auf diese rechte Seite beziehen. Dabei werden äußere Deklarationen der gleichen Bezeichner von der lokalen Deklaration überdeckt. Die Grammatikregeln für $\underline{\mathsf{fundecl}}$ und $\underline{\mathsf{patdecl}}$ werden daher wie folgt geändert. Hierbei bedeuten eckige Klammern in der Grammatik, dass die darin befindlichen Ausdrücke optional sind.

$$\begin{array}{lcl} \underline{\mathsf{fundecl}} & \rightarrow & \underline{\mathsf{funlhs}}\ \underline{\mathsf{rhs}}\ [\mathtt{where}\ \underline{\mathsf{decls}}] \\ \underline{\mathsf{patdecl}} & \rightarrow & \underline{\mathsf{pat}}\ \underline{\mathsf{rhs}}\ [\mathtt{where}\ \underline{\mathsf{decls}}] \\ \underline{\mathsf{decls}} & \rightarrow & \{\underline{\mathsf{decl}}_1; \ldots ; \underline{\mathsf{decl}}_n\},\ \text{wobei } n \geq 0 \end{array}$$

Als Beispiel betrachten wir das folgende Programm, das die Lösungen einer quadratischen Gleichung mit Hilfe der folgenden Formel berechnet.

$$ax^2 + bx + c = 0 \Longleftrightarrow x = \frac{-b \pm \sqrt{b^2 - 4ac}}{2a}$$

```
roots :: Float -> Float -> Float -> (Float, Float)
roots a b c = ((-b - d)/e, (-b + d)/e)
              where { d = sqrt (b*b - 4*a*c); e = 2*a }
```

Ein wichtiger Vorteil lokaler Deklarationen ist, dass die darin deklarierten Werte nur einmal berechnet werden. Der Aufruf von `roots 1 5 3` erzeugt daher einen Graph

```
((-5 - ^d)/ ^e, (-5 + ^d)/ ^e),
```

wobei `^d` ein Zeiger auf eine Speicherzelle mit dem Ausdruck `sqrt (5*5 - 4*1*3)` und `^e` ein Zeiger auf `2*1` ist. Damit müssen diese beiden Ausdrücke also nur einmal ausgewertet werden und man kann mehrfache Auswertungen der gleichen Ausdrücke vermeiden.

Um Klammern zu vermeiden und die Lesbarkeit zu erhöhen, existiert in HASKELL die sogenannte *Offside-Regel* zur Schreibweise von (lokalen) Deklarationen:

1. Das erste Symbol in einer Sammlung <u>decls</u> von Deklarationen bestimmt den linken Rand des Deklarationsblocks.

2. Eine neue Zeile, die an diesem linken Rand anfängt, ist eine neue Deklaration in diesem Block.

3. Eine neue Zeile, die weiter rechts beginnt als dieser linke Rand, gehört zur selben Deklaration (d.h., sie ist die Fortsetzung der darüberliegenden Zeile). Beispielsweise steht

   ```
   d = sqrt (b*b -
             4*a*c)
   ```

   für

   ```
   d = sqrt (b*b - 4*a*c).
   ```

4. Eine neue Zeile, die weiter links beginnt als der linke Rand, bedeutet, dass der <u>decls</u>-Block beendet ist und sie nicht mehr zu dieser Sammlung von Deklarationen gehört.

Man kann also `decls` auch wie ein eingerücktes Programm schreiben (d.h., als Folge von Deklarationen, die linksbündig untereinander stehen). Beispielsweise ließe sich also die Deklaration von `roots` auch wie folgt schreiben:

```
roots a b c = ((-b - d)/e, (-b + d)/e)
              where d = sqrt (b*b - 4*a*c)
                    e = 2*a
```

Operatoren und Infixdeklarationen

Manche Funktionen sollten nicht in Präfix-, sondern in Infix-Schreibweise verwendet werden, um die Lesbarkeit von Programmen zu erhöhen. Beispiele hierfür sind `+`, `*`, `==` oder auch der Listenkonstruktor `:`, der verwendet wird, um Elemente in Listen einzufügen. Solche Funktionssymbole nennt man *Operatoren*. Wie bei den Präfix-Symbolen unterscheidet man auch hier zwischen Variablen und Konstruktoren. Letztere erhalten keine Funktionsdeklaration, sondern sie werden verwendet, um Objekte einer Datenstruktur zu repräsentieren. Operatoren werden in HASKELL durch Folgen von Sonderzeichen repräsentiert. Konstruktoroperatoren (wie `:`) beginnen dabei mit einem Doppelpunkt und Variablenoperatoren (wie `+` oder `==`) beginnen mit einem anderen Zeichen.

Jeder Infix-Operator kann durch Klammerung zu einer Präfix-Funktion umgewandelt werden. So kann man "`(+) 2 3`" statt "`2 + 3`" schreiben. Analog kann auch jede zweistellige Präfix-Funktion (mit einem Typ $\underline{\texttt{type}}_1$ `->` $\underline{\texttt{type}}_2$ `->` $\underline{\texttt{type}}_3$) in einen Infix-Operator durch Verwendung von "Backquotes" gewandelt werden. So kann man "`2 ‘plus‘ 3`" statt "`plus 2 3`" schreiben. Die Verwendung von Infix-Operatoren bedeutet insofern wirklich nur eine andere Schreibweise. Wir werden daher in den folgenden Definitionen der Syntax immer nur auf Präfix-Funktionen eingehen. Die Verwendung der alternativen Schreibweise mit Infix-Operatoren ist aber in konkreten Programmen oft hilfreich. Zwei Eigenschaften sind bei Infix-Operatoren wichtig:

1. *Assoziation*
   Betrachten wir den folgenden Algorithmus.

   ```
   divide :: Float -> Float -> Float
   divide x y = x/y
   ```

   In dem Ausdruck

   ```
   36 ‘divide‘ 6 ‘divide‘ 2
   ```

   ist zunächst nicht klar, ob das Ergebnis `3` oder `12` ist. Hierzu muss man festlegen, zu welcher Seite der Operator `‘divide‘` *assoziiert*. Daher kann man bei Infix-Operatoren die Assoziation deklarieren. Falls `divide` nach links assoziieren soll, so fügt man die Deklaration

   ```
   infixl ‘divide‘
   ```

   ein. Dies ist auch der Default für Operatoren in HASKELL. In diesem Fall steht der obige Ausdruck für

   ```
   (36 ‘divide‘ 6) ‘divide‘ 2
   ```

   und das Ergebnis ist somit `3`. Deklariert man hingegen

   `infixr ‘divide‘`,

   so assoziiert `‘divide‘` nach rechts. Der obige Ausdruck steht dann für `36 ‘divide‘ (6 ‘divide‘ 2)`, so dass sich `12` ergibt. Eine dritte Möglichkeit ist die Deklaration

   `infix ‘divide‘`.

   Dies bedeutet, dass `‘divide‘` gar keine Assoziation besitzt. Dann würde der Ausdruck `36 ‘divide‘ 6 ‘divide‘ 2` zu einer Fehlermeldung führen.

   Das Konzept der Assoziation haben wir bereits bei dem Funktionsraumkonstruktor und der Funktionsanwendung kennen gelernt. Wie erwähnt, assoziiert der Funktionsraumkonstruktor `->` nach rechts, d.h., `Int -> Int -> Int` steht für `Int -> (Int ->`

`Int`). Die Funktionsanwendung assoziiert nach links. Somit würde ein Ausdruck wie `square square 3` für `(square square) 3` stehen, d.h. für einen nicht typkorrekten Ausdruck, der zu einer Fehlermeldung führt.

2. *Bindungspriorität*
   Wir definieren die folgenden beiden Funktionen.

```
(%%) :: Int -> Int -> Int
x %% y = x + y

(@@) :: Int -> Int ->Int
x @@ y = x * y
```

Die Frage ist nun, zu welchem Wert der Ausdruck

```
1 %% 2 @@ 3
```

auswertet, d.h., die Frage ist, welcher der beiden Operatoren `%%` und `@@` höhere Priorität besitzt. Hierzu kann man bei Infixdeklarationen (mit `infixl`, `infixr` oder `infix`) die Bindungspriorität mit Hilfe einer Zahl zwischen `0` und `9` angeben, wobei `9` die höchste Bindungspriorität repräsentiert. (Falls keine Bindungspriorität angegeben ist, so ist `9` der Defaultwert.) Beispielsweise könnte man folgendes deklarieren.

```
infixl 9 %%
infixl 8 @@
```

Dann steht `1 %% 2 @@ 3` für `(1 %% 2) @@ 3` und das Ergebnis ist `9`. Vertauscht man hingegen die Bindungsprioritäten `9` und `8`, so steht der Ausdruck für `1 %% (2 @@ 3)` und es ergibt sich `7`. Bei gleichen Bindungsprioritäten wird die Auswertung von links nach rechts vorgenommen.

Da es nun also auch Infixdeklarationen gibt, muss die Grammatikregel für Deklarationen noch einmal erweitert werden. Hierbei stehen große geschweifte Klammern für Wahlmöglichkeiten in der Grammatikregel.

$$
\begin{array}{rcl}
\underline{\mathsf{decl}} & \rightarrow & \underline{\mathsf{typedecl}} \mid \underline{\mathsf{fundecl}} \mid \underline{\mathsf{patdecl}} \mid \underline{\mathsf{infixdecl}} \\
\underline{\mathsf{infixdecl}} & \rightarrow & \left\{ \begin{array}{l} \texttt{infix} \\ \texttt{infixl} \\ \texttt{infixr} \end{array} \right\} \ [\left\{ \begin{array}{c} \texttt{0} \\ \texttt{1} \\ \vdots \\ \texttt{9} \end{array} \right\}] \ \underline{\mathsf{op}}_1, \ldots, \underline{\mathsf{op}}_n, \ \text{wobei } n \geq 1 \\
\underline{\mathsf{op}} & \rightarrow & \underline{\mathsf{varop}} \mid \underline{\mathsf{constrop}}
\end{array}
$$

Schließlich sei noch erwähnt, dass Operatoren (ähnlich wie Präfix-Funktionen) auch partiell angewendet werden können (d.h., eine Anwendung ist auch möglich, wenn nicht alle benötigten Argumente vorliegen). So ist `(+ 2)` die Funktion vom Typ `Int -> Int`, die Zahlen um `2` erhöht. Die Funktion ``(6 `divide`)`` vom Typ `Float -> Float` nimmt ein Argument und dividiert die Zahl `6` durch dieses Argument. Die Funktion ``` `divide` 6``` hingegen ist die Funktion vom Typ `Float -> Float`, die ihr Argument durch `6` teilt.

Zusammenfassung der Syntax für Deklarationen

Zusammenfassend ergibt sich die folgende Grammatik zur Erzeugung von Deklarationen in HASKELL.

decl	→ typedecl | fundecl | patdecl | infixdecl	
typedecl	→ $\text{var}_1, \ldots, \text{var}_n$ :: type,	wobei $n \geq 1$
var	→ String von Buchstaben und Zahlen mit Kleinbuchstaben am Anfang	
fundecl	→ funlhs rhs [`where` decls]	
funlhs	→ var $\text{pat}_1 \ldots \text{pat}_n$	wobei $n \geq 1$
rhs	→ = exp| $\text{condrhs}_1 \ldots \text{condrhs}_n$	wobei $n \geq 1$
condrhs	→ | exp = exp	
decls	→ $\{ \text{decl}_1; \ldots; \text{decl}_n \}$,	wobei $n \geq 0$
patdecl	→ pat rhs [`where` decls]	
infixdecl	→ $\left\{ \begin{array}{l} \texttt{infix} \\ \texttt{infixl} \\ \texttt{infixr} \end{array} \right\} \left[\left\{ \begin{array}{c} 0 \\ 1 \\ \vdots \\ 9 \end{array} \right\} \right] \text{op}_1, \ldots, \text{op}_n$,	wobei $n \geq 1$
op	→ varop | constrop	
varop	→ String von Sonderzeichen, der nicht mit : beginnt	
constrop	→ String von Sonderzeichen, der mit : beginnt	

1.1.2 Ausdrücke

Ausdrücke exp (Expressions) stellen das zentrale Konzept der funktionalen Programmierung dar. Ein Ausdruck beschreibt einen Wert (z.B. eine Zahl, einen Buchstaben oder eine Funktion). Die Eingabe eines Ausdrucks in den Interpreter bewirkt seine Auswertung. Darüber hinaus besitzt jeder Ausdruck einen Typ. Bei der Eingabe von "`:t` exp" in den HASKELL-Interpreter HUGS wird der Typ von exp berechnet und ausgegeben. Bei der Auswertung eines Ausdrucks wird ebenfalls zuerst überprüft, ob der Ausdruck korrekt getypt ist und nur im Erfolgsfall wird die Auswertung tatsächlich durchgeführt. Ein Ausdruck exp kann folgende Gestalt haben:

- var
  Variablenbezeichner wie `x` sind Ausdrücke. Wie erwähnt, werden Variablenbezeichner in HASKELL durch Strings gebildet, die mit einem Kleinbuchstaben beginnen.

- constr
  Eine andere Möglichkeit für Ausdrücke sind *Datenkonstruktoren*. Datenkonstruktoren dienen zum Aufbau von Objekten einer Datenstruktur und werden bei der Datentypdefinition eingeführt. In HASKELL werden Bezeichner für Datenkonstruktoren durch Strings gebildet, die mit Großbuchstaben beginnen. Beispiele hierfür sind die Datenkonstruktoren `True` und `False` der vordefinierten Datenstruktur `Bool`. Ein weiteres Beispiel sind die Datenkonstruktoren `[]` und : für die vordefinierte Datenstruktur der Listen.

- integer
  Auch die ganzen Zahlen `0, 1, -1, 2, -2, ...` sind Ausdrücke.

- float
  Gleitkommazahlen wie `-2.5` oder `3.4e+23` sind ebenfalls Ausdrücke.

- char
  Weitere Ausdrücke sind `'a',...,'z','A',...,'Z','0',...,'9'` sowie das Leerzeichen `' '` und nicht druckbare Kontrollzeichen wie `'\n'` für das Zeilenende-Zeichen. All diese Zeichen werden zu sich selbst (in Apostrophen (Quotes)) ausgewertet.

- `[`$\text{exp}_1, \ldots, \text{exp}_n$`]`, wobei $n \geq 0$
  Solch ein Ausdruck bezeichnet eine Liste von n Ausdrücken. Wie erwähnt, repräsentiert `[]` hierbei die leere Liste und `[0,1,2,3]` ist eine Abkürzung für `0 : 1 : 2 : 3 : []`, wobei `:` nach rechts assoziiert. Alle Elemente einer Liste müssen denselben Typ haben. Der Typ der obigen Liste wäre z.B. `[Int]`, d.h. der Typ der Listen von ganzen Zahlen.

- string
  Ein string ist eine Liste von Zeichen char (d.h., es ist ein Ausdruck vom Typ `[Char]`). Statt `['h','a','l','l','o']` schreibt man oft `"hallo"`. Solch ein String wird zu sich selbst ausgewertet. Der vordefinierte Typ `String` in Haskell ist identisch mit dem Typ `[Char]`.

- ($\text{exp}_1, \ldots, \text{exp}_n$), wobei $n \geq 0$
  Dies ist ein Tupel von Ausdrücken. Anders als bei der Liste können die Ausdrücke in einem Tupel verschiedene Typen haben. Ein Beispiel wäre der Ausdruck `(10, False)`. Dieser Ausdruck hätte z.B. den Typ `(Int, Bool)`. Einelementige Tupel `(`exp`)` werden zu exp ausgewertet. Nullelementige Tupel `()` haben den speziellen Typ `()`.

- ($\text{exp}_1 \ldots \text{exp}_n$), wobei $n \geq 2$
  Solch ein Ausdruck steht für die Funktionsanwendung von Ausdrücken. Hierbei lassen wir die Klammerung soweit wie möglich weg. Die Funktionsanwendung hat die höchste Bindungspriorität und assoziiert nach links. Beispiele für solche Ausdrücke sind `square 10` (vom Typ `Int`) oder `plus 5 3` (ebenfalls vom Typ `Int`) oder `plus 5` (vom Typ `Int -> Int`). Der Wert eines Ausdrucks kann also wieder eine Funktion sein.

- `if` exp_1 `then` exp_2 `else` exp_3
  Hierbei muss exp_1 vom Typ `Bool` sein und exp_2 und exp_3 müssen denselben Typ haben. Bei der Auswertung wird erst der Wert von exp_1 bestimmt und danach in Abhängigkeit dieses Werts der Wert von exp_2 oder exp_3. Statt

```
maxi(x, y) | x >= y     = x
           | otherwise  = y
```

  kann man also auch folgendes schreiben:

```
maxi(x, y) = if x >= y then x else y
```

- `let` decls `in` exp
  In diesem Ausdruck wird eine lokale Deklarationsfolge decls für den Ausdruck exp definiert. Dies ist analog zur lokalen Deklaration mit Hilfe von `where`, nur wird jetzt die lokale Deklaration voran- statt nachgestellt. Statt

```
roots a b c = ((-b - d)/e, (-b + d)/e)
              where d = sqrt (b*b - 4*a*c)
                    e = 2*a
```

  kann man also auch folgendes schreiben:

```
roots a b c = let d = sqrt (b*b - 4*a*c)
                  e = 2*a
                  in ((-b - d)/e, (-b + d)/e)
```

- `case` exp `of` {pat_1 `->` exp_1 `;` ... `;` pat_n `->` exp_n}, wobei $n \geq 1$
  Bei der Auswertung dieses Ausdrucks wird versucht, den Pattern pat_1 auf den Ausdruck exp zu matchen. Gelingt dies, so ist das Ergebnis der Ausdruck exp_1, wobei die Variablen mit der verwendeten Matching-Substitution instantiiert werden. Ansonsten wird anschließend versucht, den Pattern pat_2 auf exp zu matchen, etc. Hierbei ist wieder die Offside-Regel zur Schreibweise verwendbar. Statt

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

  kann man also auch folgendes schreiben:

```
und x y = case x
            of True  -> y
               False -> False
```

  Außerdem kann man statt der Ausdrücke exp_i auch Folgen von bedingten Ausdrücken `|` exp `->` exp verwenden und darüber hinaus ist es in jeder Alternative des `case`-Ausdrucks möglich, lokale Deklarationen mit `where` zu vereinbaren.

- `\`$\text{pat}_1 \ldots \text{pat}_n$ `->` exp, wobei $n \geq 1$
  Solch ein Ausdruck wird als "Lambda-Ausdruck" oder "Lambda-Abstraktion" bezeichnet, denn das Zeichen \ (backslash) repräsentiert den griechischen Buchstaben λ. Der Wert dieses Ausdrucks ist die Funktion, die die Argumente $\text{pat}_1 \ldots \text{pat}_n$ auf exp abbildet. Beispielsweise ist `\x -> 2 * x` die Funktion, die ein Argument nimmt und es verdoppelt. Ihr Typ ist `Int -> Int`. Mit "Lambda" bildet man also sogenannte

"unbenannte Funktionen", die nur an der Stelle ihrer Definition verwendet werden können. Der Ausdruck

```
(\x -> 2 * x) 5
```

wertet daher zu `10` aus. Die Funktion `\x y -> x + y` ist die Additionsfunktion vom Typ `Int -> Int -> Int`. Allgemein hat der Ausdruck $\backslash\underline{\mathsf{pat}}_1 \ldots \underline{\mathsf{pat}}_n$ `->` $\underline{\mathsf{exp}}$ den Typ $\underline{\mathsf{type}}_1$ `->...->` $\underline{\mathsf{type}}_n$ `->` $\underline{\mathsf{type}}$, falls $\underline{\mathsf{pat}}_i$ jeweils den Typ $\underline{\mathsf{type}}_i$ und $\underline{\mathsf{exp}}$ den Typ $\underline{\mathsf{type}}$ hat. Bei Lambda-Ausdrücken sind beliebige Patterns möglich, d.h., man kann auch Ausdrücke wie `\(x, y) -> x + y` vom Typ `(Int, Int) -> Int` bilden. An den Lambda-Ausdrücken wird deutlich, dass Funktionen in funktionalen Programmiersprachen wirklich gleichberechtigte Datenobjekte sind, denn man kann sie nun komplett durch geeignete Ausdrücke beschreiben.

Anstelle der Funktionsdeklaration

```
plus x y = x + y
```

kann man nun also

```
plus = \x y -> x + y
```

oder

```
plus x = \y -> x + y
```

definieren.

Zusammenfassung der Syntax für Ausdrücke

Zusammenfassend ergibt sich die folgende Grammatik für Ausdrücke in HASKELL.

$$
\begin{array}{lcll}
\underline{\mathsf{exp}} & \rightarrow & \underline{\mathsf{var}} & \\
& | & \underline{\mathsf{constr}} & \\
& | & \underline{\mathsf{integer}} & \\
& | & \underline{\mathsf{float}} & \\
& | & \underline{\mathsf{char}} & \\
& | & \texttt{[}\underline{\mathsf{exp}}_1, \ldots, \underline{\mathsf{exp}}_n\texttt{]}, & \text{wobei } n \geq 0 \\
& | & \underline{\mathsf{string}} & \\
& | & (\underline{\mathsf{exp}}_1, \ldots, \underline{\mathsf{exp}}_n), & \text{wobei } n \geq 0 \\
& | & (\underline{\mathsf{exp}}_1\ \ldots\ \underline{\mathsf{exp}}_n), & \text{wobei } n \geq 2 \\
& | & \texttt{if}\ \underline{\mathsf{exp}}_1\ \texttt{then}\ \underline{\mathsf{exp}}_2\ \texttt{else}\ \underline{\mathsf{exp}}_3 & \\
& | & \texttt{let}\ \underline{\mathsf{decls}}\ \texttt{in}\ \underline{\mathsf{exp}} & \\
& | & \texttt{case}\ \underline{\mathsf{exp}}\ \texttt{of}\ \{\underline{\mathsf{pat}}_1\ \texttt{->}\ \underline{\mathsf{exp}}_1; \ldots; \underline{\mathsf{pat}}_n\ \texttt{->}\ \underline{\mathsf{exp}}_n\}, & \text{wobei } n \geq 1 \\
& | & \backslash\underline{\mathsf{pat}}_1 \ldots \underline{\mathsf{pat}}_n\ \texttt{->}\ \underline{\mathsf{exp}}, & \text{wobei } n \geq 1
\end{array}
$$

$\underline{\mathsf{constr}}$ $\rightarrow$ String von Buchstaben und Zahlen mit Großbuchstaben am Anfang

1.1.3 Patterns

Bei der Funktionsdeklaration werden sogenannte *Patterns* für die Argumente angegeben. Sie schränken die Form der erlaubten Argumente ein. Die Syntax von Patterns ist daher ähnlich wie die Syntax von Ausdrücken, denn Patterns sind Prototypen für die erwarteten Werte. Die Form der Werte wird durch die vorkommenden Datenkonstruktoren beschrieben, wobei statt mancher Teilwerte im Pattern Variablen stehen. (Wir verwenden Datenkonstruktoren nun also zur Zerlegung statt zur Konstruktion von Objekten.) Ein Pattern passt zu einem Ausdruck (bzw. er *matcht* diesen Ausdruck), wenn dieser aus dem Pattern bei einer Ersetzung der Variablen durch andere Teil-Ausdrücke hervorgeht. Als Beispiel hatten wir bereits die Algorithmen `und`, `len`, `second`, `fac` und `half` in Abschnitt 1.1.1 betrachtet.

Als weiteres Beispiel betrachten wir den Algorithmus `append`. (Eine analoge (Infix)-Funktion `++` (auf Listen mit Elementen beliebigen Typs) ist in Haskell vordefiniert.)

```
append :: [Int]  -> [Int]  -> [Int]
append []     ys = ys
append (x:xs) ys = x : app xs y
```

Um `len (append [1] [2])` zu berechnen, wird das Argument `append [1] [2]` von `len` nur so weit ausgewertet, bis man entscheiden kann, welcher Pattern in der Definition von `len` matcht. Hier würde man also das Argument nur zu `1:append [] [2]` auswerten. An dieser Stelle ist bereits klar, dass nur die zweite Gleichung von `len` verwendbar ist und man erhält `1 + len (append [] [2])`, was dann weiter ausgewertet wird. Lässt sich ohne Auswertung des Arguments nicht feststellen, ob der betrachtete Pattern matcht, so wird der Argumentausdruck zunächst nur so lange ausgewertet, bis der äußerste Konstruktor des Arguments feststeht. (Man bezeichnet dies als *Weak Head Normal Form*, vgl. Kapitel 3.) Dann kann man überprüfen, ob dieser Konstruktor mit dem äußersten Konstruktor des Patterns übereinstimmt. Gegebenenfalls kann es dann zu einem weiteren rekursiven Aufruf des Pattern Matching-Verfahrens für die Teil-Argumente kommen.

Betrachten wir beispielsweise die folgenden Definitionen.

```
zeros :: [Int]
zeros = 0 : zeros

f :: [Int]  -> [Int]  -> [Int]
f [] ys = []
f xs [] = []
```

Die Auswertung von `f [] zeros` terminiert, obwohl `zeros` für sich alleine genommen nicht terminiert. Der Grund ist, dass keine Auswertung von `zeros` nötig ist, um herauszufinden, dass die erste Gleichung von `f` anwendbar ist. Aber auch `f zeros []` terminiert. Hier wird zunächst `zeros` in einem Schritt zu `0 : zeros` ausgewertet. Nun liegt der äußerste Konstruktor “`:`” von `f`’s erstem Argument fest. Da dieser Konstruktor verschieden von dem Konstruktor `[]` ist, kann die erste Gleichung nicht anwendbar sein und man verwendet daher die zweite Gleichung.

Ein Beispiel für die Anwendung des Pattern Matching in Patterndeklarationen ist

```
let x:xs = [1,2,3] in xs
```

Hier ist `x:xs` ein Pattern, der auf den Ausdruck `[1,2,3]` gematcht wird. Das Matchen ist erfolgreich bei der Substitution [`x/1, xs/[2,3]`]. Der obige Ausdruck wird daher zu `[2,3]` ausgewertet.

Eine Einschränkung an Patterns ist, dass sie *linear* sein müssen, d.h., keine Variable darf in einem Pattern mehrfach vorkommen. Der Grund dafür ist, dass sonst nicht mehr alle Auswertungsstrategien dasselbe Ergebnis liefern. Beispielsweise könnte man dann folgende Funktionen deklarieren.

```
equal :: [Int] -> [Int] -> Bool
equal xs xs     = True
equal xs (x:xs) = False
```

Der Ausdruck `equal zeros zeros` könnte nun je nach Auswertungsstrategie sowohl zu `True` als auch zu `False` ausgewertet werden. Im allgemeinen kann ein Pattern $\underline{\textsf{pat}}$ folgende Gestalt haben:

- $\underline{\textsf{var}}$
  Jeder Variablenbezeichner ist auch ein Pattern. Dieser Pattern passt auf jeden Wert, wobei die Variable beim Matching an diesen Wert gebunden wird. Ein Beispiel für eine Funktionsdeklaration, bei der solch ein Pattern verwendet wird, ist

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

- _
  Das Zeichen _ (underscore) ist der Joker-Pattern. Er passt ebenfalls auf jeden Wert, aber es erfolgt keine Variablenbindung. Der Joker _ darf daher auch mehrmals in einem Pattern auftreten. Beispielsweise kann man die Funktion `und` also auch wie folgt definieren:

  ```
  und True y = y
  und _ _ = False
  ```

- $\underline{\textsf{integer}}$ oder $\underline{\textsf{float}}$ oder $\underline{\textsf{char}}$ oder $\underline{\textsf{string}}$
  Diese Patterns passen jeweils nur auf sich selbst und es findet keine Variablenbindung beim Matching statt.

- $(\underline{\textsf{constr}}\ \underline{\textsf{pat}}_1 \ldots \underline{\textsf{pat}}_n)$, wobei $n \geq 0$
  Hierbei ist $\underline{\textsf{constr}}$ ein n-stelliger Datenkonstruktor. Dieser Pattern matcht Werte, die mit demselben Datenkonstruktor gebildet werden, falls jeweils $\underline{\textsf{pat}}_i$ das i-te Argument des Werts matcht. Beispiele hierfür hatten wir bei der Deklaration der Algorithmen `und`, `len` und `append` gesehen. (Hierbei ist "`:`" ein Infix-Konstruktor, deshalb steht er nicht außen. (Man kann stattdessen auch `((:) x xs)` schreiben.) Wie üblich lassen wir Klammern soweit möglich weg, um die Lesbarkeit zu erhöhen.

- var@pat
  Dieser Pattern verhält sich wie pat, aber falls pat auf den zu matchenden Ausdruck passt, wird zusätzlich die Variable var an den gesamten Ausdruck gebunden. Als Beispiel betrachten wir die folgende Funktion, die das erste Element einer Liste kopiert.

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

  Man könnte also nun statt der zweiten definierenden Gleichung auch folgende Gleichung verwenden.

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

- (var + integer)
  Natürliche Zahlen werden beim Pattern Matching so behandelt, also ob sie mit Hilfe von (nicht-vorhandenen) Datenkonstruktoren `0 :: Int` und `...+ 1 :: Int -> Int` konstruiert werden würden. (Hier steht `Int` also nur für nicht-negative ganze Zahlen.) Eine Zahl k würde dann als $\texttt{0} + \texttt{1} \ldots + \texttt{1}$ mit k `1`en dargestellt. Der Pattern var $+ k$ (mit $k \in \mathbb{N}$) ist dann als Abkürzung für var$+\texttt{1} \ldots + \texttt{1}$ mit k `1`en zu verstehen. Der Pattern var$+k$ passt nur auf natürliche Zahlen n mit $n \geq k$ und die Variable var wird an den Wert $n - k$ gebunden. Beispiele hierfür waren die Funktionen `fac` und `half` aus Abschnitt 1.1.1 oder auch die folgende Funktion.

  ```
  sub7 :: Int -> Int
  sub7 (x + 7) = x
  ```

  Die Funktion `sub7` subtrahiert `7` von ihrem Argument, falls dieses größer oder gleich `7` ist. Anderenfalls ist `sub7` undefiniert.

  Um zu entscheiden, ob var$+k$ oder auch eine integer-Zahl einen `Int`-Ausdruck matcht, muss dieser *komplett* ausgewertet werden. Betrachten wir hierzu folgendes Beispiel.

  ```
  infinity :: Int
  infinity = infinity + 1

  f :: Int  -> Int  -> Int
  f 0 y = 0
  f x 0 = 0
  ```

  Die Auswertung von `f 0 infinity` terminiert natürlich, aber `f infinity 0` terminiert nicht. Der Grund ist, dass man zwar `infinity` zu `infinity + 1` auswerten kann. Hieraus kann man aber nicht schließen, dass die erste Gleichung nicht anwendbar ist, denn `infinity` könnte ja den Wert `-1` haben und dann würde `infinity + 1` dem Pattern `0` entsprechen.

  Im Unterschied zu den anderen Patterns dürfen solche Patterns nicht in Patterndeklarationen verwendet werden. Ein Ausdruck wie `let (n+7) = 8 in n` ist also nicht zulässig.

- $[\underline{\mathsf{pat}}_1, \ldots, \underline{\mathsf{pat}}_n]$, wobei $n \geq 0$
  Solch ein Pattern matcht Listen der Länge n, falls $\underline{\mathsf{pat}}_i$ jeweils das i-te Element der Liste matcht. Das folgende Beispiel dient dazu, Listen der Länge 3 zu erkennen:

```
has_length_three :: [Int] -> Bool
has_length_three [x,y,z] = True
has_length_three _       = False
```

- $(\underline{\mathsf{pat}}_1, \ldots, \underline{\mathsf{pat}}_n)$, wobei $n \geq 0$
  Analog matcht ein solcher Tupelpattern Tupel mit n Komponenten, falls $\underline{\mathsf{pat}}_i$ jeweils die i-te Komponente des Tupels matcht. Der Pattern `()` matcht nur den Wert `()`. Hierdurch kann man `maxi` alternativ wie folgt definieren:

```
maxi :: (Int, Int) -> Int
maxi (0,y)     = y
maxi (x,0)     = x
maxi (x+1,y+1) = 1 + maxi (x,y)
```

Generell ist also jeder lineare Term aus Datenkonstruktoren und Variablen ein Pattern.

Zusammenfassung der Syntax für Patterns

Wir erhalten die folgenden Regeln zur Konstruktion von Patterns.

$$
\begin{array}{rcll}
\underline{\mathsf{pat}} & \rightarrow & \underline{\mathsf{var}} & \\
& | & \_ & \\
& | & \underline{\mathsf{integer}} & \\
& | & \underline{\mathsf{float}} & \\
& | & \underline{\mathsf{char}} & \\
& | & \underline{\mathsf{string}} & \\
& | & (\underline{\mathsf{constr}}\ \underline{\mathsf{pat}}_1\ \ldots\ \underline{\mathsf{pat}}_n), & \text{wobei } n \geq 0 \\
& | & \underline{\mathsf{var}}@\underline{\mathsf{pat}} & \\
& | & (\underline{\mathsf{var}} + \underline{\mathsf{integer}}) & \\
& | & [\underline{\mathsf{pat}}_1, \ldots, \underline{\mathsf{pat}}_n], & \text{wobei } n \geq 0 \\
& | & (\underline{\mathsf{pat}}_1, \ldots, \underline{\mathsf{pat}}_n), & \text{wobei } n \geq 0
\end{array}
$$

1.1.4 Typen

Jeder Ausdruck in HASKELL hat einen Typ. Typen sind Mengen von gleichartigen Werten, die durch entsprechende Typausdrücke bezeichnet werden. Beispiele für uns bereits bekannte Typen sind die vordefinierten Typen `Bool`, `Int`, `Float` und `Char` sowie zusammengesetzte Typen wie `(Int,Int)`, `Int -> Int`, `(Int,Int) -> Int`, `[Int]`, `[Int -> Bool]`, `[[Int]]`, etc. Allgemein verwendet man die folgenden Arten von Typen $\underline{\mathsf{type}}$:

- $(\underline{\mathsf{tyconstr}}\ \underline{\mathsf{type}}_1\ \ldots\ \underline{\mathsf{type}}_n)$, wobei $n \geq 0$
  Typen werden im allgemeinen mit Hilfe von *Typkonstruktoren* $\underline{\mathsf{tyconstr}}$ aus anderen

Typen $\underline{\text{type}}_1, \dots, \underline{\text{type}}_n$ erzeugt. Beispiele für nullstellige (und vordefinierte) Typkonstruktoren sind `Bool`, `Int`, `Float` und `Char`. In HASKELL werden Typkonstruktoren mit Strings bezeichnet, die mit einem Großbuchstaben beginnen (leider sind sie also syntaktisch nicht von Datenkonstruktoren zu unterscheiden, die nicht Typen, sondern Objekte eines Datentyps erzeugen). Hierbei lassen wir wieder Klammern soweit wie möglich weg.

- [$\underline{\text{type}}$]
  Ein weiterer vordefinierter einstelliger Typkonstruktor ist `[...]`, der einen Typ als Eingabe bekommt und daraus einen neuen Typ erzeugt, dessen Objekte Listen aus Elementen des Ursprungstyps sind. Statt `[...]` $\underline{\text{type}}$ schreibt man [$\underline{\text{type}}$]. Beispiele für solche Typen sind `[Int]` und `[[Int]]` (der Typ der Listen von Listen ganzer Zahlen).

- ($\underline{\text{type}}_1$ -> $\underline{\text{type}}_2$)
  Ein weiterer vordefinierter Typkonstruktor ist der Funktionsraumkonstruktor `->`, der aus zwei Typen einen neuen Typ der Funktionen zwischen ihnen generiert. Ein Beispiel hierfür ist der Typ `Int -> Int`, den beispielsweise die Funktion `square` zur Quadrierung von Zahlen hat.

- ($\underline{\text{type}}_1, \dots, \underline{\text{type}}_n$), wobei $n \geq 0$
  Außerdem gibt es noch den vordefinierten und beliebigstelligen Tupelkonstruktor, mit dem Tupeltypen erzeugt werden können. Ein Beispiel hierfür ist `(Int, Bool, [Int -> Int])`. Wir werden sehen, dass neben diesen vordefinierten Typkonstruktoren auch der Benutzer beliebige weitere Typkonstruktoren definieren kann.

- $\underline{\text{var}}$
  Schließlich ist auch eine (Typ)variable ein Typ. Dies ist nötig, um parametrische Polymorphie zu erreichen, wie im folgenden erklärt wird.

Parametrische Polymorphie

"Polymorphie" bedeutet "Vielgestaltigkeit" und wird in der Informatik meistens verwendet, um auszudrücken, dass gleiche bzw. gleich heißende Funktionen für verschiedene Arten von Argumenten verwendet werden können. Man unterscheidet hierbei die *parametrische Polymorphie* und die *Ad-hoc-Polymorphie*. Bei der parametrischen Polymorphie wird ein und dieselbe Funktion für Argumente verschiedener Typen verwendet. Bei der Ad-hoc-Polymorphie wird zwar das gleiche Funktionssymbol für Argumente verschiedener Typen verwendet, aber abhängig vom Typ der Argumente werden verschiedene Funktionen ausgeführt. Funktionale Sprachen wie HASKELL besitzen beide Arten der Polymorphie, wie im folgenden erläutert wird.

Wir betrachten zunächst die parametrische Polymorphie. Hierbei wirkt eine Funktion gleichartig auf eine ganze Sammlung von Datenobjekten. Beispiele hierfür sind die folgenden Funktionen.

```
id :: a -> a
id x = x
```

```
len :: [a] -> Int
len []     = 0
len (x:xs) = len xs + 1
```

Wir haben im Typ der Funktionen `id` und `len` eine Typvariable `a` verwendet. Dies bedeutet, dass diese Funktionen für jede mögliche Ersetzung der Typvariablen durch Typen definiert sind. Beispielsweise darf man nun sowohl `len [True, False]` als auch `len [1,2,3]` aufrufen.

Analog verhält es sich auch bei der Funktion `append` (bzw. `++`, die in HASKELL vordefiniert ist).

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

Das mehrfache Vorkommen der gleichen Typvariable `a` im Typ `[a] -> [a] -> [a]` erzwingt die Übereinstimmung der Typen der beiden Argumente von `++`. Eine Funktion vom Typ $\underline{\mathsf{type}}_1$ `->` $\underline{\mathsf{type}}_2$ kann auf ein Argument vom Typ $\underline{\mathsf{type}}$ angewendet werden, falls es eine (allgemeinste) Ersetzung σ der Typvariablen (d.h. einen *allgemeinsten Unifikator*) gibt, so dass $\sigma(\underline{\mathsf{type}}_1) = \sigma(\underline{\mathsf{type}})$ ist. Das Ergebnis hat dann den Typ $\sigma(\underline{\mathsf{type}}_2)$.

Als Beispiel betrachten wir den Ausdruck `[True] ++ []`. Der Teilausdruck `[True]` hat den Typ `[Bool]` und das zweite Argument `[]` hat den Typ `[b]`. Die gesuchte Substitution σ mit $\sigma(\texttt{[a]}) = \sigma(\texttt{[Bool]}) = \sigma(\texttt{[b]})$ ist $\sigma = [\,\texttt{a/Bool},\ \texttt{b/Bool}\,]$. Also ist dieser Ausdruck korrekt getypt und er hat den Typ `[Bool]`. Die Untersuchung auf Typkorrektheit und die Berechnung allgemeinster Typen kann automatisch durchgeführt werden. Wir werden hierauf in Kapitel 4 genauer eingehen. Lässt man also die Typdeklaration von `app` weg, so wird automatisch der Typ `[a] -> [a] -> [a]` bestimmt.

Typdefinitionen: Einführung neuer Typen

Um neue Typen bzw. neue Typkonstruktoren einzuführen, gibt es in HASKELL eigene Formen der Deklaration. Diese Deklarationen sind aber (im Gegensatz zu den bisher betrachteten Deklarationen) nur auf der obersten Programmebene und nicht in lokalen Deklarationsblöcken möglich. Aus diesem Grund unterscheiden wir nun zwischen allgemeinen Deklarationen $\underline{\mathsf{decl}}$ und Deklarationen $\underline{\mathsf{topdecl}}$, die nur auf dieser obersten Ebene erlaubt sind. Ein Programm ist dann eine Folge von linksbündig untereinander stehenden $\underline{\mathsf{topdecl}}$-Deklarationen. Die Grammatik für $\underline{\mathsf{topdecl}}$ lautet wie folgt:

$$
\begin{array}{lll}
\underline{\mathsf{topdecl}} & \rightarrow & \underline{\mathsf{decl}} \\
 & | & \texttt{type}\ \underline{\mathsf{tyconstr}}\ \underline{\mathsf{var}}_1 \ldots \underline{\mathsf{var}}_n = \underline{\mathsf{type}}, \qquad \text{wobei } n \geq 0 \\
 & | & \texttt{data}\ \underline{\mathsf{tyconstr}}\ \underline{\mathsf{var}}_1 \ldots \underline{\mathsf{var}}_n = \\
 & & \qquad\quad \underline{\mathsf{constr}}_1\ \underline{\mathsf{type}}_{1,1} \ldots \underline{\mathsf{type}}_{1,n_1} \\
 & & \qquad \mid \ldots \\
 & & \qquad \mid \underline{\mathsf{constr}}_k\ \underline{\mathsf{type}}_{k,1} \ldots \underline{\mathsf{type}}_{k,n_k}, \qquad \text{wobei } n \geq 0, k \geq 1, n_i \geq 0
\end{array}
$$

Insbesondere können alle bislang behandelten Deklarationen decl also auch auf der obersten Programmebene auftreten. Zusätzlich kann man aber nun mit Hilfe der Schlüsselworte `type` und `data` neue Typen einführen.

Die erste Möglichkeit, neue Typen bzw. Typkonstruktoren zu definieren, ist die sogenannte *Typabkürzung* (type synonym) mit Hilfe des Schlüsselworts `type`. Beispielsweise kann man durch

```
type Position = (Float, Float)
type String  = [Char]
type Pair a b = (a, b)
```

drei neue Typkonstruktoren deklarieren (wobei `String` bereits auf diese Weise in HASKELL vordefiniert ist). Der Typ (bzw. der nullstellige Typkonstruktor) `Position` ist dann lediglich eine *Abkürzung* für den Typ `(Float, Float)`, d.h., diese beiden Typen werden als gleich betrachtet. Eine Typabkürzung kann Parameter haben, d.h., `Pair` ist ein zweistelliger Typkonstruktor. Wiederum handelt es sich hierbei aber nur um Abkürzungen, d.h., die Typen `Pair Float Float` und `Position` sind identisch.

Eine Einschränkung bei Typabkürzungen ist, dass die Variablen $\underline{\mathsf{var}}_1, \ldots, \underline{\mathsf{var}}_n$ paarweise verschieden sein müssen und dass der Typ type auf der rechten Seite keine Variablen außer diesen enthalten darf (ähnliche Einschränkungen gibt es auch bei Funktionsdeklarationen). Außerdem dürfen Typabkürzungen nicht rekursiv sein, d.h., der Typ type darf nicht von dem Typkonstruktor tyconstr, der gerade definiert wird, abhängen.

Die andere Möglichkeit zur Definition neuer Typen ist die Einführung von algebraischen Datentypen durch Angabe einer EBNF-artigen kontextfreien Grammatik. Dies geschieht mit Hilfe des Schlüsselworts `data`. Beispielsweise kann man die folgenden Aufzählungstypen definieren.

```
data Color = Red | Yellow | Green
data MyBool = MyTrue | MyFalse
```

In der Definition eines algebraischen Datentyps wie `Color` werden also die verschiedenen Möglichkeiten aufgezählt, wie mit entsprechenden Datenkonstruktoren (wie `Red`, `Yellow`, `Green`) Objekte dieses Typs konstruiert werden können. Durch diese Definitionen sind nun zwei neue nullstellige Typkonstruktoren `Color` und `MyBool` eingeführt worden. Die folgenden beiden Funktionen verdeutlichen, dass das Pattern Matching auch bei selbstdefinierten Datenstrukturen verwendbar ist. Jeder lineare Term aus Variablen und Datenkonstruktoren ist ein Pattern.

```
traffic_light :: Color -> Color
traffic_light Red    = Green
traffic_light Green  = Yellow
traffic_light Yellow = Red

und :: MyBool -> MyBool -> MyBool
und MyTrue y = y
und _      _ = MyFalse
```

Man kann Werte von selbstdefinierten Typen nicht direkt ausgeben. Bei der Ausgabe auf dem Bildschirm wird eine vordefinierte Funktion `show` aufgerufen, die den Wert in einen String umwandelt. Diese existiert bei den vordefinierten Typen. Bei eigenen Typen kann sie selbst geschrieben werden. Alternativ kann man sie automatisch erzeugen lassen. Hierzu muss bei der Datentypdeklaration `deriving Show` hinzugefügt werden. Zum eigenen Schreiben dieser Funktion muss man das Konzept des Überschreibens in HASKELL einführen, das anschließend betrachtet wird.

Betrachten wir ein weiteres Beispiel, bei dem die Datenstruktur der natürlichen Zahlen definiert wird.

```
data Nats = Zero | Succ Nats
```

Der Datentyp `Nats` besitzt zwei Datenkonstruktoren `Zero` und `Succ`. In der Datentypdeklaration werden jeweils nach dem Namen des Datenkonstruktors die Typen seiner Argumente angegeben. Somit hat `Zero` keine Argumente, d.h., sein Typ ist `Nats`. `Succ` hingegen hat den Typ `Nats -> Nats`. Wenn `Succ` die Nachfolgerfunktion repräsentiert, so steht `Succ (Succ Zero)` für die Zahl 2. Nun lassen sich auf diesem Datentyp Funktionen wie `plus` oder `half` definieren.

```
plus :: Nats -> Nats -> Nats
plus Zero     y = y
plus (Succ x) y = Succ (plus x y)

half :: Nats -> Nats
half Zero            = Zero
half (Succ Zero)     = Zero
half (Succ (Succ x)) = Succ (half x)
```

Der Unterschied zwischen `Nats` und `Int` wird deutlich, wenn man das `infinity`-Beispiel nun auf `Nats` betrachtet.

```
infinity :: Nats
infinity = Succ infinity

f :: Nats  -> Nats  -> Nats
f Zero y    = Zero
f x    Zero = Zero
```

Im Unterschied zu der Version auf `Int` terminiert nun `f infinity Zero`. Wenn man nämlich `infinity` einen Schritt auswertet, stellt man fest, dass das Ergebnis der Auswertung mit `Succ` beginnt. Da die Darstellung mit Hilfe von Datenkonstruktoren (außer bei `Int`) eindeutig ist, kann daher die erste Gleichung von `f` nicht anwendbar sein. Man kann daher die zweite Gleichung anwenden und erhält das Ergebnis `Zero`.

Natürlich können aber auch parametrische Typen (d.h. nicht-nullstellige Typkonstruktoren) definiert werden. Auf folgende Weise kann man einen Listentyp definieren, der den in HASKELL bereits vordefinierten Listen entspricht.

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

Durch diese Deklaration wird ein neuer parametrisierter algebraischer Datentyp `List a` für Listen mit Elementen vom Typ `a` eingeführt. `List` ist ein einstelliger Typkonstruktor, d.h., `List Int` wären z.B. Listen von ganzen Zahlen. Der Datentyp `List a` besitzt zwei Datenkonstruktoren `Nil` und `Cons`. `Nil` hat keine Argumente, d.h., sein Typ ist `List a`. `Cons` hingegen hat den Typ `a -> (List a) -> (List a)`. Wenn `Cons` das Einfügen eines neuen Elements in eine Liste repräsentiert, so steht `Cons 1 Nil` für die einelementige Liste mit dem Element `1`. Die folgenden Beispiele zeigen die Algorithmen `len` und `append` auf selbstdefinierten Listen und natürlichen Zahlen

```
len :: List a -> Nats
len Nil         = Zero
len (Cons x xs) = Succ (len xs)

append :: List a -> List a -> List a
append Nil         ys = ys
append (Cons x xs) ys = Cons x (append xs ys)
```

Wie bei Typabkürzungen müssen auch bei der Definition algebraischer Datentypen die Variablen $\underline{\mathsf{var}}_1, \ldots, \underline{\mathsf{var}}_n$ paarweise verschieden sein und der Typ $\underline{\mathsf{type}}$ auf der rechten Seite darf keine Variablen außer diesen enthalten. Anders als bei der Typabkürzung dürfen Datentypen aber rekursiv definiert sein. Dies ist z.B. bei `Nats` der Fall, da Objekte des Typs `Nats` mit Hilfe eines Konstruktors `Succ` gebildet werden, dessen Argumente wiederum vom Typ `Nats` sind. Analog verhält es sich bei `List a`.

Auch verschränkt rekursive Datentypen sind möglich. Im folgenden Beispiel wird der Typkonstruktor `Tree` mit Hilfe von `Forest` definiert und die Definition von `Forest` benötigt wiederum `Tree`. Hierbei realisiert `Tree` Vielwegbäume (d.h. Bäume mit beliebiger Anzahl von Kindern) und `Forest` repräsentiert Listen von Vielwegbäumen.

```
data Tree element = Node element (Forest element)

data Forest element = NoTrees | Trees (Tree element) (Forest element)
```

Typklassen

Eine *Typklasse* ist eine Menge von Typen. Ihre Elemente bezeichnet man als *Instanzen* der Typklasse. Die Namen der Funktionen auf diesen Instanz-Typen sind überladen, d.h., die Funktionen heißen in all diesen Typen gleich, sie können aber bei jedem Typ eine unterschiedliche Definition haben. Dies bezeichnet man als *Ad-hoc-Polymorphie*, da ad-hoc (beim Aufruf eines Funktionssymbols) aufgrund des Typs der Argumente entschieden wird, welche Funktion tatsächlich ausgeführt wird.

Die Funktionen für Gleichheit und Ungleichheit sind in HASKELL vordefiniert. Man würde zunächst erwarten, dass ihre Typdeklaration wie folgt lautet.

```
(==), (/=) :: a -> a -> Bool
```

Die Gleichheitsfunktion `==` ist zwar für viele Datentypen sinnvoll, jedoch nicht für alle (z.B. ist bei Funktionen die Gleichheit nicht entscheidbar). Hinzu kommt, dass die Implementierung von `==` für verschiedene Datentypen unterschiedlich ist. Beim Vergleich von zwei

Zeichenfolgen muss eine andere Operation durchgeführt werden als beim Vergleich von zwei Zahlen.

Die Typen, auf denen die Gleichheit `==` definiert ist, werden daher in einer Typklasse `Eq` zusammengefasst. Daher hat der Typ der beiden Funktionen `==` und `/=` einen sogenannten *Kontext* `Eq a`.

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

Dies bedeutet, dass für die Typvariable `a` nur Typen aus der Typklasse `Eq` eingesetzt werden dürfen.

Die Syntax zur Deklaration von Typklassen (wie `Eq`) ist wie folgt. Hierzu muss eine weitere Grammatikregel für das Nichtterminal $\underline{\textsf{topdecl}}$ eingeführt werden.

$$\begin{array}{lcll} \underline{\textsf{topdecl}} & \rightarrow & \texttt{class}\ \underline{\textsf{tyconstr}}\ \underline{\textsf{var}}\ [\texttt{where}\ \{\underline{\textsf{cdecl}}_1;\ldots;\underline{\textsf{cdecl}}_n\}], & \text{wobei } n \geq 1 \\ \underline{\textsf{cdecl}} & \rightarrow & \underline{\textsf{typedecl}} \mid \underline{\textsf{fundecl}} \mid \underline{\textsf{infixdecl}} \mid \underline{\textsf{var}}\ \underline{\textsf{rhs}} & \end{array}$$

Die Typklasse `Eq` ist in HASKELL vordefiniert. Ihre Definition könnte z.B. wie folgt lauten.

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

Die erste Zeile deklariert die Typklasse `Eq` mit dem Parameter `a`. Die zweite Zeile gibt den Typ der Operationen dieser Klasse an, wobei die Typvariable `a` implizit durch den Kontext `Eq a =>` eingeschränkt ist. Diese Operationen bezeichnet man auch als *Methoden* der Klasse. In der dritten Zeile wird eine Default-Implementierung von `/=` angegeben. In Instanzen dieser Klasse (d.h. in konkreten Typen, die Mitglied von `Eq` sind) können die Methoden der Klasse überschrieben werden. Nur wenn keine eigene Definition für `==` oder `/=` vorhanden ist, wird die Default-Implementierung verwendet. Dies bedeutet in unserem Beispiel, dass man bei den Instanzentypen jeweils nur die Definition von `==` angeben muss. Die Funktion `/=` berechnet dann automatisch die Ungleichheit auf korrekte Weise.

Die Typ- und Methodendeklarationen einer Klasse werden durch das Nichtterminalsymbol $\underline{\textsf{cdecl}}$ beschrieben. Man erkennt an der Grammatik, dass hier nahezu beliebige Deklarationen erlaubt sind. Die einzige Einschränkung im Vergleich zu $\underline{\textsf{decl}}$ ist, dass keine beliebigen Patterndeklarationen zugelassen sind, sondern nur Patterndeklarationen, bei denen auf der linken Seite nur eine Variable statt eines beliebigen Patterns steht.

Durch die Deklaration von `Eq` allein ist aber noch nicht klar, welche Typen zu der Typklasse `Eq` gehören. Hierzu verwendet man *Instanzendeklarationen*. Die zugehörigen Grammatikregeln lauten wie folgt.

$$\begin{array}{lcll} \underline{\textsf{topdecl}} & \rightarrow & \texttt{instance}\ \underline{\textsf{tyconstr}}\ \underline{\textsf{instype}}\ [\texttt{where}\ \{\underline{\textsf{idecl}}_1;\ldots;\underline{\textsf{idecl}}_n\}], & \text{wobei } n \geq 1 \\ & & & \\ \underline{\textsf{instype}} & \rightarrow & (\underline{\textsf{tyconstr}}\ \underline{\textsf{var}}_1\ \ldots\ \underline{\textsf{var}}_n), & \text{wobei } n \geq 0 \\ & & \mid\ [\underline{\textsf{var}}] & \\ & & \mid\ (\underline{\textsf{var}}_1\ \texttt{->}\ \underline{\textsf{var}}_2) & \\ & & \mid\ (\underline{\textsf{var}}_1,\ldots,\underline{\textsf{var}}_n), & \text{wobei } n \geq 2 \\ & & & \\ \underline{\textsf{idecl}} & \rightarrow & \underline{\textsf{fundecl}} \mid \underline{\textsf{var}}\ \underline{\textsf{rhs}} & \end{array}$$

Beispielsweise kann man deklarieren, dass `Int` ein Exemplar der Typklasse `Eq` ist.

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

Dies besagt zum einen, dass `Int` ein Mitglied der Typklasse `Eq` ist und zum anderen wird hierdurch eine Implementierung der Gleichheit `==` auf `Int` angegeben. Wäre in `Eq` bereits eine Default-Implementierung von `==` angegeben worden, so würde sie (bei Argumenten vom Typ `Int`) durch diese Implementierung überschrieben werden. Bei der Funktion `primEqInt` handelt es sich um eine primitive vordefinierte Funktion. Die Definition der Ungleichheit wird aus der Klassendeklaration abgeleitet (d.h., `x /= y` ist auf `Int` definiert als `not (primEqInt x y)`).

Wie aus der Grammatikregel für Instanzendeklarationen hervorgeht, kann man nur eingeschränkte Formen von Typen $\underline{\mathsf{instype}}$ als Instanzen von Typklassen deklarieren: Typkonstruktoren dürfen hier nur auf paarweise verschiedene Typvariablen und nicht auf beliebige Argumenttypen angewendet werden, Typtupel müssen mindestens zwei Komponenten haben und eine Typvariable allein kann kein Exemplar einer Typklasse sein.

Die Deklarationen $\underline{\mathsf{idecl}}$, um Methoden in Instanzendeklarationen zu implementieren, sind noch eingeschränkter als in der Klassendeklaration: Man kann keine Typdeklarationen (denn die Typen werden ja schon in der Klassendeklaration festgelegt), keine Infixdeklarationen und keine beliebigen Patterndeklarationen verwenden.

Generell kann man durch die Verwendung von Typklassen jetzt bei jedem Typ einen *Kontext* mit angeben. Dieser Kontext besagt, dass bestimmte Typvariablen nur mit Typen einer bestimmten Klasse instantiiert werden dürfen. Die hierfür benötigten (geänderten) Grammatikregeln sind folgende.

$$\begin{array}{lcll}
\underline{\mathsf{context}} & \rightarrow & (\underline{\mathsf{tyconstr}}_1\ \underline{\mathsf{var}}_1, \ldots, \underline{\mathsf{tyconstr}}_n\ \underline{\mathsf{var}}_n), & \text{wobei } n \geq 1 \\[1em]
\underline{\mathsf{typedecl}} & \rightarrow & \underline{\mathsf{var}}_1, \ldots, \underline{\mathsf{var}}_n :: [\underline{\mathsf{context}} \Rightarrow]\ \underline{\mathsf{type}}, & \text{wobei } n \geq 1 \\[1em]
\underline{\mathsf{topdecl}} & \rightarrow & \quad \underline{\mathsf{decl}} & \\
 & & \mid\ \texttt{type} \ldots & \\
 & & \mid\ \texttt{data}\ [\underline{\mathsf{context}} \Rightarrow]\ \underline{\mathsf{tyconstr}}\ \underline{\mathsf{var}}_1 \ldots \underline{\mathsf{var}}_n = \ldots & \\
 & & \mid\ \texttt{class}\ [\underline{\mathsf{context}} \Rightarrow]\ \underline{\mathsf{tyconstr}}\ \underline{\mathsf{var}} \ldots & \\
 & & \mid\ \texttt{instance}\ [\underline{\mathsf{context}} \Rightarrow]\ \underline{\mathsf{tyconstr}}\ \underline{\mathsf{instype}} \ldots &
\end{array}$$

Die Verwendung von Kontexten in Instanzendeklarationen ist bei Instanzendeklarationen für Typkonstruktoren mit Parametern sinnvoll. Die nächste Instanzendeklaration besagt folgendes: Falls Elemente vom Typ `a` auf ihre Gleichheit überprüft werden können, dann ist die Gleichheit auch bei Listen mit Elementen vom Typ `a` definiert. Der Typ `[a]` ist also in der Klasse `Eq`, falls auch der Typ `a` schon in der Klasse `Eq` ist. Um dies darzustellen, verwendet man den Kontext `Eq a =>` in der Instanzendeklaration.

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

Hierbei ist `&&` die vordefinierte Konjunktion auf booleschen Werten. In dem Ausdruck `x == y` auf der rechten Seite der zweiten definierenden Gleichung bezeichnet `==` die Gleichheit auf dem Typ `a`, wohingegen im zweiten Ausdruck `xs == ys` die Gleichheit auf dem Typ `[a]` gemeint ist (d.h., dies ist ein rekursiver Aufruf).

Zur Behandlung von Paaren sind Einschränkungen an zwei Typvariablen erforderlich.

```
instance (Eq a, Eq b) => Eq (a,b) where
    (x,y) == (x′,y′) = x == x′ && y == y′
```

Falls die Typen `a` und `b` Exemplare der Typklasse `Eq` sind, so ist auch der Typ `(a,b)` ein Exemplar der Typklasse `Eq`, wobei die Gleichheit komponentenweise definiert ist.

Man erkennt, dass auf diese Weise der Operator `==` in der Tat überladen ist, da er für verschiedene Datentypen unterschiedlich definiert ist. Eine besonders einfache Form der Instanzendeklaration besteht darin, dass man bei der Deklaration von algebraischen Datentypen (mittels `data`) `deriving`-Klauseln anhängt (wie z.B. `deriving Show` oder `deriving (Eq, Show)` etc.). Hierdurch wird festgelegt, dass der Typ eine Instanz der jeweiligen Typklassen sein soll und es wird automatisch eine Standardimplementierung für die Methoden der Klasse erzeugt.

Hierarchische Organisation von Typklassen

Durch Verwendung von Kontexten in Klassendeklarationen kann man Unterklassen zu bereits eingeführten Typklassen deklarieren. Beispielsweise ist in HASKELL eine Typklasse `Ord` vordefiniert, in der Methoden wie `<`, `>`, `<=`, `>=`, etc. zur Verfügung stehen. Offensichtlich gilt `x <= y && y <= x` genau dann, wenn `x == y` gilt. Das bedeutet, dass nur Typen der Klasse `Eq` auch in der Klasse `Ord` liegen können. `Ord` sollte daher eine *Unterklasse* von `Eq` sein. Man könnte die Klasse `Ord` daher wie folgt deklarieren.

```
class Eq a => Ord a where
   (<),(>) :: a -> a -> Bool
   x < y = x <= y && x /= y
           ...
```

In der ersten Zeile wird die Beziehung zwischen den Klassen `Eq` und `Ord` festgelegt: Nur wenn `a` zur Typklasse `Eq` gehört, kann `a` auch zur Typklasse `Ord` gehören. Danach folgen die Typdeklarationen der Methoden und ihre Default-Implementierungen.

Eine weitere vordefinierte Typklasse ist die Klasse `Show`, die diejenigen Typen enthält, deren Objekte auf dem Bildschirm angezeigt werden können. Für diese Objekte existiert also eine Methode `show`, die die Objekte in Strings wandelt, die dann ausgegeben werden können. Die Deklaration der Klasse `Show` könnte wie folgt lauten.

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

Bei Eingabe eines Ausdrucks wertet der Interpreter diesen zunächst aus. Anschließend versucht er, den resultierenden Wert mit Hilfe der Funktion `show` auf dem Bildschirm auszugeben. Dies ist aber nur möglich, wenn der Typ dieses Werts in der Klasse `Show` enthalten ist. Ansonsten erhält man eine Fehlermeldung.

Um auch Werte von benutzerdefinierten Datenstrukturen auf dem Bildschirm ausgeben zu können, muss man diese Datenstrukturen als Instanzen der Klasse `Show` deklarieren und eine geeignete Implementierung der Methode `show` angeben. (Durch die Klausel "`deriving Show`" ist dies natürlich möglich, aber dann muss man die automatisch erzeugte Implementierung von `show` für diesen Datentyp verwenden.) Um die benutzerdefinierte Datenstruktur

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

selbst als Instanz der Klasse `Show` zu deklarieren, kann man z.B. die folgende Instanzendeklaration schreiben:

```
instance Show a => Show (List a) where
         show Nil = "[]"
         show (Cons x xs) = show x ++  " : " ++ show xs
```

Gibt man nun `Cons 1 (Cons 2 Nil)` im Interpreter ein, so wird daraufhin `1 : 2 : []` auf dem Bildschirm ausgegeben.

Typklassen sind auch hilfreich bei der Überladung von arithmetischen Operatoren. Operatoren wie `+`, `-`, `*`, etc. sind sowohl auf ganzen Zahlen (`Int`) wie auf Gleitkommazahlen (`Float`) definiert. Die Typklasse, die die Zahlentypen und ihre Operationen zusammenfasst, ist in HASKELL die vordefinierte Klasse `Num`.

```
class (Eq a, Show a) => Num a where
                                    ...
```

Zahlen wie `0`, `1`, `2` etc. sind vom Typ `Num a => a`, d.h., `2` kann sowohl als ganze Zahl (`Int`) als auch als Gleitkommazahl (`Float`) verwendet werden.

Zusammenfassung der Syntax für Typen

Die Syntaxregeln für Typen, Typdefinitionen und Typklassen etc. lauten zusammenfassend wie folgt:

$$
\begin{array}{lcll}
\underline{\mathsf{type}} & \rightarrow & (\underline{\mathsf{tyconstr}}\ \underline{\mathsf{type}}_1\ \ldots\ \underline{\mathsf{type}}_n), & \text{wobei } n \geq 0 \\
 & | & [\underline{\mathsf{type}}] & \\
 & | & (\underline{\mathsf{type}}_1\ \mathsf{->}\ \underline{\mathsf{type}}_2) & \\
 & | & (\underline{\mathsf{type}}_1, \ldots, \underline{\mathsf{type}}_n), & \text{wobei } n \geq 0 \\
 & | & \underline{\mathsf{var}} & \\
\\
\underline{\mathsf{tyconstr}} & \rightarrow & \text{String von Buchstaben und Zahlen mit Großbuchstaben am Anfang} &
\end{array}
$$

$$
\begin{array}{lcll}
\underline{\mathsf{topdecl}} & \rightarrow & \underline{\mathsf{decl}} & \\
 & & \mid\ \texttt{type}\ \underline{\mathsf{tyconstr}}\ \underline{\mathsf{var}}_1 \ldots \underline{\mathsf{var}}_n = \underline{\mathsf{type}}, & \text{wobei } n \geq 0 \\
 & & \mid\ \texttt{data}\ [\underline{\mathsf{context}} \Rightarrow]\ \underline{\mathsf{tyconstr}}\ \underline{\mathsf{var}}_1 \ldots \underline{\mathsf{var}}_n = & \\
 & & \qquad \underline{\mathsf{constr}}_1\ \underline{\mathsf{type}}_{1,1} \ldots \underline{\mathsf{type}}_{1,n_1} & \\
 & & \qquad \mid \ldots & \\
 & & \qquad \mid \underline{\mathsf{constr}}_k\ \underline{\mathsf{type}}_{k,1} \ldots \underline{\mathsf{type}}_{k,n_k}, & \text{wobei } n \geq 0, k \geq 1, n_i \geq 0 \\
 & & \mid\ \texttt{class}\ [\underline{\mathsf{context}} \Rightarrow]\ \underline{\mathsf{tyconstr}}\ \underline{\mathsf{var}} & \\
 & & \qquad [\texttt{where}\ \{\underline{\mathsf{cdecl}}_1; \ldots; \underline{\mathsf{cdecl}}_n\}], & \text{wobei } n \geq 1 \\
 & & \mid\ \texttt{instance}\ [\underline{\mathsf{context}} \Rightarrow]\ \underline{\mathsf{tyconstr}}\ \underline{\mathsf{instype}} & \\
 & & \qquad [\texttt{where}\ \{\underline{\mathsf{idecl}}_1; \ldots; \underline{\mathsf{idecl}}_n\}], & \text{wobei } n \geq 1 \\
 & & & \\
\underline{\mathsf{cdecl}} & \rightarrow & \underline{\mathsf{typedecl}} \mid \underline{\mathsf{fundecl}} \mid \underline{\mathsf{infixdecl}} \mid \underline{\mathsf{var}}\ \underline{\mathsf{rhs}} & \\
 & & & \\
\underline{\mathsf{instype}} & \rightarrow & (\underline{\mathsf{tyconstr}}\ \underline{\mathsf{var}}_1\ \ldots\ \underline{\mathsf{var}}_n), & \text{wobei } n \geq 0 \\
 & & \mid\ [\underline{\mathsf{var}}] & \\
 & & \mid\ (\underline{\mathsf{var}}_1 \texttt{->} \underline{\mathsf{var}}_2) & \\
 & & \mid\ (\underline{\mathsf{var}}_1, \ldots, \underline{\mathsf{var}}_n), & \text{wobei } n \geq 2 \\
 & & & \\
\underline{\mathsf{idecl}} & \rightarrow & \underline{\mathsf{fundecl}} \mid \underline{\mathsf{var}}\ \underline{\mathsf{rhs}} & \\
 & & & \\
\underline{\mathsf{context}} & \rightarrow & (\underline{\mathsf{tyconstr}}_1\ \underline{\mathsf{var}}_1, \ldots, \underline{\mathsf{tyconstr}}_n\ \underline{\mathsf{var}}_n), & \text{wobei } n \geq 1 \\
 & & & \\
\underline{\mathsf{typedecl}} & \rightarrow & \underline{\mathsf{var}}_1, \ldots, \underline{\mathsf{var}}_n :: [\underline{\mathsf{context}} \Rightarrow]\ \underline{\mathsf{type}}, & \text{wobei } n \geq 1
\end{array}
$$

1.2 Funktionen höherer Ordnung

Funktionen höherer Ordnung ("higher-order functions" oder auch "Funktionale") sind dadurch charakterisiert, dass ihre Argumente oder ihr Resultat selbst wieder Funktionen sind. Beispielsweise ist `square :: Int -> Int` eine Funktion erster Ordnung, wohingegen `plus :: Int -> Int -> Int` eine Funktion höherer Ordnung ist, denn `plus 1` (das Resultat von `plus` bei Anwendung auf das Argument `1`) ist wieder eine Funktion. Zunächst betrachten wir einige typische Funktionen höherer Ordnung.

Die Funktionskomposition "."

Eine sehr oft verwendete Funktion höherer Ordnung ist die Funktionskomposition $(f \circ g)$. In HASKELL ist die Funktionskomposition für zwei einstellige Funktionen bereits als Infix-Operator `.` vordefiniert. Wenn `g` eine Funktion vom Typ `a -> b` und `f` eine Funktion vom Typ `b -> c` ist, so ist `f.g` die Funktion, die entsteht, indem man erst `g` und dann `f` anwendet.

```
infixr 9 .
(.) :: (b -> c) -> (a -> b) -> (a -> c)
f . g = \x -> f (g x)
```

Beispielsweise ist `half.square` die Funktion, die ein Argument x erst quadriert und dann halbiert (d.h., sie berechnet $\frac{x^2}{2}$). Beispielsweise ergibt die Auswertung von `(half.square) 4` das Ergebnis `8` $(= \frac{4^2}{2})$ und `((\x -> x+1).square) 5` ergibt `26`.

Die Funktionen `curry` und `uncurry`

Wie bereits erwähnt, bedeutet *Currying* die Überführung von Tupelargumenten in eine Folge von Argumenten. Beispielsweise geschieht der Schritt von der ersten der beiden folgenden Definitionen von `plus` zur zweiten durch Currying und der Schritt zurück geschieht durch Uncurrying.

```
plus :: (Int, Int) -> Int
plus (x,y) = x + y
```

bzw.

```
plus :: Int -> Int -> Int
plus x y = x + y
```

Im allgemeinen kann man diese Überführung durch (in HASKELL vordefinierte) Funktionen höherer Ordnung vornehmen.

```
curry :: ((a,b) -> c) -> a -> b -> c
curry f = g
      where g x y = f (x,y)
```

bzw.

```
uncurry :: (a -> b -> c) -> (a,b) -> c
uncurry g = f
        where f (x,y) = g x y
```

Es gilt `curry (uncurry g) = g` und `uncurry (curry f) = f`. Mit Hilfe dieser beiden Funktionen ist es nun möglich, eine Funktion beliebig als curried- oder uncurried-Variante zu verwenden. Die Auswertung von `uncurry (+) (1,2)` ergibt z.B. `3`.

Die Funktion `map`

Funktionen höherer Ordnung ermöglichen es insbesondere, Probleme und Programme übersichtlicher zu strukturieren. Hierzu verwendet man typischerweise eine Menge von klassischen, oft einsetzbaren Funktionen höherer Ordnung, die bestimmte Rekursionsmuster implementieren. Programme, die auf diese Weise erstellt werden, sind besser lesbar und einfacher wiederzuverwenden. Im folgenden sollen einige dieser klassischen Funktionen höherer Ordnung vorgestellt werden.

Betrachten wir eine Funktion `suclist`, die alle Zahlen in einer Liste von ganzen Zahlen um `1` erhöht.

```
suc :: Int -> Int
suc = plus 1

suclist :: [Int] -> [Int]
suclist []     = []
suclist (x:xs) = suc x : suclist xs
```

Bei einem Aufruf von `suclist` mit dem Argument

$$[\mathtt{x}_1, \mathtt{x}_2, \ldots, \mathtt{x}_n]$$

erhält man also das Ergebnis

$$[\mathtt{suc}\,\mathtt{x}_1,\ \mathtt{suc}\,\mathtt{x}_2,\ \ldots, \mathtt{suc}\,\mathtt{x}_n].$$

Analog dazu berechnet die Funktion `sqrtlist` die Wurzel für jedes Element einer Liste von Gleitkommazahlen.

```
sqrtlist :: [Float] -> [Float]
sqrtlist []     = []
sqrtlist (x:xs) = sqrt x : sqrtlist xs
```

Bei einem Aufruf von `sqrtlist` mit dem Argument

$$[\mathtt{x}_1, \mathtt{x}_2, \ldots, \mathtt{x}_n]$$

erhält man demnach das Ergebnis

$$[\mathtt{sqrt}\,\mathtt{x}_1,\ \mathtt{sqrt}\,\mathtt{x}_2,\ \ldots, \mathtt{sqrt}\,\mathtt{x}_n].$$

Man erkennt, dass die beiden Funktionen `suclist` und `sqrtlist` sehr ähnlich sind, da beide Funktionen eine Liste elementweise abarbeiten (durch die Funktionen `suc` und `sqrt`) und dann die verarbeitete Liste zurückgeben (d.h., die eigentliche Datenstruktur der Liste bleibt erhalten). Es liegt auf der Hand, diese beiden Funktionen durch ein und dieselbe Funktion zu realisieren. Dazu sind folgende Schritte notwendig:

- Abstraktion vom Datentyp der Listenelemente (`Int` bzw. `Float`). Dies ist nur in Sprachen mit (parametrischer) Polymorphie möglich.
- Abstraktion von der Funktion, die auf jedes Element der Liste angewandt werden soll (`suc` bzw. `sqrt`). Dies ist nur in Sprachen möglich, in denen Funktionen als gleichberechtigte Datenobjekte behandelt werden.

Allgemein wird also eine Funktion `g` auf alle Elemente der Liste angewandt. Man benötigt allgemein also eine Funktion `f`, so dass man beim Aufruf von `f` mit dem Argument

$$[\mathtt{x}_1, \mathtt{x}_2, \ldots, \mathtt{x}_n]$$

das folgende Ergebnis erhält.

$$[\mathtt{g}\,\mathtt{x}_1,\ \mathtt{g}\,\mathtt{x}_2,\ \ldots, \mathtt{g}\,\mathtt{x}_n]$$

Die allgemeine Form von Funktionen dieser Bauart ist demnach wie folgt.

```
f :: [a] -> [b]
f [] = []
f (x:xs) = g x : f xs
```

Da `g` eine beliebige Funktion ist, sollte sie zusätzliches Eingabeargument der Funktion sein. Auf diese Weise erhält man die Funktion `map` (wobei `f` von oben nun der Funktion `map g` entspricht).

```
map :: (a -> b) -> [a] -> [b]
map g [] = []
map g (x:xs) = g x : map g xs
```

Die Funktion `map` ist eine Funktion höherer Ordnung, da sowohl ihr Resultat als auch ihr Argument Funktionen sind. Sie ist in HASKELL bereits vordefiniert. Das Ergebnis von `map g` $[\mathtt{x}_1, \mathtt{x}_2, \ldots, \mathtt{x}_n]$ ist

$$[\mathtt{g}\,\mathtt{x}_1,\ \mathtt{g}\,\mathtt{x}_2,\ \ldots, \mathtt{g}\,\mathtt{x}_n].$$

Wir implementieren daher das Rekursionsmuster "Durchlaufe eine Liste und wende eine Funktion auf jedes Element an" mit einer eigenen Funktion `map`. Die Verwendung eines festen Satzes solcher Funktionen, die Rekursionsmuster realisieren, führt zu großer Modularisierbarkeit, Wiederverwendbarkeit und Lesbarkeit von Programmen.

Die weiter oben definierten Funktionen `suclist` und `sqrtlist` lassen sich nun wie folgt auf nicht-rekursive Weise definieren:

```
suclist  l = map suc  l
sqrtlist l = map sqrt l
```

oder noch einfacher als

```
suclist  = map suc
sqrtlist = map sqrt
```

Analog kann man entsprechende `map`-Funktionen auf anderen Datenstrukturen definieren, z.B. auf den selbst definierten Listen, die wie folgt definiert waren:

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

Hier lautet die entsprechende Definition wie folgt:

```
mapList :: (a -> b) -> List a -> List b
mapList g Nil         = Nil
mapList g (Cons x xs) = Cons (g x) (mapList g xs)
```

Als weiteres Beispiel betrachten wir die folgende Datenstruktur von Vielwegbäumen.

```
data Tree a = Node a [Tree a]
```

Die Funktion `map` lautet auf dieser Datenstruktur wie folgt.

```
mapTree :: (a -> b) -> Tree a -> Tree b
mapTree g (Node x ts) = Node (g x) (map (mapTree g) ts)
```

Beim Aufruf `mapTree g t` wird die Funktion `g` auf jeden Knoten des Baums `t` angewendet. Falls `t` der Baum

$$\texttt{Node}\ x_1\ \texttt{[Node}\ x_2\ \texttt{[]]}$$

ist, so ist `mapTree g t` der Baum

$$\texttt{Node}\ (g\,x_1)\ \texttt{[Node}\ (g\,x_2)\ \texttt{[]]}.$$

Um die Verwendung von `mapTree` zu verdeutlichen, implementieren wir nun eine Funktion `sucTree`, die alle Zahlen in einem Baum von ganzen Zahlen um `1` erhöht. Sie kann elegant mit Hilfe von `mapTree` formuliert werden.

```
sucTree :: Tree Int -> Tree Int
sucTree = mapTree suc
```

Generell gilt: `map`-Funktionen wenden eine Funktion `g` auf jeden Teilwert eines zusammengesetzten Datenobjekts an.

Die Funktion `zipWith`

Betrachten wir nun zwei weitere Funktionen `addlist` und `multlist`, die die Elemente zweier Listen durch Addition bzw. Multiplikation verknüpfen.

```
addlist :: Num a => [a] -> [a] -> [a]
addlist (x:xs) (y:ys) = (x + y) : addlist xs ys
addlist _      _      = []

multlist :: Num a => [a] -> [a] -> [a]
multlist (x:xs) (y:ys) = (x * y) : addlist xs ys
multlist _      _      = []
```

Da solche Funktionen wiederum sehr ähnlich implementiert werden, wollen wir nun die Funktion `zipWith` vorstellen, die das hierbei verwendete Rekursionsmuster realisiert. Diese (ebenfalls vordefinierte) Funktion arbeitet ähnlich wie `map`, sie wendet aber eine Funktion auf zwei Argumente an.

```
zipWith :: (a -> b -> c) -> [a] -> [b] -> [c]
zipWith f (x:xs) (y:ys)   = f x y : zipWith f xs ys
zipWith _ _      _        = []
```

Damit lässt sich nun das Rekursionsmuster für die Kombination von zwei Listen durch jeweiliges Anwenden einer Funktion auf Paare von Elementen beider Listen formulieren. Die nicht-rekursiven Definitionen von `addlist` und `multlist` lauten wie folgt.

```
addlist  = zipWith (+)
multlist = zipWith (*)
```

`filter`-Funktionen

Betrachten wir eine Funktion `dropEven`, die aus einer Liste von Zahlen alle geraden Zahlen löscht und eine Funktion `dropUpper`, die aus einer Liste von Zeichen alle Großbuchstaben löscht (die Hilfsfunktionen `odd` und `isLower` sind vordefiniert).

```
dropEven :: [Int] -> [Int]
dropEven [] = []
dropEven (x:xs) | odd x     = x : dropEven xs
                | otherwise = dropEven xs

dropUpper :: [Char] -> [Char]
dropUpper [] = []
dropUpper (x:xs) | isLower x = x : dropUpper xs
                 | otherwise = dropUpper xs
```

Beispielsweise ergibt `dropEven [1,2,3,4]` das Resultat `[1,3]` und `dropUpper "GmbH"` das Resultat `"mb"`.

Man erkennt, dass die beiden Funktionen `dropEven` und `dropUpper` sehr ähnlich sind, da beide eine Liste durchlaufen und dabei alle Elemente löschen, die eine bestimmte Bedingung (d.h., ein bestimmtes Prädikat) nicht erfüllen. Es liegt auf der Hand, diese beiden Funktionen durch ein und dieselbe Funktion zu realisieren. Dazu sind folgende Schritte notwendig:

- Abstraktion vom Datentyp der Listenelemente (`Int` bzw. `Char`). Dies ist wiederum nur in Sprachen mit (parametrischer) Polymorphie möglich.
- Abstraktion von dem Prädikat (d.h., der booleschen Funktion), mit der die Listenelemente gefiltert werden sollen (`dropEven` bzw. `isLower`). Dies ist nur in Sprachen möglich, in denen Funktionen als gleichberechtigte Datenobjekte behandelt werden.

Die allgemeine Form von Funktionen dieser Bauart ist demnach wie folgt (wobei `g` das Prädikat ist, mit dem gefiltert wird).

```
f :: [a] -> [a]
f [] = []
f (x:xs) | g x       = x : f xs
         | otherwise = f xs
```

Da `g` eine beliebige boolesche Funktion vom Typ `a -> Bool` ist, sollte sie zusätzliches Eingabeargument der Funktion sein. Auf diese Weise erhält man die Funktion `filter` (wobei `f` von oben nun der Funktion `filter g` entspricht). Sie ist in HASKELL bereits vordefiniert.

```
filter :: (a -> Bool) -> [a] -> [a]
filter g [] = []
filter g (x:xs) | g x       = x : filter g xs
                | otherwise = filter g xs
```

Die weiter oben definierten Funktionen `dropEven` und `dropUpper` lassen sich nun wie folgt auf nicht-rekursive Weise definieren:

```
dropEven  = filter odd
dropUpper = filter isLower
```

Analog kann man entsprechende `filter`-Funktionen auf eigenen Datenstrukturen definieren, z.B. auf den selbst definierten Listen. Generell gilt: `filter`-Funktionen löschen alle Teilwerte eines zusammengesetzten Datenobjekts, die die Bedingung `g` nicht erfüllen.

`fold`-Funktionen

Betrachten wir eine Funktion `add`, die alle Zahlen in einer Liste addiert und eine Funktion `prod`, die alle Zahlen in einer Liste multipliziert. Wir illustrieren dies zunächst an unserem selbstdefinierten Datentyp für Listen, da dies einfacher ist.

```
plus :: Int -> Int -> Int
plus x y = x + y

times :: Int -> Int -> Int
times x y = x * y

add :: (List Int) -> Int
add Nil         = 0
add (Cons x xs) = plus x (add xs)

prod :: (List Int) -> Int
prod Nil         = 1
prod (Cons x xs) = times x (prod xs)
```

Bei einem Aufruf von `add` mit dem Argument

$$\texttt{Cons}\,\texttt{x}_1\,(\texttt{Cons}\,\texttt{x}_2\,(\ldots\,(\texttt{Cons}\,\texttt{x}_{n-1}\,(\texttt{Cons}\,\texttt{x}_n\,\texttt{Nil}))\,\ldots))$$

erhält man das Ergebnis

$$\texttt{plus}\,\texttt{x}_1\,(\texttt{plus}\,\texttt{x}_2\,(\ldots\,(\texttt{plus}\,\texttt{x}_{n-1}\,(\texttt{plus}\,\texttt{x}_n\,\texttt{0}))\,\ldots)).$$

Der Konstruktor `Cons` wird also durch die Funktion `plus` ersetzt und der Konstruktor `Nil` durch die Zahl `0`. Analog erhält man beim Aufruf von `prod` mit demselben Argument das Resultat

$$\texttt{times}\,\texttt{x}_1\,(\texttt{times}\,\texttt{x}_2\,(\ldots\,(\texttt{times}\,\texttt{x}_{n-1}\,(\texttt{times}\,\texttt{x}_n\,\texttt{1}))\,\ldots)).$$

Hier wird der Konstruktor `Cons` also durch `times` und der Konstruktor `Nil` durch `1` ersetzt.

Wiederum ist unser Ziel, eine Funktion höherer Ordnung anzugeben, die das Rekursionsmuster dieser beiden Funktionen implementiert. Diese Funktion kann dann zur Implementierung von `add` und `prod` wiederverwendet werden.

Allgemein wird also der Konstruktor `Cons` durch eine Funktion `g` und der Konstruktor `Nil` durch einen Initialwert `e` ersetzt. Man benötigt also eine Funktion `f`, so dass man beim Aufruf von `f` mit dem Argument

$$\texttt{Cons}\,\texttt{x}_1\,(\texttt{Cons}\,\texttt{x}_2\,(\ldots(\texttt{Cons}\,\texttt{x}_{n-1}\,(\texttt{Cons}\,\texttt{x}_n\,\texttt{Nil}))\ldots))$$

das folgende Ergebnis erhält.

$$\texttt{g}\,\texttt{x}_1\,(\texttt{g}\,\texttt{x}_2\,(\ldots(\texttt{g}\,\texttt{x}_{n-1}\,(\texttt{g}\,\texttt{x}_n\,\texttt{e}))\ldots))$$

Wiederum muss man vom Typ der Listenelemente und von den Funktionen `g` und `e` abstrahieren. Die allgemeine Form von Funktionen dieser Bauart ist demnach wie folgt.

```
f :: (List a) -> b
f Nil         = e
f (Cons x xs) = g x (f xs)
```

Hierbei ist der Initialwert `e` vom Typ `b` des Ergebnisses und die Hilfsfunktion `g` muss den Typ `a -> b -> b` haben. Da `e` und `g` beliebige Funktionen sind, sollten sie zusätzliche Eingabeargumente der Funktion sein. Auf diese Weise erhält man die Funktion `fold` (wobei `f` von oben nun der Funktion `fold g e` entspricht).

```
fold :: (a -> b -> b) -> b -> (List a) -> b
fold g e Nil         = e
fold g e (Cons x xs) = g x (fold g e xs)
```

Das Ergebnis von

$$\texttt{fold}\,\texttt{g}\,\texttt{e}\,(\texttt{Cons}\,\texttt{x}_1\,(\texttt{Cons}\,\texttt{x}_2\,(\ldots(\texttt{Cons}\,\texttt{x}_{n-1}\,(\texttt{Cons}\,\texttt{x}_n\,\texttt{Nil}))\ldots)))$$

ist also

$$\texttt{g}\,\texttt{x}_1\,(\texttt{g}\,\texttt{x}_2\,(\ldots(\texttt{g}\,\texttt{x}_{n-1}\,(\texttt{g}\,\texttt{x}_n\,\texttt{e}))\ldots)).$$

Damit sind nun neue (nicht-rekursive) Definitionen von `add` und `prod` möglich.

```
add  = fold plus  0
prod = fold times 1
```

Ein weiteres Beispiel ist die Funktion `conc`, die eine Liste von Listen als Eingabe erhält und als Ergebnis die Konkatenation aller dieser Listen liefert. Hierbei verwenden wir den folgenden Algorithmus `append`.

```
append :: List a -> List a -> List a
append Nil         ys  = ys
append (Cons x xs) ys  = Cons x (append xs ys)
```

Bei einem Aufruf von `conc` mit dem Argument

$$\texttt{Cons}\,\texttt{l}_1\,(\texttt{Cons}\,\texttt{l}_2\,(\ldots(\texttt{Cons}\,\texttt{l}_{n-1}\,(\texttt{Cons}\,\texttt{l}_n\,\texttt{Nil}))\ldots))$$

erhält man das Ergebnis

$$\texttt{append}\,\texttt{l}_1\,(\texttt{append}\,\texttt{l}_2\,(\ldots(\texttt{append}\,\texttt{l}_{n-1}\,(\texttt{append}\,\texttt{l}_n\,\texttt{Nil}))\ldots)).$$

Die Implementierung von `conc` ist mit Hilfe des durch `fold` realisierten Rekursionsmusters leicht möglich.

```
conc :: List (List a) -> List a
conc = fold append Nil
```

Eine analoge Version zu `fold` auf den vordefinierten Listen ist in HASKELL vordefiniert. Sie heißt `foldr` und ist wie folgt definiert:

```
foldr :: (a -> b -> b) -> b -> [a] -> b
foldr g e []     = e
foldr g e (x:xs) = g x (foldr g e xs)
```

Realisiert man z.B. `add`, `prod`, und `conc` mit den vordefinierten Listen, so ist also `foldr` statt `fold` zu verwenden. Die Funktion `add` ist unter dem Namen `sum` und `conc` ist unter dem Namen `concat` bereits in HASKELL vordefiniert.

```
add :: [Int] -> Int
add  = foldr plus  0

prod :: [Int] -> Int
prod = foldr times 1

conc :: [[a]] -> [a]
conc = foldr (++) []
```

Betrachten wir nun auch noch `fold` auf der Datenstruktur der Vielwegbäume.

```
data Tree a = Node a [Tree a]
```

Die folgende Funktion ersetzt alle Vorkommen des Konstruktors `Node` durch die Funktion `g`.

```
foldTree :: (a -> [b] -> b) -> Tree a -> b
foldTree g (Node x ts) = g x (map (foldTree g) ts)
```

Falls `t` der Baum

$$\texttt{Node}\ \texttt{x}_1\ \texttt{[Node}\ \texttt{x}_2\ \texttt{[]]}$$

ist, so ist `fold g t` also

$$\texttt{g}\ \texttt{x}_1\ \texttt{[g}\ \texttt{x}_2\ \texttt{[]]}.$$

Um die Verwendung von `foldTree` zu verdeutlichen, implementieren wir die Funktion `addTree`, die alle Zahlen in den Knoten eines Baums addiert. Man benötigt hierzu noch eine Funktion `addtolist`, wobei $\texttt{addtolist}\ \texttt{x}\ [\texttt{y}_1, \ldots, \texttt{y}_n] = \texttt{y}_1\texttt{+}(\texttt{y}_2\texttt{+} \ldots \texttt{+}(\texttt{y}_n\texttt{+x}) \ldots)$ ist.

```
addtolist :: Num a => a -> [a] -> a
addtolist = foldr (+)

addTree :: Num a => Tree a -> a
addTree = foldTree addtolist
```

Man erhält demnach `addTree (Node 1 [Node 2 []])` $=$ `addtolist 1 [addtolist 2 []]` $=$ 3.

Generell gilt also: `fold`-Funktionen ersetzen die Konstruktoren einer Datenstruktur durch anzugebende Funktionen `g`, `e`, etc.

Listenkomprehension

In der Mathematik benutzt man häufig Mengenschreibweisen wie $\{x * x \mid x \in \{1, \ldots, 5\}, odd(x)\}$. An diese Schreibweise angelehnt, bieten funktionale Sprachen wie HASKELL eine alternative Listenschreibweise (sogenannte *Listenkomprehensionen*), um Berechnungen mit Hilfe von `map` elegant auszudrücken. Der Ausdruck

```
[ x * x | x <- [1 .. 5], odd x]
```

wertet in HASKELL zu der Liste `[1,9,25]` aus. Hierbei ist `[a .. b]` allgemein eine Kurzschreibweise für die Liste `[a,a+1,...,b]`. Dann bedeutet der obige Ausdruck die Liste aller Quadratzahlen `x * x`, wobei `x` alle Zahlen von `1` bis `5` durchläuft, wir aber nur die ungeraden Zahlen davon betrachten.

Formal hat eine Listenkomprehension die Gestalt $\texttt{[}\underline{\mathsf{exp}}\texttt{|}\underline{\mathsf{qual}}_1\texttt{,...,}\underline{\mathsf{qual}}_n\texttt{]}$, wobei $\underline{\mathsf{exp}}$ ein Ausdruck ist und die $\underline{\mathsf{qual}}_i$ sogenannte *Qualifikatoren* sind. Qualifikatoren teilen sich auf in *Generatoren* und *Einschränkungen* (Guards). Ein Generator hat die Form $\underline{\mathsf{var}}$ `<-` $\underline{\mathsf{exp}}$, wobei $\underline{\mathsf{exp}}$ ein Ausdruck von einem Listentyp ist. Ein Beispiel für einen Generator ist `x <- [1 .. 5]`. Die Bedeutung hiervon ist, dass die Variable $\underline{\mathsf{var}}$ alle Werte aus der Liste $\underline{\mathsf{exp}}$ annehmen kann. Eine Einschränkung ist ein boolescher Ausdruck wie `odd x`, der die möglichen Werte der im Ausdruck vorkommenden Variablen einschränkt. Wir müssen also die Grammatikregeln zur Definition von Ausdrücken um folgende Regeln erweitern:[2]

$$\underline{\mathsf{exp}} \;\rightarrow\; \texttt{[}\,\underline{\mathsf{exp}}\ \texttt{|}\ \underline{\mathsf{qual}}_1, \ldots, \underline{\mathsf{qual}}_n\,\texttt{]}, \quad \text{wobei } n \geq 1$$

$$\underline{\mathsf{qual}} \;\rightarrow\; \underline{\mathsf{var}}\ \texttt{<-}\ \underline{\mathsf{exp}} \mid \underline{\mathsf{exp}}$$

Die Bedeutung von Listenkomprehensionen ist wie folgt als Abkürzung für einen Ausdruck mit der Funktion `map` definiert. Hierbei ist `concat` wieder die Funktion, die eine Liste von Listen zu einer einzigen Liste verschmilzt. `Q` steht für eine Liste von Qualifikatoren. Falls diese leer ist, so ist $\texttt{[}\underline{\mathsf{exp}}\texttt{|Q]}$ als $\texttt{[}\underline{\mathsf{exp}}\texttt{]}$ zu lesen.

$$\begin{array}{lcl}
\texttt{[}\underline{\mathsf{exp}}\texttt{ | }\underline{\mathsf{var}}\texttt{ <- }\underline{\mathsf{exp}}'\texttt{, Q]} & = & \texttt{concat (map f }\underline{\mathsf{exp}}'\texttt{) where f }\underline{\mathsf{var}}\texttt{ = [}\underline{\mathsf{exp}}\texttt{|Q]}\\
\texttt{[}\underline{\mathsf{exp}}\texttt{ | }\underline{\mathsf{exp}}'\texttt{, Q]} & = & \texttt{if }\underline{\mathsf{exp}}'\texttt{ then [}\underline{\mathsf{exp}}\texttt{|Q] else []}
\end{array}$$

Die erste Regel heißt *Generatorregel* und die zweite bezeichnet man als *Einschränkungsregel.* Die Qualifikatoren werden also der Reihe nach abgearbeitet. Dabei bedeutet die Generatorregel:

$$\begin{array}{cl}
 & \texttt{[}\underline{\mathsf{exp}}\texttt{ | }\underline{\mathsf{var}}\texttt{ <- [a}_1\texttt{,...,a}_n\texttt{], Q]}\\
= & \texttt{concat (map f [a}_1\texttt{,...,a}_n\texttt{]) where f }\underline{\mathsf{var}}\texttt{ = [}\underline{\mathsf{exp}}\texttt{|Q]}\\
= & \texttt{f a}_1\texttt{ ++...++ f a}_n\texttt{ where f }\underline{\mathsf{var}}\texttt{ = [}\underline{\mathsf{exp}}\texttt{|Q]}\\
= & \texttt{[}\underline{\mathsf{exp}}\texttt{ | Q] [x/a}_1\texttt{] ++...++ [}\underline{\mathsf{exp}}\texttt{ | Q] [x/a}_n\texttt{]}.
\end{array}$$

Man bildet also $\texttt{[}\underline{\mathsf{exp}}\texttt{ | Q]}$, wobei $\underline{\mathsf{var}}$ alle Werte aus $\texttt{[a}_1\texttt{,...,a}_n\texttt{]}$ durchläuft.

[2]In HASKELL sind sogar Patterns statt nur Variablen in Generatoren möglich und darüber hinaus sind auch lokale Deklarationen in Qualifikatoren erlaubt.

Mit den obigen Regeln kann nun der Ausdruck `[x * x | x <- [1 .. 5], odd x]` ausgewertet werden:

```
    [x * x | x <- [1 .. 5], odd x]
 =  concat (map f [1 .. 5]) where f x = [x * x | odd x]
 =  concat [f 1, f 2, f 3, f 4, f 5] where f x = [x * x | odd x]
 =  f 1 ++ f 2 ++ f 3 ++ f 4 ++ f 5 where f x = [x * x | odd x]
 =  f 1 ++ f 2 ++ f 3 ++ f 4 ++ f 5 where f x = if odd x then [x * x] else []
 =  [1] ++ [] ++ [9] ++ [] ++ [25]
 =  [1,9,25]
```

Die folgenden Beispiele sollen verdeutlichen, dass die Reihenfolge von Qualifikatoren eine Rolle spielt. So wertet `[(a, b) | a <- [1 .. 3], b <- [1 .. 2]]` zu

```
[(1,1), (1,2), (2,1), (2,2), (3,1), (3,2)]
```

aus, wohingegen `[(a, b) | b <- [1 .. 2], a <- [1 .. 3]]` zu

```
[(1,1), (2,1), (3,1), (1,2), (2,2), (3,2)]
```

auswertet. Spätere Qualifikatoren können von vorher eingeführten Variablen abhängen. So wertet `[(a,b) | a <- [1 .. 4], b <- [a+1 .. 4]]` zu

```
[(1,2), (1,3), (1,4), (2,3), (2,4), (3,4)]
```

aus. Generatoren und Einschränkungen können in beliebiger Reihenfolge auftreten. Die Auswertung von `[(a,b) | a <- [1 .. 4], even a, b <-[a+1 .. 4], odd b]` ergibt daher nur `[(2,3)]`.

Auch die Funktion `map` lässt sich mit Listenkomprehension wie folgt definieren.

```
map :: (a -> b) -> [a] -> [b]
map f xs = [f x | x <- xs]
```

Als letztes Beispiel zeigen wir, wie man den bekannten *Quicksort*-Algorithmus mit Hilfe von Listenkomprehension auf äußerst einfache Weise implementieren kann.

```
qsort :: Ord a => [a] -> [a]
qsort []     = []
qsort (x:xs) = qsort l1 ++ [x] ++ qsort l2
               where l1 = [y | y <- xs, y <  x]
                     l2 = [y | y <- xs, y >= x]
```

Wenn man dies mit der entsprechenden Implementierung in einer imperativen Sprache vergleicht, wird deutlich, dass funktionale Programme oftmals wirklich deutlich kürzer und lesbarer sind. Eine Implementierung in JAVA lautet beispielsweise wie folgt.[3]

[3] Um ein Array `a` zu sortieren, muss man hierbei `qsort(a,0,a.length-1)` aufrufen.

```
static void qsort(int[] a, int lo, int hi) {
      int h, l, p, t;

      if (lo <= hi) {
         l = lo;
         h = hi;
         p = a[hi];

         do {

           while ((l < h) && (a[l] <= p))
                 l = l+1;

           while ((h > l) && (a[h] >= p))
                 h = h-1;


           if (l < h) {
              t = a[l];
              a[l] = a[h];
              a[h] = t;
           }

         } while (l < h);

         t = a[l];
         a[l] = a[hi];
         a[hi] = t;

         qsort( a, lo, l-1);
         qsort( a, l+1, hi);
      }
}
```

1.3 Programmieren mit Lazy Evaluation

Die Programmiersprache HASKELL verwendet eine nicht-strikte Auswertungsstrategie:

- Generell findet die Auswertung mit der leftmost outermost Strategie statt.
- Vordefinierte arithmetische Operatoren und Vergleichsoperatoren erfordern jedoch zunächst die Auswertung ihrer Argumente.
- Beim Pattern Matching werden die Argumente nur so weit ausgewertet, bis entschieden werden kann, welcher Pattern matcht.

Diese Punkte werden durch folgendes Beispiel verdeutlicht.

```
infinity :: Int
infinity = infinity + 1

mult :: Int -> Int -> Int
mult 0     y = 0
mult (x+1) y = y + mult x y
```

Die Auswertung von `mult 0 infinity` terminiert mit dem Ergebnis `0`. Hingegen führt die Auswertung des Ausdrucks `0 * infinity` zur Nicht-Terminierung. Hierdurch wird der Unterschied zwischen der generellen nicht-strikten Auswertung (bei `mult`) und der Auswertung bei vordefinierten Operatoren (wie `*`) deutlich. Man erkennt auch, dass bereits ohne Auswertung des Arguments `infinity` festgestellt werden kann, dass die Patterns der ersten definierenden Gleichung von `mult` auf die Argumente `0` und `infinity` matchen.

In HASKELL ist die Definition von unendlichen Datenobjekten möglich. Betrachten wir hierzu den folgenden Algorithmus.

```
from :: Num a => a -> [a]
from x = x : from (x+1)
```

Der Ausdruck `from x` entspricht der unendlichen Liste `[x, x+1, x+2, ...]`. Sie kann in HASKELL auch als `[x ..]` geschrieben werden. Obwohl die Auswertung des Ausdrucks `from 5` natürlich nicht terminiert, können solche unendlichen Listen dennoch sehr nützlich für die Programmierung sein. In HASKELL ist eine Funktion `take` vordefiniert, die das erste Teilstück einer Liste zurückliefert. Es gilt also $\mathtt{take\ n\ [x_1,\ldots,x_n,x_{n+1},\ldots]} = \mathtt{[x_1,\ldots,x_n]}$.

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

Da beim Pattern Matching das Argument immer nur so weit wie nötig ausgewertet wird, ergibt sich[4]

```
    take 2 (from 5)
=   take 2 (5 : from 6)
=   5 : take 1 (from 6)
=   5 : take 1 (6 : from 7)
=   5 : 6 : take 0 (from 7)
=   5 : 6 : []
=   [5,6].
```

[4] Hierbei steht "6" abkürzend für "`5 + 1`" und "7" abkürzend für "`5 + 1 + 1`".

Die Auswertung von `take 2 (from 5)` terminiert also. Funktionen, die niemals definiert sind, falls die Auswertung eines ihrer Argumente undefiniert ist, heißen *strikt*. Beispiele hierfür sind also arithmetische Grundoperationen und Vergleichsoperationen. Die Funktionen `mult` und `take` hingegen sind *nicht-strikt*. Man unterscheidet auch manchmal Striktheit für die verschiedenen Argumente. Dann sind `mult` und `take` strikt auf ihrem ersten Argument, aber nicht-strikt auf ihrem zweiten Argument.

Programmieren mit unendlichen Datenobjekten

Das generelle Vorgehen beim Programmieren mit unendlichen Datenobjekten ist wie folgt: Man erzeugt zuerst eine potentiell unendliche Liste von Approximationen an die Lösung. Anschließend filtert man daraus die wirklich gesuchte Lösung heraus.

Als Beispiel betrachten wir die Programmierung des *Sieb des Eratosthenes* zur Bestimmung von Primzahlen. Der Algorithmus arbeitet wie folgt:

1. Erstelle die Liste aller natürlichen Zahlen beginnend mit `2`.
2. Markiere die erste unmarkierte Zahl in der Liste.
3. Streiche alle Vielfachen der letzten markierten Zahl.
4. Gehe zurück zu Schritt 2.

Man beginnt also mit der Liste `[2,3,4,...]`. Wenn wir das Markieren durch Unterstreichung deutlich machen, markiert man nun die erste (unmarkierte) Zahl `2`. Dies ergibt die Liste [$\underline{2}$,3,4,...]. Nun werden alle Vielfachen der letzten markierten Zahl (d.h. `2`) gestrichen. Dies führt zu der Liste [$\underline{2}$,3,5,7,9,11,...]. Nun wird die nächste unmarkierte Zahl in der Liste markiert, was [$\underline{2}$,$\underline{3}$,5,7,9,11,...] ergibt. Anschließend werden die Vielfachen der letzten markierten Zahl (d.h. `3`) gestrichen. Dies ergibt [$\underline{2}$,$\underline{3}$,5,7,11,13,17,...]. Man erkennt, dass im Endeffekt nur die Liste der Primzahlen übrig bleibt.

Diese natürlichsprachliche Beschreibung des Vorgehens kann man bei der Verwendung unendlicher Datenobjekte und der nicht-strikten Auswertung nahezu direkt in Programmcode überführen. Hierbei muss man natürlich mit unendlichen Listen umgehen (denn in der Tat entsteht dabei ja die unendliche Liste aller Primzahlen). Wenn man aber nur an den ersten 100 Primzahlen oder allen Primzahlen kleiner als `42` interessiert ist, so muss das Sieben nur für ein endliches Anfangsstück der natürlichen Zahlen durchgeführt werden.

Die Implementierung von Schritt 1 ist naheliegend, da `from 2` oder `[2 ..]` die unendliche Liste der natürlichen Zahlen ab `2` berechnet. Zum Streichen von Vielfachen einer Zahl `x` aus einer Liste `xs` benutzen wir die folgende Funktion. Sie berechnet alle Elemente `y` aus `xs`, die nicht durch `x` teilbar sind. Hierbei ist `y` durch `x` teilbar, falls sich bei der Ganzzahldivision kein Rest `` y `mod` x `` ergibt.

```
drop_mult :: Int -> [Int] -> [Int]
drop_mult x xs = [y | y <- xs, y `mod` x /= 0]
```

Der Ausdruck `drop_mult 2 [3 ..]` berechnet also die unendliche Liste `[3,5,7,9,11,...]`.

Um wiederholt alle Vielfachen der ersten unmarkierten Zahl zu löschen, verwenden wir die Funktion `dropall`. Sie löscht zunächst alle Vielfachen des ersten Elements aus dem Rest einer Liste. Anschließend ruft sie sich rekursiv auf der entstehenden Liste ohne ihr erstes Element auf. Nun werden also die Vielfachen des zweiten Elements gelöscht, etc.

```
dropall :: [Int] -> [Int]
dropall (x:xs) = x : dropall (drop_mult x xs)
```

Die Liste `primes` aller Primzahlen kann dann wie folgt berechnet werden.

```
primes :: [Int]
primes = dropall [2 ..]
```

Bei der Auswertung von `primes` ergibt sich also die unendliche Liste

`[2,3,5,7,11,13,17,19,23,29,31,37,41,43,...]`.

Die Liste der ersten 100 Primzahlen berechnet der Ausdruck `take 100 primes`. Man erkennt also, dass man tatsächlich mit unendlichen Datenobjekten rechnen kann, falls während der Rechnung immer nur ein endlicher Teil dieser Objekte betrachtet wird.

Um die Liste aller Primzahlen zu berechnen, die kleiner als `42` sind, ist der Ausdruck `[ x | x <- primes, x < 42]` nicht geeignet. Dessen Auswertung terminiert nämlich nicht, da die Bedingung `x < 42` auf allen (unendlich vielen) Elementen von `primes` getestet werden muss. (Der Auswerter kann ja nicht wissen, dass es sich um eine monoton steigende Folge handelt.) Stattdessen sollte man die in HASKELL vordefinierte Funktion `takeWhile` verwenden.

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

Die Auswertung von `takeWhile` hält an, sobald für ein Listenelement die Bedingung `p` nicht mehr zutrifft. Die Auswertung des Ausdrucks `takeWhile (< 42) primes` terminiert daher und liefert in der Tat die Liste `[2,3,5,7,11,13,17,19,23,29,31,37,41]`.

Zirkuläre Datenobjekte

Um die Effizienz beim Rechnen mit unendlichen Datenobjekten zu erhöhen, sollte man versuchen, solche Datenobjekte (falls möglich) auf zirkuläre Weise im Speicher zu repräsentieren. Das einfachste Beispiel ist die unendliche Liste `ones` von der Form `[1,1,1,1,...]`.

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

Wenn eine Nicht-Funktions-Variable wie `ones` in ihrer eigenen Definition auftritt, wird das entstehende Datenobjekt als zyklisches Objekt wie in Abb. 1.2 gespeichert. Der Vorteil solcher Objekte liegt in ihrem geringen Speicherbedarf und darin, dass (teilweise) Berechnungen dadurch nur einmal statt mehrmals durchgeführt werden.

Um diesen Effizienzgewinn zu illustrieren, betrachten wir das Hamming-Problem (nach dem Mathematiker W. R. Hamming), das sich mit Hilfe von unendlichen (und zirkulären) Datenobjekten sehr effizient lösen lässt. Die Aufgabe besteht darin, eine Liste mit folgenden Eigenschaften zu generieren:

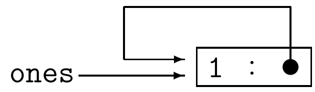

Abbildung 1.2: Zirkuläres Datenobjekt

- Die Liste ist aufsteigend sortiert und es gibt keine Duplikate.
- Die Liste beginnt mit `1`.
- Wenn die Liste das Element x enthält, dann enthält sie auch die Elemente $2x$, $3x$ und $5x$.
- Außer diesen Zahlen enthält die Liste keine weiteren Elemente.

Die Liste hat also folgende Gestalt.

```
[1,2,3,4,5,6,8,9,10,12,15,16,...]
```

Das Hamming-Problem wird oft verwendet, um Programmiersprachen auf ihre Eignung zur effizienten Implementierung bestimmter Klassen von Algorithmen zu untersuchen.

Die Idee für eine effiziente Implementierung dieses Problems ist, eine Funktion `mer` (für "merge") zu verwenden, die zwei (potentiell unendliche) geordnete Listen zu einer einzigen geordneten Liste ohne Duplikate verschmilzt.

```
mer Ord a => [a] -> [a] -> [a]
mer (x : xs) (y: ys) | x < y     = x : mer xs (y:ys)
                     | x == y    = x : mer xs ys
                     | otherwise = y : mer (x:xs) ys
```

Die Funktion `hamming` lässt sich nun wie folgt definieren:

```
hamming :: [Int]
hamming = 1 : mer (map (2*) hamming)
                  (mer (map (3*) hamming)
                       (map (5*) hamming))
```

Zu Anfang wird `hamming` durch ein zyklisches Objekt repräsentiert, in dem die drei Vorkommen von `hamming` auf der rechten Seite wieder durch Zeiger auf den Gesamtausdruck realisiert sind. Es ist eine gute Übung, einige Schritte der Auswertung von `hamming` auf diesem zyklischen Objekt nachzuverfolgen. Man erkennt, dass die Berechnung eines neuen Listenelements von `hamming` höchstens drei Multiplikationen (um die ersten Elemente von `(map (2*) hamming)`, `(map (3*) hamming)` und `(map (5*) hamming)` zu berechnen) und vier Vergleiche (für `<` und `==` in beiden `mer`-Aufrufen) benötigt. Die Berechnungszeit ist also *linear* in der Anzahl der benötigten Elemente von `hamming`. Das heißt, die Komplexität der Berechnung von `take n hamming` ist $O(\mathtt{n})$. Falls möglich, so sollte man daher immer versuchen, Nicht-Funktions-Variablen, die unendliche Datenobjekte realisieren, so zu definieren, dass diese Variablen auf der rechten Seite ihrer Definition wieder auftreten. Wie erwähnt, kann man auf diese Weise zyklische Datenobjekte erzeugen. Zusammenfassend erkennt man, dass unendliche Datenstrukturen sowohl für die Effizienz als auch für die Klarheit des Programmcodes sehr hilfreich sein können.