TECHNISCHE UNIVERSITÄT CAROLO-WILHELMINA ZU BRAUNSCHWEIG

Übungsskript

Eine Einführung in die funktionale Programmierung mit Haskell

Jan Oliver Ringert,
überarbeitet von
Henning Basold und
Christoph Szeppek

Wintersemester 2009/10

Institut für Programmierung und Reaktive Systeme
Werner Struckmann

Vorwort

Dieses Dokument bietet eine Einführung in die funktionale Programmierung am Beispiel von Haskell. Es orientiert sich hauptsächlich an [3]. Zusätzlich wurde Material aus einer Vorlesung über funktionale Programmierung in Scheme von Dr. W. Struckmann übernommen.

Geeignete Einführungen in Haskell 98 stellen auch [9] und [11] dar. Eine umfassende Referenz findet sich in dem überarbeiteten Haskell 98 Report [5], welcher zur Zeit Grundlage für die Sprache Haskell ist.

Inhaltsverzeichnis

1 Zur Geschichte **1**

1.1 Haskell Brooks Curry 1
1.2 Entwicklung funktionaler Programmiersprachen 1
1.2.1 LISP 2
1.2.2 ML 2
1.2.3 Lazy Evaluation 2
1.2.4 Funktionale Programmiersprachen geraten in Mode 3
1.3 Die Geburt von Haskell 3
1.3.1 Das Komitee 3
1.3.2 Der Name 4
1.3.3 Die Entwicklung 4

2 Einführung und Grundlagen **6**

2.1 Die Arbeitsumgebung 6
2.2 Typen 7
2.3 Funktionen 8
2.4 Verzweigungen 8
2.5 Variablen 9
2.6 Tupel und Listen 9
2.6.1 Tupel 10
2.6.2 Listen 10

3 Funktionen **12**

3.1 Rekursion 12
3.1.1 Summe der natürlichen Zahlen 12
3.1.2 Rekursionsschritt und Basis 12
3.2 Mustervergleich (pattern matching) 13
3.3 Polymorphie 13
3.3.1 Parametrische Polymorphie in Haskell 14
3.3.2 Ad-Hoc-Polymorphie 15
3.4 Typklassen 15
3.5 Lokale Variablen 16
3.6 Praxisbeispiel: Listen im und um den Supermarkt 18
3.7 Funktionen höherer Ordnung 18
3.8 Anonyme Funktionen – Lambda Abstraktion 19

3.9 Praxisbeispiel: Numerisches Differenzieren 20
3.10 Listenkomprehension 22

4 Benutzerdefinierte Typen **23**
4.1 Parametrische Typsynonyme 23
4.2 Algebraische Sicht der Typdefinition 24
4.3 Aufzählungstypen 24
4.4 Parametrisierte Alternativen 25
4.5 Rekursive Typen 26
4.6 Typklassen für eigene Typen 27

5 Module **29**
5.1 Export 30
5.2 Import 31
5.3 Namen und Geltungsbereich 32

6 Ein-/Ausgabe **34**
6.1 Einleitung 34
6.2 Ablaufsteuerung und Auswertungsreihenfolge 35
6.3 I/O-Funktionen 36
6.4 Datei-Ein-/Ausgabe 37
6.5 Werte darstellen und lesen 39
6.6 Ausführbare Programme erzeugen 39
6.7 Weiteres zur Ein-/Ausgabe 40

7 Anwendung: Parsen mit Parsec **42**

8 Anwendungen: Bäume **45**
8.1 Einfache Bäume zum Speichern von Daten 45
8.2 AVL-Bäume 45
8.3 Trie-Strukturen 46
8.3.1 Suchen und Einfügen von Wörtern 47
8.3.2 Löschen von Einträgen 47

Literatur **48**

1 Zur Geschichte

Dieser Abschnitt gibt einen kleinen Überblick über Entwicklungen im Bereich der funktionalen Programmiersprachen. Es wird dabei hauptsächlich auf Ereignisse und Ideen aus dem Umfeld von Haskell eingegangen, welche Beiträge zur Entwicklung der Sprache geliefert haben. Eine umfassende Dokumentation der Geschichte von Haskell findet sich in [4]. Eine Biographie des Mathematikers Haskell Brooks Curry findet sich in [**?**].

1.1 Haskell Brooks Curry

Haskell Brooks Curry (12.09.1900, Millis, Massachusetts - 01.09.1982, State College, Pennsylvania) war ein amerikanischer Mathematiker und Logiker. Er ist bekannt als der Mitbegründer der kombinatorischen Logik, welche eine Grundlage für funktionale Programmiersprachen (ähnlich dem Lambda Kalkül) bildet. Eigentlich wollte Haskell Curry eine Dissertation zum Thema Differentialgleichungen in Harvard anfertigen. Er las jedoch unter anderem einen Artikel von Moses Schönfinkel dem Erfinder der kombinatorischen Logik, welcher ihn sehr interessierte und dazu veranlasste, diese Theorie weiter zu entwickeln. Haskell Curry promovierte innerhalb von einem Jahr in Göttingen und kehrte danach wieder in die USA zurück.

Der Begriff Currying, für den Haskell Curry bekannt ist, wurde erst 1967 eingeführt und war/ist auch unter dem Begriff Schönfinkeln nach seinem eigentlichen Entwickler aus dem Jahr 1920 bekannt. Das Currying ist eine Methode, um aus einer Funktion des Typs $f\colon (X \times Y) \to Z$ eine Funktion des Typs $\mathrm{curry}(f)\colon X \to (Y \to Z)$ zu erzeugen. Dadurch können Funktionen mit einfacheren und gut entwickelten Theorien (wie dem Lambda-Kalkül) untersucht werden.

1.2 Entwicklung funktionaler Programmiersprachen

Die erste heute als funktionale Programmiersprache angesehene Sprache ist LISP, welche Ende der fünfziger Jahre entwickelt wurde. Seit dem hat sich in der Theorie der Informatik und der Programmierung einiges getan und es wurden weitere Sprachen mit neuen theoretischen Hintergründen entwickelt. In diesem Abschnitt werden einige dieser Sprachen und ihre Besonderheit aufgelistet.

1.2.1 LISP

LISP (List Processing Language) ist eine der bekanntesten Programmiersprachen und wurde seit 1956 von John McCarthy am MIT entwickelt. Eine erste Implementierung begann im Herbst 1958. Ungefähr zwanzig Jahre später schrieb McCarthy die umfassende Entstehungsgeschichte der Sprache in [8] aus seinem Gedächtnis nieder. LISP wurde entwickelt, um mathematische Notationen für die Programmierung nutzen zu können. Es existiert eine starke Anlehnung an das Lambda-Kalkül von Alonzo Church.

Mittlerweile gilt LISP nicht mehr als Sprache, sondern ist als Familie von Sprachen bekannt. Es existieren sehr viele Dialekte und Weiterentwicklungen der Sprache, von denen zum Teil wiederum unterschiedliche Implementierungen existieren. Die beiden bekanntesten Weiterentwicklungen von LISP sind:

- **Scheme:** eleganter kleiner Sprachumfang (wenige Konstrukte) hauptsächlich in der Lehre verwendet
- **Common Lisp:** sehr umfangreiche Sprache, welche internationaler Industriestandard ist und Unterstützung prozeduraler Makros bietet

1.2.2 ML

Die funktionale Programmiersprache ML wurde 1973 an der University of Edinburgh von Robin Milner entwickelt. Es handelt sich jedoch nicht um eine reine funktionale Sprache, da auch imperative Bestandteile vorhanden sind und so eine imperative Progammierung möglich ist. Das Typsystem von ML ist jedoch statisch (Typprüfungen zur Kompilierzeit) und stark (keine Zuweisungen unterschiedlicher Typen zu einer Variablen), wie es auch in Haskell der Fall ist.

Wie bei den meisten Sprachen zu dieser Zeit, war die Auswertung von Ausdrücken in ML strikt. Das bedeutet, sie werden ausgewertet bevor sie als Parameter übergeben werden.

1.2.3 Lazy Evaluation

Die Lazy Evaluation oder auch Bedarfsauswertung wurde 1976 gleich dreifach an unabhängigen Stellen erfunden. Ein Vorteil gegenüber der strikten Auswertung war, dass nun z. B. unendliche Datenstrukturen verwendet werden konnten. Weiterhin wurde durch Beschränken der Auswertung von Ausdrücken ein Laufzeitgewinn verzeichnet. Einer der Erfinder der Lazy Evaluation ist David Turner, der dieses Feature in seiner funktionalen Sprache SASL (St. Andrews Standard Language) implementierte.

Einige Jahre später war der Hype um die Bedarfsauswertung so weit vorgedrungen, dass sogar spezielle Hardware entwickelt wurde. In den Jahren 1977 bis 1987 wurden

unterschiedliche große Projekte in Cambridge, am MIT und vielen anderen Stellen ins Leben gerufen. Die Entwicklungen stellten sich jedoch später als nicht sehr erfolgreich heraus, da die selben Aufgaben auch mit guten Compilern und Standard-Hardware gelöst werden konnten.

1.2.4 Funktionale Programmiersprachen geraten in Mode

John Backus erhielt 1977 den ACM Turing Award und stellte in einer Vorlesung anlässlich der Verleihung seine neusten Arbeiten zur funktionalen Programmierung vor. Seine Rede trug den Titel „Can Programming be liberated from the von Neumann Stype?“. Durch diesen Vortrag wurde die funktionale Programmierung weiter in das Interesse der Öffentlichkeit getragen und Ende der 70er sowie Anfang der 80er wurden viele verschiedene funktionale Sprachen entwickelt.

Zu Beginn der 80er Jahre fanden viele Konferenzen zu dem Thema statt und langsam wurde der Markt der funktionalen Programmiersprachen unübersichtlich. Viele Forschungsgruppen arbeiteten an eigenen Ideen und Implementierungen von reinen funktionalen Programmiersprachen mit Bedarfsauswertung. Bis auf eine Ausnahme wurden alle diese Sprachen jedoch nur an wenigen Standorten eingesetzt. Diese Ausnahme bildete die von David Turner kommerziell entwickelt und verbreitete Sprache Miranda, welche auf SASL und Ideen aus ML basierte.

1.3 Die Geburt von Haskell

Die Existenz der vielen unterschiedlichen funktionalen Programmiersprachen warf einige Probleme auf. Man hatte den Wunsch nach einer Sprache mit allen neuen eleganten Features und einem Standard, dem sich alle anschließen würden. Es etablierten sich kaum Anwendungen, da der Markt der Sprachen noch viel zu sehr in Bewegung war. Peyton Jones und Paul Hudak beschlossen 1987 auf der Konferenz „Functional Programming and Computer Architecture“ ein Treffen zur Diskussion der Entwicklung eines neuen Standards abzuhalten.

1.3.1 Das Komitee

Es wurde ein großes offenes Komitee von Wissenschaftlern gebildet, die den Standard für eine neue funktional Programmiersprache erarbeiten wollten. Die erste Idee war es, auf der existierenden Sprache Miranda aufzubauen. David Turner befürchtete jedoch, dass dadurch unterschiedliche Varianten seiner kommerziell vertriebenen Sprache entstehen würden und somit der „bisherige Standard“ gefährdet sei. Man entschied sich also nach seiner Absage, eine komplett neue Sprache zu definieren. Die Vorteile dieses Vorgehens lagen darin, die Möglichkeit zu haben eine Reihe von Zielen zu verwirklichen, wie z. B. eine vollständige Definition von Syntax und Semantik.

Leider wurde dieses Ziel jedoch nicht erreicht und eine vollständige Definition der Semantik ist ausgeblieben.

Im Januar 1988 traf sich das „FPLang Komitee“ zum ersten Mal in Yale, um die Grundlagen für die Sprache sowie das weitere Vorgehen der Entwicklung festzulegen. Es bestanden eine Menge Ziele, die in [4] zu finden sind und teilweise umgesetzt wurden. Ein Vorteil des Komitees und der verwendeten Mailinglisten war die schnelle Kommunikation neuer Ideen und eine Abstimmung mit vielen Wissenschaftlern von unterschiedlichen Kontinenten.

1.3.2 Der Name

Zur Findung eines Namens für die neue Sprache wurden viele Vorschläge auf einer Tafel gesammelt, von denen nach und nach einige wieder gestrichen wurden. Man einigte sich schließlich darauf, dass der Name der Sprache „Curry“ sein sollte. Damit sollte sie nach dem mittlerweile verstorbenen Mathematiker Haskell Curry benannt werden. Der Überlieferung in [4] zu folge, befürchtete man doch jedoch nach einer Nacht Reflektion über den Namen, eine Verwechslung mit dem Gewürz Curry oder dem Schauspieler Tim Curry. Dieser hatte gerade einen großen Auftritt in der „Rocky Horror Picture Show“ hinter sich. Später spielte er auch in „Scary Movie 2“, „3 Engel für Charlie“, „Loaded Weapon“ und „Kevin allein in New York“ mit. Man entschied sich also für ein besseres Image der Sprache, diese nach dem Vornamen von Curry zu benennen.

Die Witwe von Haskell Curry sollte noch um ihr Einverständnis gefragt werden und wurde sogar zu einem Vortrag über die neue Sprache eingeladen, von dem sie angeblich nicht viel verstand. Sie stimmte jedoch zu, dass der Name Haskell verwendet werden könne und meinte noch, ihr Mann hätte den Namen gar nicht so gut leiden können.

1.3.3 Die Entwicklung

Hudak und Wadler übernahmen die Aufgabe der Chefredakteure für den ersten Haskell Report, der angefertigt werden sollte. Viele Ideen wurden ausschließlich über eine Mailingliste diskutiert. Am ersten April 1990 war es so weit und der Haskell Report 1.0 wurde veröffentlicht. Weitere 16 Monate später veröffentlichte man die Version 1.1 und im März 1992 schon die Version 1.2 des Haskell Reports.

Als Paul Hudak 1994 die Domain „haskell.org“ registrierte, begann Haskell weiter ins Licht der breiten Öffentlichkeit zu gelangen. Mit der letzten großen Änderung wurde im Februar 1999 der Haskell 98 Report veröffentlicht. Das formale Komitee hörte auf zu existieren. Haskell erfuhr dank des neuen Standards eine große Akzeptanz in der Lehre und im professionellen Einsatz. Die letzte Veröffentlichung eines Haskell Reports fand im Januar 2003 statt. Der „Haskell 98 Report (Revised)“ (siehe [5]) behob hauptsächlich kleinere Fehler und Unzulänglichkeiten.

An einer neuen Version der Sprache Haskell wird weiterhin öffentlich gearbeitet. Man kann sich online in einem Wiki auf der Seite von Haskell `http://hackage.haskell.org/trac/haskell-prime` an dem Design der Sprachversion „Haskell Prime“ beteiligen.

2 Einführung und Grundlagen

2.1 Die Arbeitsumgebung

In den Übungen wird der Glasgow Haskell Compiler (bzw. die interaktive Shell GHCi) verwendet. Das Paket kann unter `http://haskell.org/ghc/` bezogen werden. Eine gute Alternative für Windows ist der Interpreter Hugs98, der Haskell98 fast vollständig implementiert. Eine Beschreibung der Einschränkungen findet sich in [1] in Kapitel 5. Hugs kann unter `http://haskell.org/hugs/` bezogen werden.

Im Unterschied zu einem Compiler erzeugt ein Interpreter keine eigenständig ausführbaren Dateien, sondern interpretiert den Quellcode zur Laufzeit des Systems. Ein Compiler erzeugt beim Übersetzen sog. „Objectcode“, der optimiert werden kann und so die Ausführung des Programms beschleunigt.

Nach der Installation bieten Hugs und der GHC eine interaktive Haskell-Shell an (GHCi bzw. WinHugs). Der Benutzer erhält eine Eingabeaufforderung der Form `Prelude>_`. Hier ist `Prelude` ein Standardmodul, das Funktionen und Typen beinhaltet, die z. B. zur Ein-/Ausgabe und zur Behandlung von Listen dienen. Außerdem sind arithmetische Operatoren enthalten.

Um eigene Funktionsdefinitionen komfortabel nutzen zu können, sollten diese in einer separaten Datei gespeichert werden (meistens mit Endung „.hs“). Es gibt auch die Möglichkeit, einfache Funktionsdefinitionen über ein `let`-Konstrukt direkt in der interaktiven Shell einzugeben (siehe Abschnitt 3.5).

Dateien können mit dem Befehl `:load` *`Dateiname.hs`* (kurz `:l` *`Dateiname.hs`*) geladen werden. Durch den Befehl `:cd` *`Verzeichnis`* kann in ein Verzeichnis gewechselt werden. Mit dem Befehl `:edit` *`Dateiname.hs`* (kurz `:e` *`Dateiname.hs`*) können Dateien editiert werden. Die Angabe des Dateinamens ist optional. Wenn er weggelassen wird, öffnet sich der Editor mit der aktuell geladenen Datei. Über den Befehl `:reload` (kurz `:r`) kann eine modifizierte Datei erneut geladen werden.

Beispiel 2.1.

```
1    ___         ___ _
2   / _ \ /\  /\/ __(_)
3  / /_\// /_/ / /  | |      GHC Interactive, version 6.6.1
4 / /_\\/ __  / /___| |      http://www.haskell.org/ghc/
5 \____/\/ /_/\____/|_|      Type :? for help.
6
7 Loading package base ... linking ... done.
```

```
 8  Prelude> 1+2
 9  3
10  Prelude> (23-22)/15
11  6.666666666666667e-2
12  Prelude> mod (2^31-1) (2^9-2)
13  127
```

Listing 2.1: Einige einfache Berechnungen in Haskell

Informationen zu Kommandos in der interaktiven Shell erhalten Sie über den Befehl `:help` (kurz `:?`). Sie verlassen das Programm mit dem Befehl `:quit` (kurz `:q`).

2.2 Typen

Jeder gültige Ausdruck in Haskell muss einen eindeutigen Typen besitzen. In Haskell gibt es einen Typchecker, der eine *starke statische Typisierung* durchsetzt. Nach diesem Konzept sollten keine Typfehler mehr zur Laufzeit auftreten.

Einige Typen, die dem Benutzer zur Verfügung stehen sind `Int, Float, Char, String` und `Bool`. Zu einem Wert kann in Haskell „per Hand“ der Typ angegeben werden. Man schreibt dazu z. B. `23::Int` oder `„Hello World!“::String`. Normalerweise versucht Haskell selbst einen passenden Typen zu bestimmen. Interessant wird diese Notation bei Funktionen, die bestimmte Typen benötigen. Hier kann es manchmal zu „falsch“ bestimmten Typen kommen (falsch im Sinn der Intention des Programmierers). Es bietet sich an, jedes Mal den Typen explizit anzugeben, da sich so die Lesbarkeit des Codes erhöht und Fehler in der Verwendung von Funktionen schneller erkannt werden.

```
 1  module Simple where
 2
 3  -- Erhöhen eines Integers um 1
 4  inc :: Int -> Int   -- Signatur
 5  inc x = x + 1       -- Implementierung/Berechnungsvorschrift
 6
 7  -- Addieren von zwei Integern
 8  sum' :: Int -> Int -> Int
 9  sum' x y = x + y
10
11  -- Berechnen des Durchschnitts von zwei Werten
12  averageOf2 :: (Fractional a) => a -> a -> a
13  averageOf2 a b = (a + b)/2
14
15  -- Bemerkung:
```

```
16 -- Mit "Fractional a" wird die s. g. Typklasse von a festgelegt, d.h.
       welche
17 -- Funktionen für a nötig sind. Hier ist das der "/"-Operator.
18 -- Dieses Konzept wird in Kapitel 3 näher erläutert.
19
20 -- Einer Zeichenkette "!" anfügen
21 exclaim :: String -> String
22 exclaim a = a ++ "!"
```

Listing 2.2: Datei Simple.hs

Die Signatur einer Funktion oder der Typ eines Wertes kann durch den Befehl `:type <Ausdruck>` (kurz `:t`) ausgegeben werden.

2.3 Funktionen

Funktionen bestehen in Haskell aus zwei Teilen: der Signatur (der Typ) und der Definition. Die Signatur hat dabei die Form: f :: A -> B. Dies entspricht bis auf einen zusätzlichen Doppelpunkt der mathematischen Notation für eine Abbildung f von der Menge A in die Menge B. In Haskell sind diese Mengen Typen.

Mehrstellige Funktionen besitzen folgenden Typ: f :: A_1 -> A_2 -> ... -> A_n -> B. Dabei sind die Mengen $A_1 ... A_n$ die Parametertypen und B der Rückgabetyp. Der Grund für diese etwas ungewöhnliche Notation ist, dass in Haskell jede Belegung mit Parametern eine neue Funktion mit entsprechend weniger Parametern erzeugt. Dies wird als Currying bezeichnet.

Die Definition einer Funktion hat folgenden Aufbau: f a_1 a_2 ... a_n = <Ausdruck>.

Beispiel: Die Funktion `inc` in Listing 2.2 besitzt einen Parameter vom Typ `Int` und bildet auf einen Wert vom Typ `Int` ab. Für die Funktion `sum'` müssen zwei Zahlen angegeben werden. Es ist jedoch möglich, auch eine Zahl anzugeben. In diesem Fall ergibt sich kein Wert, sondern eine neue Funktion! Es kann also durch die Funktion `sum'` die Funktion `inc` realisiert werden:

```
1 -- Zweite Möglichkeit inc zu definieren
2 inc' = sum' 1
```

Listing 2.3: Alternative zur Definition von `inc`

2.4 Verzweigungen

Durch ein `if-then-else`-Konstrukt kann abhängig von einer Bedingung mit Rückgabewert `Bool` das Ergebnis bestimmt werden. Das Besondere ist hier (im Vergleich

zu anderen Programmiersprachen wie z. B. Java oder C++), dass beide Zweige einen gültigen Wert liefern müssen.

Fallunterscheidungen oder Funktionsdefinitionen mit Nebenbedingungen können auch wie im zweiten Beispiel in Listing 2.4 realisiert werden. Zu beachten ist, dass die Einrückung vor dem | zwingend erforderlich ist.

```
1  -- Maximum von zwei Zahlen berechnen
2  max :: Int -> Int -> Int
3  max a b = if (a > b) then a else b
4
5  -- Realisierung der signum Funktion
6  signum x | x < 0    = -1
7           | x == 0   = 0
8           | x > 0    = 1
```

Listing 2.4: Möglichkeiten zur Auswahl von Werten

2.5 Variablen

In Haskell werden Variablen wie Funktionen ohne Eingabeparameter definiert. Sie können einen Typen besitzen und einen Wert annehmen. Es gibt neben den global definierten Variablen lokale Variablen, die innerhalb einer Funktion verwendet werden können.

```
1  -- Globale Variable (Symbol für einen Wert)
2  pi :: Float
3  pi = 3.141592
4
5  -- radiusQuadrat ist eine lokale Variable
6  kreisFlaeche radius = pi * radiusQuadrat
7          where radiusQuadrat = radius * radius
```

Listing 2.5: Globale und lokale Variablen

2.6 Tupel und Listen

In Haskell gibt es zwei typische Basisdatenstrukturen: Tupel und Listen. Der Unterschied zwischen diesen beiden ist, dass Tupel verschiedene Typen zu einer Struktur fester Größe (zur Laufzeit) zusammenfassen. Listen dagegen können beliebig groß werden, aber dürfen auch nur Werte eines Typs enthalten.

2.6.1 Tupel

Für den Typen eines Tupels kann mit Hilfe von `type` ein Name vergeben werden (s. Listing 2.6). Dieser kann dann einfach wiederverwendet werden.

Jede Größe von Tupeln ist erlaubt, bis auf Tupel der Größe eins ((a)). Die Schreibweise würde sich mit der Klammerung von Ausdrücken überschneiden.

```
 1  -- Rückgabe eines Tupels aus Kreisfläche und Umfang
 2  kreisFlUm radius = (flaeche, umfang)
 3          where   flaeche = kreisFlaeche radius
 4                  umfang  = 2 * radius * pi
 5
 6
 7  -- Darstellung eines Punktes
 8  type Punkt = (Float, Float)
 9
10  -- Verwendung des neuen Typs
11  norm :: Punkt -> Float
12  norm (x,y)= sqrt (x*x + y*y)
```

Listing 2.6: Tupel und Typen

Die Notation norm (x,y) wird als Mustervergleich bezeichnet. Dadurch kann auf die einzelnen Elemente des übergebenen Tupels zugegriffen werden. Dies wird näher in Kapitel 3 erläutert.

2.6.2 Listen

Durch das Modul `Prelude` wird auch eine Unterstützung von Listen bereitgestellt. So ist z. B. der Typ `String` eine Liste mit Elementen des Typen `Char`.

Listen können von jedem Typen angelegt werden (somit auch von Tupeln). Jedes Element muss jedoch dem gleichen Typen entsprechen. Listen können potenziell unendlich lang werden, was wegen Lazy-Evaluation (Auswertung nur wenn nötig) kein Problem darstellt, solange man nicht alle Elemente auswertet.

Hilfreiche Funktionen zum Erstellen von Listen sind der Konstruktor `[]` (genannt `nil`, erzeugt eine leere Liste) und die Funktion `(:) :: a -> [a] -> [a]` zum Voranstellen eines Elements vor eine Liste (genannt `cons`) sowie die Funktion `(++) :: [a] -> [a] -> [a]` zum Hintereinanderhängen von zwei Listen. Eine Liste kann durch `l = [„Hallo“]` (ein Element) oder auch `l = [1, 2 .. 99]` (99 Elemente) erzeugt werden.

```
 1  -- Ein einfaches Tupel für ein Bankkonto
 2  -- Name des Kunden in der ersten Komponente
 3  -- Saldo in der zweiten
```

```
 4  type Konto = (String, Int)
 5
 6  saldo :: Konto -> Int
 7  saldo (n, s) = s
 8
 9  name :: Konto -> String
10  name (n, s) = n
11
12  -- Eine Liste von Konten
13  type Bank = [Konto]
14
15  -- Berechnen des Geldbestandes auf allen Konten
16  gesamtBestand :: Bank -> Int
17  gesamtBestand [] = 0
18  gesamtBestand (k:ks) = (saldo k) + (gesamtBestand ks)
```

Listing 2.7: Einfaches Modell einer Bank

3 Funktionen

3.1 Rekursion

Auf dem letzten Blatt ist bereits eine rekursiv definierte Funktion aufgetreten. Die Funktion `gesamtBestand` taucht auf der rechten Seite ihrer eigenen Definition auf (siehe Listing 2.7). Einzelheiten zum Thema „rekursive Spezifikation“ werden in einer eigenen Übung behandelt.

3.1.1 Summe der natürlichen Zahlen

Ein typisches Beispiel für die Rekursion ist das Aufsummieren der natürlichen Zahlen bis zu einer gegebenen Zahl n (als richtige Programmierer lassen wir den Rechner alles durchrechnen und benutzen keine Tricks wie der kleine Gauß).

Die gesuchte Funktion trägt den Namen `natSum` und ist intuitiv definiert als

```
natSum n = 1 + 2 + 3 + ... + n.
```

Wenn man auf der rechten Seite der Gleichung die Definition von `natSum (n-1)` einsetzt, erhält man die Rekursion

```
natSum n = natSum (n-1)+ n.
```

3.1.2 Rekursionsschritt und Basis

Somit ist der *Rekursionsschritt* gefunden, da er in dem Aufruf der Funktion mit dem um eins verminderten Argument besteht. Die Funktion wird so jedoch potentiell unendlich lange laufen, da `n` immer wieder einen Vorgänger besitzt. Es muss also eine *Abbruchbedingung (Basis)* angegeben werden, die immer erreicht wird. Für positive `n` wird die Folge schließlich `0`. Die Abbruchbedingung lautet also `natSum 0 = 0`.

Jede rekursive Funktion benötigt mindestens eine Basis und mindestens einen Rekursionsschritt. Die jeweilige Anzahl ist jedoch nach oben offen.

```
1 natSum :: Int -> Int
2 natSum 0 = 0                 -- Basis
3 natSum n = natSum (n-1) + n -- Rekursionsschritt
```

Listing 3.1: Einfache rekursive Funktion zur Berechnung der Summe 1 + ... + n

```
1 solange b != 0
2     wenn a > b
3         dann a := a - b
4         sonst b := b - a
5 return a
```

Listing 3.2: Iterative Variante des euklidischen Algorithmus

3.2 Mustervergleich (pattern matching)

Für die Definition von Funktionen wird häufig die Technik des „pattern matching“ verwendet. Hierbei nutzt man die zugrunde liegenden Informationen über den Aufbau von Typen, um auf tiefer liegende Werte zugreifen zu können.

Beispiel 3.1. Verschiedene Verwendungen des Mustervergleichs in der Funktionsdefinition:

```
1 -- Deklaration des Datentypen
2 type Punkt = (Int, Int)
3
4 -- Zugriff auf die X-Koordinate eines Punktes
5 xCoord :: Punkt -> Int
6 xCoord (a, b) = a
```

Listing 3.3: Mustervergleich bei Verwendung von Tupeln

```
1 -- Ausgeben des ersten Elements einer Liste
2 head :: [a] -> a
3 head [] = error "head:␣Liste␣ist␣leer!"
4 head (x:xs) = x
```

Listing 3.4: Mustervergleich unter Verwendung von Konstruktoren

3.3 Polymorphie

Polymorphie bedeutet „Vielgestaltigkeit“, da dieses Konzept z. B. die „Vielgestaltigkeit“ eines Arguments erlaubt. Es wird zwischen der parametrischen und der Ad-Hoc-Polymorphie unterschieden.

3.3.1 Parametrische Polymorphie in Haskell

Die verwendeten Funktionen **head, tail, fst, ...** sind nicht auf einen Typen festgelegt, sondern enthalten in der Signatur Typvariablen (beginnend mit Kleinbuchstaben). Die konkrete Instanz (der Typ) wird erst bei der Anwendung der Funktion festgelegt (Instantiierung der Typvariablen).

In den Funktionen, die bisher definiert wurden, war in der Signatur der Typ festgelegt, mit dem sie gearbeitet haben. Wenn auf diese Art z.B. die Funktion **head** definiert werden soll, ist dies recht mühsehlig:

```
1 intHead :: [Int] -> Int
2 intHead (x:_) = x
3
4 floatHead :: [Float] -> Float
5 floatHead (x:_) = x
6
7 ...
```

So müsste man für jeden neuen Typ **head** neu definieren. Wenn man sich die beiden Definitionen ansieht, erkennt man, dass sie zeichenweise identisch sind und sich nur die Typen unterscheiden.

Um dies zu vermeiden, existiert in Haskell die so genannte (parametrische) Polymorpie. Das bedeutet, man paramtrisiert den Typ in der Signatur einer Funktion. Der konkrete Typ wird dabei durch eine Typvariable ersetzt. Konkrete Typen und Typvariablen werden in Haskell dadurch unterschieden, dass konkrete Typen mit einem Großbuchstaben und Variablen mit einem Kleinbuchstaben beginnen.

Damit kann **head** allgemein definiert werden:

```
1 head :: [a] -> a
2 head (x:_) = x
```

Beispiel 3.2.

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

Listing 3.5: Infix-Funktion `(++)` zum Konkatenieren von Listen beliebigen Typs

Es ist möglich, mehrere Typvariablen zu verwenden wie z. B. in Tupeln `f :: (a, b) -> c` oder Mischungen zu verwenden wie bei `length :: [a] -> Int`. Jede Typvariable wird bei der Instantiierung eindeutig festgelegt (z. B. alle `a`'s werden zu **Int** und alle `b`'s zu **String**).

3.3.2 Ad-Hoc-Polymorphie

Diese Art von Polymorphie ist in OO-Sprachen als „Überladen von Funktionen“ bekannt. Für eine Menge von Typen werden jeweils einzelne Implementierungen mit unterschiedlichen Parametern angelegt. In Haskell wird diese Art der Polymorphie durch Typklassen ermöglicht. Dies wird im nächsten Abschnitt erläutert.

3.4 Typklassen

Typklassen unterteilen in Haskell Typen in Klassen, die jeweils eine bestimmte Menge von Operationen zur Verfügung stellen.

Zum Beispiel implementieren alle Typen der Typklasse **Eq** (Equality) einen Vergleich ihrer Elemente. Der Compiler bzw. der Interpreter sucht sich bei Anwendung der Funktion `(==)` die zum Typ passende Implementierung aus und wendet sie an.

Die Verwendung von Typklassen für eigene Typen (um z.B den Operator `(==)` zu implementieren) wird in Kapitel 4 erläutert.

Weitere wichtige Typklassen sind **Ord, Num, Eq, Integral** und **Fractional**. Informationen über die Klassen können in der interaktiven Haskell Shell durch den Befehl `:info` *Typklasse* (kurz `:i` *Typklasse*) abgerufen werden:

Beispiel 3.3.

```
1  Prelude> :i Num
2  class (Eq a, Show a) => Num a where
3    (+) :: a -> a -> a
4    (*) :: a -> a -> a
5    (-) :: a -> a -> a
6    negate :: a -> a
7    abs :: a -> a
8    signum :: a -> a
9    fromInteger :: Integer -> a
10         -- Defined in GHC.Num
11 instance Num Double -- Defined in GHC.Float
12 instance Num Float -- Defined in GHC.Float
13 instance Num Int -- Defined in GHC.Num
14 instance Num Integer -- Defined in GHC.Num
```

Listing 3.6: Informationen über Typklassen in der interaktiven Shell

In Zeile 2 – 10 steht die Definition der Typklasse. In Zeile 2 beginnt die Definition der Typklasse **Num**, eingeleitet durch **class**. Außerdem wird in der Zeile festgelegt, dass alle Instanzen der Klasse **Num** auch vergleichbar (**Eq**) und ausgebbar (**Show**) sein sollen.

Hier sieht man auch die Syntax bei der Verwendung von Typklassen: vor einem Doppelpfeil (`=>`) stehen die Typklassen, in denen der für `a` eingesetzte Typ sein muss. Hinter dem Pfeil kann `a` verwendet werden.

In Zeile 3 – 10 werden die Funktionen festgelegt, die für `a` definiert sind, wenn der Typ der Klasse `Num` angehört.

Verwendung mit Funktionen

Die Verwendung in der Signatur einer Funktion ist analog zu der Definition von Typklassen:

Beispiel 3.4.

```
1 -- Funktion stellt fest, ob ein Element in einer Liste vorhanden ist
2 contains :: Eq a => a -> [a] -> Bool
3 contains y [] = False
4 contains y (x:xs) = if (x == y)
5                     then True
6                     else contains y xs
```

Listing 3.7: Verwendung der Typklasse `Eq`

In Listing 3.7 müssen alle in den Argumenten zugelassenen Typen Typen der Typklasse `Eq` sein und damit die beiden Funktionen `(==):: a -> a -> Bool` und `(/=):: a -> a -> Bool` implementieren, mit denen Elemente auf Gleichheit bzw. Ungleichheit geprüft werden können.

3.5 Lokale Variablen

Es existiert in Haskell, neben dem `where`-Schlüsselwort in Funktionen, noch eine zweite Möglichkeit, lokale Variablen zu definieren. Hierzu gibt es den `let ... in`-Ausdruck, der einen Wert direkt aus einem Ausdruck liefert. Er wird wie in Listing 3.8 verwendet.

```
1 variable =
2     let
3         -- Deklaration von Variablen
4         -- Zuweisungen
5     in  -- Ausdruck
```

Listing 3.8: Struktur des `let ... in`-Ausdrucks

Alle lokalen Variablen müssen mindestens einmal innerhalb der Zuweisungen oder dem Ausdruck gelesen werden!

Beispiel 3.5. In der Funktion `average` (Listing 3.9), die den Durchschnitt der Werte in einer Liste berechnet, wird das `let-in`-Konstrukt eingesetzt.

```
1  average :: (Fractional b, Integral a) => [a] -> b
2  average xs =
3      let s = (sum xs)
4          w = (length xs)
5      in (fromIntegral s)/(fromIntegral w)
```

Listing 3.9: Berechnung des arithmetischen Mittels mit `let ... in`

Neben der Verwendung in Definitionen von Funktionen kann der `let-in`-Ausdruck auch in der interaktiven Shell des GHC verwendet werden. In diesem Fall muss jedoch alles in einer Zeile stehen. Die Zeilen, welche dem `let` folgen werden durch `;` beendet. Zur besseren Übersicht können die Zeilen durch `{..}` eingefasst werden.

```
1  Prelude> let {xs = [1..100]; s = (sum xs); w = (length xs); } in (
       fromIntegral s)/(fromIntegral w)
2  50.5
```

Listing 3.10: Verwendung des `let ... in`-Ausdrucks in der Shell

Wenn nur der `let`-Teil verwendet wird, können in der Shell Funktionen und Variablen definiert werden. Diese interaktiven Definitionen sind in der Shell des Interpreters Hugs 98 leider nicht möglich (siehe [7]).

Beispiel 3.6.

```
1  -- Definition und Ausgabe einer Variablen
2  Prelude> let pi = 3.14159
3  Prelude> pi
4  3.14159
5
6  -- Definition von Funktionen in der interaktiven Shell
7  Prelude> let areaRectangle l w = l * w
8  Prelude> let areaSquare s = areaRectangle s s
9  Prelude> areaSquare 5
10 25
```

Listing 3.11: Interaktive Definition von Variablen und Funktionen

3.6 Praxisbeispiel: Listen im und um den Supermarkt

Es wird ein simpler Supermarkt modelliert, in dem es Warenlisten mit den dazugehörigen Preisen gibt:

```
1  module Supermarkt where
2
3  type Cent = Int
4  type Produkt = String
5
6  -- Liste mit Einträgen der Form (Produkt, Preis)
7  type PreisListe = [(Produkt, Cent)]
8
9  -- Liste mit Einträgen der Form (Produkt, Menge)
10 type EinkaufsListe = [(Produkt, Int)]
```

Listing 3.12: Datenstrukturen im Supermarkt-Beispiel

Listen sortieren

Eine einfache Liste vom Typ `[Int]` soll sortiert werden. Der vorgeschlagene Sortieralgorithmus ist „Insertion Sort". Dieser basiert auf der Funktion `insertSorted`, die ein Element an der richtigen Stelle in einer sortierten Liste einfügt. Die verwendeten Typen müssen Instanzen der Klasse `Ord` sein, da die Infix-Relation `(<)` benötigt wird.

Die Schritte des Algorithmus sind wie folgt:

- Wenn die Liste leer ist: Gib eine leere Liste aus.
- Sonst: Füge den `head` der Liste in den sortierten `tail` ein.

3.7 Funktionen höherer Ordnung

Haskell realisiert wie alle funktionalen Sprachen das Konzept „Funktionen höherer Ordnung" (s. die theoretische Übung zum Thema λ-Kalkül). Die bisher behandelten Funktionen waren erster Ordnung, da ihre Parameter einfache Typen waren und sie einfache Werte lieferten. Es gibt jedoch auch die Möglichkeit, Funktionen selbst als Parameter zu verwenden. Dies ist z. B. dann sinnvoll, wenn alle Elemente einer Liste nach der selben Vorschrift modifiziert werden sollen. Man kann in diesem Fall eine Funktion erster Ordnung definieren, die die Veränderung an einem Element durchführt:

```
1  double :: Num a => a -> a
```

```
2  double x = 2*x
3
4  inc :: Num a => a -> a
5  inc x = x + 1
```

Listing 3.13: Funktionen erster Ordnung zur Modifikation von Listenelementen

Anstelle zweimal eine ähnliche Definition für eine Funktion anzugeben, die `inc` bzw. `double` elementweise auf die ganze Liste anwendet, kann die Funktion im Modul **`Prelude`** definierte Funktion **`map`** wie in Listing 3.15 verwendet werden.

```
1  map :: (a -> b) -> [a] -> [b]
2  map f []     = []
3  map f (x:xs) = f x : map f xs
```

Listing 3.14: Definition der Funktion **`map`**

Beispiel 3.7.

```
1  *Main> let l = [1..10] in map inc l
2  [2,3,4,5,6,7,8,9,10,11]
3  *Main> let l = [1..10] in map double l
4  [2,4,6,8,10,12,14,16,18,20]
```

Listing 3.15: Verwendung der Funktion **`map`**

3.8 Anonyme Funktionen – Lambda Abstraktion

Es ist möglich in Haskell mit Funktionen zu arbeiten, die keine normale Definition mit einem Funktionsnamen besitzen. Diese Funktionen nennt man anonym und sie sind stark an das Lambda-Kalkül angelehnt. Der Syntax dieser Funktionsdefinition ist \\p_1 p_2 ... p_n `->` exp. Dabei stehen die p_i für Muster wie z. B. einfache Variablen oder auch aufwändigere Muster wie `(x:xs)`, also eine Liste mit dem ersten Element `x`. Wie bei der normalen Funktionsdefinition durch Mustervergleich, darf keine Variable auf der linken Seite mehrfach vorkommen.

Nach dem Pfeil (->) folgt die eigentliche Definition der Berechnung also das Ergebnis der Funktion. Eine partielle Anwendung der Funktion auf eine Anzahl i der n Parameter ist auch hier möglich. Die Funktion verhält sich so wie alle anderen Funktionen und liefert eine Funktion, die die restlichen $(n - i)$ Parameter als Definitionsbereich besitzt.

Beispiel 3.8.

- Die Funktion `\x -> x * 2` verdoppelt jeden übergebenen Wert. Eine Anwendung wäre z. B. `(\x -> x * 2)21`, welche den Wert `42` liefert.
- Die Funktion `\x y (z:zs)-> max x (max y z)` findet das Maximum von zwei Zahlen und dem ersten Element einer Liste. Eine Anwendung wäre `(\x y (z: zs)-> max x (max y z))123 456 [300, 34, 3243, 0]` und liefert den Wert `456`.

3.9 Praxisbeispiel: Numerisches Differenzieren

Theorie

Es wird nun ein etwas größeres Beispiel betrachtet, um die bisherigen Konzepte in der Praxis zu sehen.

Aus der Einführung von Ableitungen ist der sog. *symmetrische Differenzenquotient* bekannt. Dieser hat die Form

$$q(x,h) = \frac{f(x+h) - f(x-h)}{2h}$$

und damit

$$f'(x) := \lim_{h \to 0} q(x,h)$$

Diesen Umstand kann man sich für das numerische Ableiten zu Nutze machen: man startet bei einem beliebigen Wert für h und lässt diesen gegen 0 laufen. Eine einfache Variante ist, das Intervall in jedem Schritt zu halbieren: $h, \frac{h}{2}, \frac{h}{4}, \frac{h}{16}, \ldots$ also $h_0 = Startwert$ und $h_n = \frac{h_{n-1}}{2}$.

Implementierung

Dies lässt sich nun fast eins zu eins in Haskell übertragen:

In den ersten beiden Zeilen in Listing 3.16 wird ein ungefährer Vergleich für Fließkommazahlen implementiert. Dieser hat aber nichts mit der eigentlichen Implementierung zu tun. Hier kann man aber sehen, wie ein eigener Infix-Operator implementiert werden kann. Dieser kann in Klammern (`(~=)`) wie eine normale Funktion genutzt werden.

`diffQuote` ist die direkte implementierung von $q(x,h)$. Hier übernimmt `diffQuote` aber die Funktion $f(x)$ als Parameter. Damit ist sie eine Funktion höherer Ordnung.

`diffIterate` berechnet das nächste Intervall und q. Wenn das der Quotient sich nicht mehr vom vorherigen Unterscheidet, wurde die Ableitung an der Stelle x gefunden. Ansonsten wird die nächste Iteration ausgeführt.

`diff` schließlich ist die Schnittstelle für den Benutzer und startet die Iteration lediglich mit geeigneten Startwerten.

```
1  (~=) :: Double -> Double -> Bool
2  x ~= y = (abs (x - y)) < 0.00000001
3
4  diffQuote :: (Double -> Double) -> Double -> Double -> Double
5  diffQuote f x h = ((f (x + h)) - (f (x - h))) / (2.0 * h)
6
7  diffIterate :: (Double -> Double) -> Double -> Double -> Double ->
       Double
8  diffIterate f x hOld dOld =
9      let
10         hNew = hOld / 2.0
11         dNew = diffQuote f x hNew
12     in if (dOld ~= dNew)
13         then dNew
14         else diffIterate f x hNew dNew
15
16 diff :: (Double -> Double) -> Double -> Double
17 diff f x =
18     let
19         h0 = 0.01
20         d0 = diffQuote f x h0
21     in diffIterate f x h0 d0
```

Listing 3.16: Numerisches Differenzieren in Haskell

3.10 Listenkomprehension

Die Listenkomprehension ist eine Methode, um Listen aus anderen Listen zu erzeugen. Die Syntax für die Listenkomprehension ist: `[`*exp* `|` $q_1, ..., q_n$`]` wobei *exp* ein Ausdruck ist und q_i Qualifikatoren sind. Variablen aus dem Ausdruck *exp* können über die Qualifikatoren definiert werden. Es gibt drei verschiedene Arten von Qualifikatoren:

- Generatoren – diese haben die Form `x <- e`, wobei `x` ein Muster des Typen `T` ist und der Ausdruck `e` den Typen `[T]` besitzt.
- Prädikate – Ausdrücke des Typs **`Bool`** über vorher definierten Variablen.
- Lokale Deklarationen – haben die Form **`let`** `decls` ($= q_i$), wobei die Deklarationen in *decls* in dem Ausdruck *exp* und in den q_{i+1} bis q_n sichtbar sind.

Das Ergebnis der Listenkomprehension ist eine neue Liste, deren Elemente aus dem Ausdruck *exp* entstehen, indem die Generatoren „depth-first" abgearbeitet werden. Die Auswertungsreihenfolge ist von links nach rechts. Sollten die Elemente der Liste e bei den Generatoren `x <- e` nicht dem angegebenen Muster in x entsprechen, werden sie einfach übersprungen (also liefert `[x | (3, x)<- [(3,5),(2,5),(3,6)]]` die Liste `[5, 6]`).

Falls eines der Prädikate nicht zutrifft, wird das betroffene Listenelement ebenfalls übersprungen. Demnach ist die Liste `[x | x <- [1..],` **`even`** `x]` eine Liste mit allen geraden Zahlen. Die Notation `[1..]` zählt alle natürlichen Zahlen auf. Die Liste `[x | x <- [1..],` **`even`** `x]` auszugeben ist wegen potentieller Unendlichkeit also keine gute Idee. In solchen Fällen lohnt es sich manchmal zum Verständnis schon, nur die ersten Elemente der neuen Liste zu betrachten. Dies kann mit der Funktion **`take`** `::` **`Int`** `-> [a] -> [a]` getan werden. Wegen der Bedarfsauswertung werden nur die Listenelemente berechnet, welche benötigt werden.

Beispiel 3.9. Listing 3.17 zeigt eine Funktion, die alle geraden Zahlen zwischen `x` und `y` liefert. Der Generator ist dabei `a <- [x..y]`, der `a` nacheinander alle ganzen Zahlen zwischen `x` und `y` zuweist. Nach jeder Zuweisung wird das Prädikat **`even`** `a` ausgewertet. Ein Aufruf könnte so aussehen: `evenBetween (-100)100`.

```
1 evenBetween :: Int -> Int -> [Int]
2 evenBetween x y =
3         let
4                 list = [x..y]
5         in
6                 [ a | a <- list, even a]
```

Listing 3.17: Funktion, die gerade Zahlen zwischen `x` und `y` liefert.

4 Benutzerdefinierte Typen

Die Bibliothek **`Prelude`** versorgt den Programmierer mit einigen grundlegenden Typen und Typklassen. Wenn die Anwendungen komplexer werden, möchte man jedoch durch Abkürzungen oder eigene zusammengesetzte Typen die Komplexität kompensieren. Bisher bekannt ist die abkürzende Schreibweise mit Hilfe des Schlüsselwortes **`type`**, wie dies z. B. in der Zeile `type Punkt = (Float, Float)` der Fall ist. Damit ist die Abkürzung `Punkt` ein einfaches Typsynonym für `(Float, Float)` und kann an seiner Stelle verwendet werden.

4.1 Parametrische Typsynonyme

Zwei Typen, für die die abkürzende Schreibweise verwendet werden soll, können sehr leicht ähnliche Strukturen aufweisen. So sind die beiden folgenden Typen sehr ähnlich.

```
1  -- Liste mit Einträgen der Form (Produkt, Preis)
2  type PreisListe = [(Produkt, Cent)]
3
4  -- Liste mit Einträgen der Form (Produkt, Menge)
5  type EinkaufsListe = [(Produkt, Float)]
```

Listing 4.1: Zwei sehr ähnlich strukturierte Listen

Bei den beiden Listen handelt es sich um so genannte Assoziationslisten. Ein Wert wird dabei einem Schlüssel zugeordnet. Dieser Schlüssel ist der erste Teil der Paares und der Wert befindet sich im zweiten Teil. Die Struktur lässt sich wie bei einem einfachen Typsynonym aufschreiben, enthält jedoch Parameter, die durch Typen ersetzt werden können.

```
1  -- Eine Assoziationsliste mit Schlüssel und Wert
2  type AssocList key value = [(key, value)]
3
4  type PreisListe = AssocList Produkt  Cent
5
6  type EinkaufsListe = AssocList Produkt Float
```

Listing 4.2: Assoziationslisten

In Zeile zwei ist die eigentliche Definition der Struktur einer Assoziationsliste zu sehen. In den darauf folgenden Zeilen wird diese Definition benutzt, um die konkreten Typen der beiden Listen anzugeben. Diese Schreibweise hat den Vorteil, dass der Programmierer auf den bekannten Typen Assoziationsliste zurückgreifen kann. Es ist so auch möglich z. B. eine allgemeine Lookup-Funktion für Assoziationslisten zu schreiben.

4.2 Algebraische Sicht der Typdefinition

Die bisher bekannten Typsynonyme kapseln nur schon vorhandene Konstruktionen aus Typen. Sollen ganz neue Typen definiert werden, verwendet man das Schlüsselwort `data`.

Algebraisch gesehen hat die Typdeklaration die Form:

$$\texttt{data}\ cx \Rightarrow T\ u_1 ... u_k = K_1\ t_{11}\ ...\ t_{1k_1}\ \texttt{|}\ ...\ \texttt{|}\ K_n\ t_{n1}\ ...\ t_{nk_n}$$

Dabei ist cx ein Kontext wie z. B. `Eq a`. T ist ein „type constructor“, der wie in Abschnitt 4.1 Typvariablen u_i als Parameter enthalten kann. Auf der rechten Seite der Gleichung befinden sich ein oder mehrere „data constructor“, die wiederum die Parameter t_{ij} besitzt. Diese geben Typen an, mit deren Instanzen der „data constructor“ benutzt werden kann. In den folgenden drei Abschnitten werden verschiedene Ausbaustufen der Definition verwendet.

4.3 Aufzählungstypen

Zum Erstellen eines einfachen Aufzählungstypen benötigt man keinen Kontext cx und der „type constructor“ besteht nur aus dem Namen des Typen. Jede Alternative benötigt einen einfachen „data constructor“, der jedoch komplett ohne Parameter auskommt.

Beispiel 4.1. Es soll ein einfacher Typ `Wochentag` definiert werden, der alle sieben Tage der Woche enthält.

```
1  data Wochentag = Montag
2                 | Dienstag
3                 | Mittwoch
4                 | Donnerstag
5                 | Freitag
6                 | Samstag
7                 | Sonntag
```

Listing 4.3: Definition des Aufzählungstypen `Wochentag`

Die Alternativen müssen jeweils durch das Zeichen „|“ getrennt werden. Funktionen auf den Typ können sehr einfach durch den schon bekannten Mustervergleich realisiert werden.

Beispiel 4.2. Es soll der Vortag zu jedem Wochentag durch die Funktion `vortag :: Wochentag -> Wochentag` berechnet werden.

```
1  -- Realisierung durch Mustervergleich
2  vortag :: Wochentag -> Wochentag
3  vortag Montag = Sonntag
4  vortag Dienstag = Montag
5  vortag Mittwoch = Dienstag
6  ...
7
8  -- Noch einmal durch Fallunterscheidung
9  vortag' :: Wochentag -> Wochentag
10 vortag' x       | x == Montag = Sonntag
11                 | x == Dienstag = Montag
12                 | x == Mittwoch = Dienstag
13                 ...
```

Listing 4.4: Zwei Varianten um den Vortag zu jedem Wochentag anzugeben

Generell sehr hilfreich ist beim Mustervergleich das Stellvertretersymbol „_“. Dieses Symbol kann mehrfach für Variablen verwendet werden, deren Wert auf der rechten Seite einer Gleichung oder Definition nicht weiter interessiert. Beim Mustervergleich kann man erst alle „interessanten“ Fälle abarbeiten und dann für alle anderen Werte eine Standardrückgabe angeben.

Beispiel 4.3.

```
1 istWochenende :: Wochentag -> Bool
2 istWochenende Samstag   = True
3 istWochenende Sonntag   = True
4 istWochenende _         = False
```

Listing 4.5: Verwendung des Stellvertretersymbols „_“

Die Reihenfolge der Zeilen ist zu beachten, da das Muster „_“ auch auf `Samstag` und `Sonntag` passt. Diese Fälle müssen also vorher behandelt werden.

4.4 Parametrisierte Alternativen

Eine Erweiterung zu den normalen Aufzählungstypen geschieht durch die Nutzung von „data constructors“ mit weiteren Parametern. Dies wird benötigt, wenn in den

einzelnen Datenistanzen noch weitere Informationen gespeichert werden sollen.

Beispiel 4.4.

```
1  -- Datentyp Figur mit versch. Ausprägungen
2  data Figur =      Kreis Punkt Float
3          -- Kreis mit Mittelpunkt und Radius
4                 | Rechteck Punkt Punkt
5          -- Rechteck mit Endpunkten einer Diagonalen
6                 | Strecke Punkt Punkt
7          -- Strecke mit Start- und Endpunkt
```

Listing 4.6: Definition eines Typen für geometrische Figuren

4.5 Rekursive Typen

Wie schon bekannt von der Definition von Funktionen kann man den zu definierenden Typen auch auf der rechten Seite selbst verwenden, da als Parameter der „data constructors“ Typen erwartet werden. Eine Neuerung in dem folgenden Beispiel ist, dass auch der „type constructor“ einen Parameter bekommt. In diesem Fall ist dies die Typvariable `a`. Weiterhin wird auch noch ein Kontext *cx* hinzugefügt. In diesem Fall ist es der Kontext `Eq a`, wodurch von den Werten der Variable `a` gefordert wird, dass sie Instanzen der Typklasse `Eq` sind (also die Funktion `(==)` anbieten).

Beispiel 4.5.

```
1  -- Definition eines Datentypen für Mengen
2  data Eq a => Set a =
3          NilSet
4          | ConsSet a (Set a)
5          deriving Show
6
7  -- Instanz mit Zahlen
8  menge :: Set Integer
9  menge = ConsSet 1 (ConsSet 5 (NilSet))
10
11 -- Instanz mit Zeichenketten
12 namen :: Set String
13 namen = ConsSet "Liesel" (ConsSet "Herbert" (ConsSet "Hans" (NilSet)))
```

Listing 4.7: Definition eines Typen für eine Menge

Der Vorteil dieser so generellen Defintionsmöglichkeiten liegt darin, dass nun z. B. eine Funktion angegeben werden kann, die für jede Instanz von `Set a` (also jede mögliche Menge) deren Mächtigkeit bestimmt.

```
1 -- Bestimmen der Größe einer Menge
2 size :: Eq a => Set a -> Int
3 size NilSet = 0
4 size (ConsSet _ set) = (size set) + 1
```

Listing 4.8: Berechnung der Mächtigkeit von Mengen verschiedenen Typs

4.6 Typklassen für eigene Typen

Damit das Arbeiten mit neu definierten Typen erst möglich ist, müssen diese mindestens einen Vergleich auf Gleichheit und nach Möglichkeit sogar eine Ordnungsrelation bieten. Diese Eigenschaften sind in den Typklassen **`Eq`** (die Gleichheitsrelation `(==)`) und **`Ord`** (Ordnungsrelationen wie `(<)` oder `(>=)`) gegeben. Ein neuer Typen kann mit dem Schlüsselwort **`deriving`** diese Typklassen ableiten.

Beispiel 4.6.

```
1 data Wochentag = Montag
2         | Dienstag
3         | Mittwoch
4         | Donnerstag
5         | Freitag
6         | Samstag
7         | Sonntag
8         deriving (Eq, Ord)
```

Listing 4.9: Ableiten der Typklassen **`Eq`** und **`Ord`**

Die Ordnung der einzelnen „data constructor“ richtet sich nach der Reihenfolge, in der sie in der Typdefinition auftauchen. So gilt hier `Montag < Donnerstag`. Die Typklassen **`Ord`** und **`Eq`** können für alle benutzerdefinierten Typen abgeleitet werden, welche aus Typen aufgebaut sind, die auch schon die jeweilige Typklasse abgeleitet haben.

Besonders wichtig für die Ausgabe auf der Konsole oder in Dateien ist die Typklasse **`Show`**. Auch ihr Pendant **`Read`** ist wichtig beim Speichern von Informationen in Dateien, da der Typ zum wieder Einlesen die Funktion **`read`** benötigt. Wie im vorherigen Listing erweitert man zum Ableiten der Typklasse einfach die Liste nach dem Schlüsselwort **`deriving`**.

Beispiel 4.7.

```
1  data Zeit = Am
2          | Vor
3          | Bis
4          deriving (Eq, Ord, Show, Read)
5
6  data Planung = Schlafen Zeit Wochentag
7          | Essen Zeit Wochentag
8          | Studieren Zeit Wochentag
9          deriving (Eq, Ord, Show, Read)
```

Listing 4.10: Ableiten der Typklassen `Show` und `Read`

Es müssen wieder alle Typen die Typklasse ableiten, damit ein zusammengesetzter Typ dies tun kann. Eine weitere Typklasse ist hingegen nur für reine Aufzählungstypen (alle „data constructor“ parameterlos) abzuleiten. Es handelt sich um die Typklasse `Enum`. Sie bietet unter anderem die Funktionen `succ` und `pred`, welche den Nachfolger und den Vorgänger zu einem Wert bestimmen. Die Reihenfolge der Elemente ist so, wie bei der Verwendung von `Ord` beschrieben. Wenn für den Typ `Wochentag` die Typklasse `Enum` abgeleitet wird, sind Konstrukte wie `[Montag..Samstag]` möglich. Dieses Konstrukt erzeugt eine Liste mit allen Tagen von `Montag` bis `Samstag` als Elemente.

5 Module

Bei größeren Softwareprojekten kann es schnell zu einer Menge von Code, Funktionen und Typen kommen. Fast überall ist eine Trennung in Module oder Typklassen sinnvoll. Die Trennung hilft und unterstützt bei den folgenden Punkten:

- Vereinfachung des Designs durch Strukturierung
- Gleichzeitige und unabhängige Entwicklung durch gute Schnittstellen
- Probleme lassen sich schneller isolieren durch Softwaretests auf verschiedenen Ebenen
- Kapselung von internen Datenstrukturen und Funktionen wird möglich, dadurch lassen sich Module einfacher ersetzen
- Plattformunabhängigkeit z. B. durch Ersetzen der Module für die Ein- und Ausgabe
- Wiederverwendbarkeit von Komponenten

Module müssen mit Großbuchstaben beginnen (wie z. B. **`Prelude`** oder `Simple` s. o.). Ein Modul sollte mit einem Kommentar starten, der den Zweck, den Autor und eine kurze Beschreibung enthält.

```
1  -- Kontrollstruktureinführung
2  -- Jan Oliver Ringert
3  -- Das Modul bietet eine Einführung in die
4  -- Kontrollstrukturen und Syntax von Haskell
```

Listing 5.1: Beispiel eines Kommentars, der ein Modul einleitet

Damit ein Modul anderen Modulen Informationen zur Verfügung stellen kann, müssen diese von dem Modul exportiert werden. Benötigt ein zweites Modul diese Informationen, so können sie über den Import verwendbar gemacht werden.

An dieser Stelle wird nur grundlegend auf den Im- und Export in Haskell eingegangen. Eine genauere Beschreibung, die den kompletten Syntax und große Teile der informellen Semantik enthält, ist in [5] zu finden.

5.1 Export

Jedes Modul kann seine Funktionen und Datenstrukturen anderen zur Verfügung stellen. Wenn der Entwickler keine Angaben macht, dann werden alle Funktionen, Datenstrukturen und Typklassen (kurz für alle: Entitäten) auf oberster Ebene des Moduls exportiert. Importierte Entitäten werden nicht automatisch exportiert.

Über die Angabe einer Exportliste, können alle auf erster Ebene definierten und auch importierte Entitäten von dem Modul exportiert werden.

Beispiel 5.1.

```
1  module Supermarkt (
2  Cent, Produkt, PreisListe, EinkaufsListe, kosten
3  ) where
4
5  type Cent = Int
6  type Produkt = String
7
8  -- Liste mit Einträgen der Form (Produkt, Preis)
9  type PreisListe = [(Produkt, Cent)]
10
11 -- Liste mit Einträgen der Form (Produkt, Menge)
12 type EinkaufsListe = [(Produkt, Int)]
13
14 ... -- Restliche Funktionen
15
16 -- Berechnet die Gesamtkosten eines Einkaufs
17 kosten :: PreisListe -> EinkaufsListe -> Cent
18 kosten ps es = gesamtPreis (preisMenge ps es)
```

Listing 5.2: Bestimmte Entitäten exportieren

In Zeile zwei ist die Exportliste zu sehen, die in runde Klammern eingeschlossen wird. Hier werden nur Typsynonyme und die Funktion zur Berechnung von Kosten exportiert. Weiter Funktionen, wie das Heraussuchen von Preisen aus der Liste sind nach Außen nicht zu sehen.

Beispiel 5.2.

```
1  module Set (
2     Set (NilSet), size, isIn, add
3  ) where
4
5  -- Definition eines Datentypen für Mengen
6  data Eq a => Set a = NilSet | ConsSet a (Set a)
```

```
 7         deriving Show
 8
 9 -- Bestimmen der Größe einer Menge
10 size :: Eq a => Set a -> Int
11 size NilSet = 0
12 size (ConsSet _ set) = (size set) + 1
13
14 ...
```

Listing 5.3: Nur Teile der „data constructor“ exportieren

In diesem Beispiel wird gezeigt, dass es unterschiedliche Möglichkeiten gibt, Typen zu exportieren. Durch die oben angegebene Syntax `Set (NilSet)` wird der Typ `Set a` exportiert und mit ihm nur der Konstruktor `NilSet`. Es ist also dem Benutzer möglich, eine leere Menge zu erstellen. Durch die weiteren exportierten Funktionen kann er diese dann verwenden. Der interne Aufbau der Menge (bzw. des Typen `Set a`) ist dem Benutzer nicht bekannt. Alle Konstruktoren können exportiert werden, indem man `Set (..)` in die Exportliste aufnimmt. Wenn nur der Typname exportiert werden soll, gibt man lediglich `Set` an.

5.2 Import

Eine weitere Selektion ist auch beim Importieren von Modulen möglich. Durch eine explizite Angabe, welche Teile eines Moduls verwendet werden, ist eine Untersuchung der Abhängigkeiten zwischen Modulen oft einfacher. Das komplettes Modul `Set` (alle Exporte des Moduls) wird mit der Zeile **`import`** `Set` nach dem **`where`** importiert. Alle Entitäten sind nun qualifiziert über den Modulnamen oder ein Synonym verwendbar (z. B. `Set.size Set.NilSet`). Sie sind gleichzeitig auch nicht qualifiziert (z. B. `size NilSet`) in dem importierenden Modul zu benutzen. Ist es gewünscht, dass auf die Entitäten nur qualifiziert zugegriffen werden kann, dann kann dies durch das Schlüsselwort **`qualified`** (also **`import qualified`** `Set`) gefordert werden.

Wenn es gewünscht ist, nur bestimmte Entitäten zu importieren, werden diese in einer Liste ähnlich der Exportliste angegeben. Dort dürfen nur Entitäten aufgeführt werden, die auch von dem Zielmodul exportiert werden. Es kann weiterhin sinnvoll sein, bestimmte Entitäten auszuschließen, aber alle anderen zu importieren. Dies ist mit dem Schlüsselwort **`hiding`** möglich.

Beispiel 5.3.

```
1 module A
2 where
3 import qualified Set
4 import Supermarkt (Cent, Produkt)
```

```
5  import Prelude hiding (print, error)
6
7  ...
```

Listing 5.4: Möglichkeiten zum Import von Modulen und darin definierten Entitäten

- In Zeile drei werden alle Entitäten des Moduls `Set` importiert. Diese sind in dem Modul `A` nur qualifiziert zu verwenden.
- Aus dem Modul `Supermarkt` werden nur die Typsynonyme `Cent` und `Produkt` importiert.
- Das Modul **`Prelude`** wird von jedem Modul standardmäßig importiert. Wenn bestimmte Funktionen ausgeschlossen werden sollen, ist dies wie in Zeile fünf möglich.

5.3 Namen und Geltungsbereich

In Haskell gibt es sechs verschiedene Arten von Namen:

- Namen für Variablen und „data constructor“ welche Werte bezeichnen
- Namen für Typvariablen, „type constructor“ und Typklassen welche alle drei Elemente des Typsystems bezeichnen
- Modulnamen für die Identifikation von Modulen

Es gibt für diese sechs Arten von Namen zwei Regeln, die bei der Vergabe der Namen eingehalten werden müssen.

1. Namen für Variablen und Typvariablen beginnen mit einem Kleinbuchstaben oder einem Unterstrich. Alle anderen Namen beginnen mit einem Großbuchstaben.
2. Ein „type constructor“ und eine Typklasse dürfen nicht den selben Bezeichner in einem Geltungsbereich haben.

Der Geltungsbereich von einer neu definierten Funktion oder Variablen lässt sich grob einteilen in lokal und global. In einem Modul sind alle auf erster Ebene definierten Variablen und Funktionen global. Das bedeutet, es kann innerhalb des Moduls von überall auf sie zugegriffen werden.

Im Gegensatz dazu gibt es die lokalen Variablen bzw. Funktionen, die z. B. nur in einem **`do`**-Block gültig sind oder bei der Verwendung von **`let`** und **`where`**.

- Variablen, die in einem `do`-Block an einen Wert gebunden werden, sind nur nach der Zeile mit ihrer Bindung und nur innerhalb des Blocks gültig.
- Der Gültigkeitsbereich der Definitionen d_i in `let` {d_1, d_2, ... } `in` *exp* ist nur innerhalb von {... d_{i+1}, d_{i+2}, ... } und *exp*.
- Der Gültigkeitsbereich von Bezeichnern, die innerhalb einer **`where`**-Klausel definiert wurden, ist die gesamte Definition, auf die sich die **`where`**-Klausel bezieht. Der Bezug wird durch Einrücken festgelegt.

Nur Bezeichner mit globalem Geltungsbereich können von einem Modul exportiert werden.

6 Ein-/Ausgabe

6.1 Einleitung

Alles was wir bisher in Haskell geschrieben haben, waren Funktionen, die Werte auf andere Werte abbilden. Dieser Funktionen konnten wir direkt in der interaktiven Shell mit Parametern aufrufen.

Die einzige Ausgabe, die wir daraufhin erhalten haben, war der Rückgabewert der Funktion. Dies ist relativ unhandlich zu benutzen. Außerdem ist die Shell nicht die einzige Möglichkeit Daten mit der realen Welt auszutauschen. Wie steht es beispielsweise um Dateien, Netzwerke etc.?

Das was wir bisher behandelt haben, wird als „pure“ bezeichnet, d.h. Code der keinerlei Seiteneffekte besitzt. Code mit Seiteneffekten wie Ein-/Ausgaben wird entsprechend als „impure“ bezeichnet. In funktionalen Programmiersprachen trennt man sehr strikt zwischen diesen beiden Arten.

Wie bringt man nun „reinen“ und „unreinen“ Code zusammen?

Sehen wir uns zunächst einmal den typischen Ablauf bei der Ein-/Ausgabe in Bild 6.1 an. Von links nach rechts werden nacheinander Funktionen (F_i) ausgeführt. Diese haben Ein- und Ausgaben (E_i, A_i).

Die Eingaben sind vom Benutzer (oder auch aus Dateien) gelesene Werte. Die Ausgabe ist ein auszuführendes Kommando (read, write, readFile, ...). Außerdem wird an das Laufzeitsystem eine Funktion F_{i+1} übergeben, die nach Beendigung des Kommandos mit dem Ergebnis des Kommandos (E_{i+1}) ausgeführt werden soll.

So lässt sich eine Kette von Ein- und Ausgaben mit verarbeitender Logik darstellen.

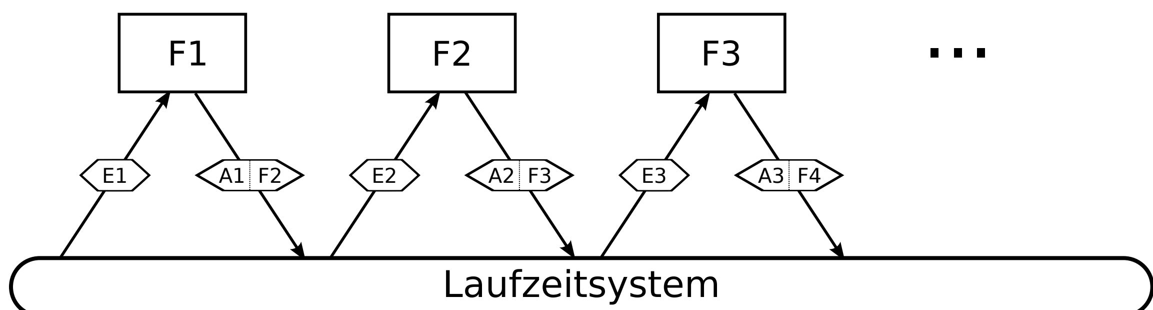

Abbildung 6.1: Typischer Ablauf bei der Ein-/Ausgabe ([12])

Dies entspricht auch genau dem Ansatz zur Darstellung von Ein-/Ausgabeabläufen

in Haskell. Die Verkettung von den Funktionsaufrufen wird dabei durch spezielle Datentypen vorgenommen.

Aber sehen wir uns erstmal ein Beispiel an.

6.2 Ablaufsteuerung und Auswertungsreihenfolge

```
1  echo :: IO ()
2  echo = do
3          putStr "Bitte etwas Eingeben: "
4          line <- getLine
5          putStr line
```

Listing 6.1: Ausgeben einer eingelesenen Zeile

Dies sieht überraschenderweise sehr ähnlich zu einem Ablauf in einer imperativen Programmiersprache aus: es werden nacheinander Aktionen ausgeführt. Die Notation `line <- getLine` bedeutet in etwa „weise an line das Ergebnis von getLine“ zu. `do` leitet eine Sequenz von Ein-/Ausgabeoperationen ein.

Wie passt diese Notation nun zu unserem Bild oben?

Dazu sehen wir uns mal die Typen von putStr und getLine an:

```
1  > :type putStr
2  putStr :: String -> IO ()
3  > :type getLine
4  getLine :: IO String
```

`putStr` übernimmt wie erwartet einen String. Diesen übergeben wir in Zeile 3. Züruckgeben tut sie einen Wert vom Typ `IO ()`. Was bedeutet das?

`IO a` ist eine Ein-/Ausgabeaktion, also ein Kommando an das Laufzeitsystem. `putStr "Hallo"` erzeugt also ein solches Kommando, führt es aber *nicht* aus. Dieses IO-Objekt kann nun innerhalb einer anderen Aktion ausgeführt werden.

Der Typ von `getLine` sieht nun noch etwas merkwürdiger aus. `IO String` bedeutet, dass ein Kommando gekapselt wird, das einen String als Ergebnis hat. Damit erklärt sich auch der vollständige Typ von `putStr`: die reine Ausgabe besitzt kein Ergebnis. Dies durch ein leeres Tupel `()` ausgedrückt.

`echo` ist also eine Funktion, die ein Objekt vom Typ `IO ()` erzeugt. Dieses wiederum kapselt das Kommando `putStr`. Wenn das ausgeführt worden ist, wird `getLine` ausgeführt usw.

Beispiel 6.1. Im Folgenden soll nocheinmal die Definition von `getLine` gegeben werden. Diese ist aber auch in Prelude definiert.

```
1 getLine :: IO String
2 getLine =
3    do
4     c <- getChar
5     if c == '\n'
6        then return ""
7        else do s <- getLine
8     return (c:s)
```

Listing 6.2: Definition der Funktion **getLine**

Hier sieht man gut das Starten einer neuen Aktion aus einer beendeten heraus. **getLine** ruft **getChar** auf. Das Ergebnis dieses Aufrufs wird auf **'\n'** geprüft. Solange dies nicht gefunden wird, wird **getLine** rekursiv aufgerufen.

Man beachte auch das etwas versteckte **do** hinter dem **else**. Dies ist notwendig, da die beiden Zweige beim **if** den gleichen Typ haben müssen. Hier wird der Typ **IO String** erwartet. Das **return ""** hat bereits diesen Typen. Das **getLine** mit der Bindung an s muss diesen erst noch bekommen.

Ein kleiner Hinweis zu der Rekursion. In einer imperativen Programmiersprache würde man erwarten, dass der Aufrufstack bei entsprechend langen Eingaben viel zu groß werden würde. In funktionalen Programmiersprachen kann der Compiler Endrekursionen, d.h. Rekursionen die nach allen anderen Operationen ausgeführt werden, sehr gut optimieren.

6.3 I/O-Funktionen

Somit können wir nun leicht zwischen Code unterscheiden, der für Ein- und Ausgabe zuständig ist und Code, der lediglich Berechnungen ausführt. Funktionen für die Ein-/Ausgabe besitzen nämlich **IO** a als Rückgabetyp.

Weiter können aber auch reine Ausgabefunktionen unterschieden werden: diese besitzen den Rückgabetyp **IO** (). Dazu ein paar Beispiele aus **Prelude**:

```
1 putChar :: Char -> IO ()
2 putStr :: String -> IO ()
3 putStrLn :: String -> IO () -- adds a newline
4 print :: Show a => a -> IO ()
```

Listing 6.3: Einige Ausgabefunktionen des Prelude Moduls

Die Typvariable a ist hier ein leeres Tupel, da diese Funktionen nur Aktionen ausführen und keinen Wert zurück liefern können. Eine Eingabe von Daten durch

den Benutzer ist mit den Funktionen im nächsten Listing möglich, welche ebenfalls von **Prelude** bereitgestellt werden.

```
1 getChar :: IO Char
2 getLine :: IO String
3 getContents :: IO String
4 interact :: (String -> String) -> IO ()
5 readIO :: Read a => String -> IO a
6 readLn :: Read a => IO a
```

Listing 6.4: Einige Eingabefunktionen des Prelude Moduls

Die meisten der obigen Funktionen sind von ihrem Namen und ihrer Signatur her verständlich. Kurze Erklärungen finden sich in [5]. Nicht unbedingt auf den ersten Blick verständlich sind die folgenden Funktionen:

- **getContents**: Alle Eingaben des Benutzers werden eingelesen und als eine einzige Zeichenkette zurück geliefert. Diese Funktion arbeitet „lazy“, also nur bei Bedarf.
- **interact**: Diese Funktion akzeptiert als ersten Parameter eine Funktion, welche von **String** nach **String** abbildet. Die Funktion **interact** leitet die Eingabe des Benutzers an die übergebene Funktion weiter und ihr Ergebnis in die Standardausgabe des Benutzers um.
- **readIO**: Diese Funktion versucht durch Parsen einer Zeichenkette einen Wert vom Typ **IO** a zu erzeugen, der, wie in 6.2 beschrieben, mit <- einer Variablen zugewiesen werden kann (Listing 6.5). a muss dabei von der Typklasse **Read** ableiten, und damit die Funktion **read** anbieten (siehe Abschnitt 6.5). Gelingt es nicht, die Zeichenkette zu parsen, führt **readIO** im Gegensatz zu **read** nicht zu einem Programmabbruch, sondern signalisiert einen Parserfehler an die IO-Monade

6.4 Datei-Ein-/Ausgabe

Um in Haskell in Dateien zu schreiben und aus diesen zu lesen, stehen die Funktionen aus Listing 6.6 zur Verfügung.

Die Funktionen **writeFile** und **appendFile** schreiben eine Zeichenkette in eine Datei bzw. hängen diese am Ende der Datei an. Wenn die Datei noch nicht existiert, wird sie angelegt. Es können nur Zeichenketten in die Dateien geschrieben werden. Falls andere Typen in Dateien geschrieben werden sollen, müssen diese vorher umgewandelt werden. Zum Umwandeln kann die in Abschnitt 6.5 vorgestellte Funktion **show** verwendet werden.

```
1  addTwoInteger = do
2          input1 <- getLine
3          -- versuche, input1 als Integer zu parsen.
4          -- Moeglich, da Integer von Read ableitet
5          -- und damit read implementiert
6          zahl1 <- readIO input1::IO Integer
7          -- nochmal mit zweiter zahl
8          input2 <- getLine
9          zahl2 <- readIO input2::IO Integer
10         -- Ausgabe mit Verwendung der
11         -- Funktion show
12         putStrLn (show(zahl1 + zahl2))
```

Listing 6.5: Verwendung von readIO

```
1  type FilePath = String
2  writeFile :: FilePath -> String -> IO ()
3  appendFile :: FilePath -> String -> IO ()
4  readFile :: FilePath -> IO String
```

Listing 6.6: Funktionen zum Arbeiten mit Dateien

Zum Lesen aus Dateien kann man die Funktion **readFile** verwenden. Die Funktion ist nach dem Prinzip der Bedarfsauswertung definiert und liest nur so viele Zeichen, wie benötigt werden.

Beispiel 6.2.

Eine Datei wird geöffnet und gelesen. Da das Ergebnis komplett mit der Funktion **print** ausgegeben wird, muss der gesamte Inhalt gelesen werden. Zeilenumbrüche werden bei der Ausgabe auf der Konsole umgewandelt in „\n“.

```
1  main = do
2     inhalt <- readFile "readFile.hs"
3     print inhalt
```

Listing 6.7: Lesen aus einer Datei

6.5 Werte darstellen und lesen

Das Schreiben von Daten in Dateien ist erst dann sinnvoll, wenn in einem Programm darstellbare Werte auch wieder eingelesen werden können. Die eben vorgestellten Funktionen schreiben und lesen jedoch nur Zeichenketten (`type String = [char]`). Auch auf der Konsole der interaktiven Shell können nur Zeichenketten ausgegeben bzw. von ihr gelesen werden. Die Umwandlung eines Typs in eine Zeichenkette geschieht in der Shell durch die Funktion `show` des entsprechenden Typen. Der Typ muss dafür von der Typklasse `Show` ableiten.

Die Signatur der Funktion `show` ist `show :: Show a =>a -> String` und sie besitzt ein Gegenstück, mit dem Werte auch wieder aus Zeichenketten eingelesen werden können. Dieses Gegenstück ist die Funktion `read :: Read a =>String -> a` aus der Typklasse `Read`.

Wenn eigene Typen ausgebbar bzw. einlesbar gemacht werden sollen, müssen diese von den Typklassen `Show` bzw. `Read` ableiten. Sie können dann mit der Funktion `print` ausgegeben bzw. mit `readIO` aus eingelesenen Zeichenkentten erzeugt werden. (siehe Listing 6.8).

```
 1 data Wochentag = Montag | Dienstag | Mittwoch | Donnerstag
 2         | Freitag | Samstag | Sonntag
 3         deriving (Eq, Ord, Read, Show)
 4
 5 fruehesterWochentagShell = do
 6         tag1eingabe <- getLine
 7         tag1 <- readIO tag1eingabe::IO Wochentag
 8         tag2eingabe <- getLine
 9         tag2 <- readIO tag2eingabe::IO Wochentag
10         if tag1 > tag2 then print tag2 else print tag1
```

Listing 6.8: Eigene Typen einlesen und ausgeben

6.6 Ausführbare Programme erzeugen

Mit den Funktionen zu Ein- und Ausgabe ist es nun möglich, eigenständige Programme zu schreiben, die für die Eingabe von Daten nicht mehr auf die interaktive Shell des Interpreters angewiesen sind. Damit der Compiler weiß, welche Funktion zum Start des Programmes ausgeführt werden soll, muss diese im Modul `Main` den Namen `main` tragen und vom Typ `IO a` sein. Für die Typvariable `a` wird meist der Typ `()` eingesetzt. Die Werte anderer Typen, welche zurückgegeben werden, werden bei der Ausführung einfach ignoriert.

Beispiel 6.3.

```
1 main :: IO ()
2 main = do
3         putStr "Hallo␣Welt!"
```

Listing 6.9: Ein Programm in Haskell

Damit aus dem Quellcode eine ausführbare Datei wird, muss der Haskell-Compiler `ghc` benutzt werden. Zum Kompilieren der Datei `hello.hs` lautet die Kommandozeile `ghc -o hello hello.hs`. Der Haskell Compiler erzeugt unter Windows die ausführbare Datei `hello.exe`, welche ohne den Haskell Interpreter gestartet werden kann.

6.7 Weiteres zur Ein-/Ausgabe

Soweit haben wir gesehen, wie mit **do** Aktionen für die Ein- und Ausgabe verknüpft werden können. Dies sieht im bisherigen Stil nicht funktional, sondern eher imperativ aus. Vielleicht fragt man sich auch, wie die IO-Objekte zusammengesetzt werden.

Wir wollen uns dazu unser erstes Beispiel nochmal angucken.

```
1 echo :: IO ()
2 echo = do
3         putStr "Bitte␣etwas␣Eingeben:␣"
4         line <- getLine
5         putStr line
```

Listing 6.10: Ausgeben einer eingelesenen Zeile mit do

Das **do** ist eigentlich eine Abkürzung, die durch den Compiler ersetzt wird. Wollen wir uns mal ansehen, was der Compiler daraus macht. Die Version ist allerdings lesbarer, als das, was vom Compiler automatisch generiert wird.

```
1 echo :: IO ()
2 echo = putStr "Bitte␣etwas␣Eingeben:␣" >>
3        getLine >>=
4        (\line -> putStr line)
```

Listing 6.11: Ausgeben einer eingelesenen Zeile ohne do

Das sieht doch schon wieder eher nach einer funktionalen Programmiersprache aus. Die Operatoren `(>>)` und `(>>=)` können als Verkettung von IO-Aktionen verstanden werden.

Sehen wir uns einmal die zugehörige Signatur an. Tatsächlich kommt in der Signatur **IO** nicht mehr vor, aber das soll uns hier nicht interessieren.

```
1 (>>) :: IO a -> IO b -> IO b
2 (>>=) :: IO a -> (a -> IO b) -> IO b
```

Der Operator `(>>)` übernimmt zwei IO-Objekte und verbindet sie zu einem, wobei das Ergebnis der ersten Aktion verworfen wird.

Dagegen übernimmt `(>>=)` ein IO-Objekt und eine Funktion. Das Ergebnis der ersten Aktion wird an die Funktion übergeben. Diese konstruiert nun mit diesem Parameter ein neues IO-Objekt, das das Ergebnis der gesamten Aktion ist.

Man beachte, dass die Funktion natürlich erst ausgewertet wird, nachdem die erste Aktion ausgeführt wurde.

Analysieren wir damit mal das neu geschriebene `echo`. **`putStr`** wird mit **`getLine`** verknüpft (die Operatoren sind linksassoziativ). **`putStr`** gibt ein Objekt vom Typ **`IO`** `()` zurück, **`getLine`** dagegen vom Typ **`IO String`**. Da `(>>)` nur das Ergebnis von **`getLine`** übernimmt, entsteht ein Objekt vom Typ **`IO String`**.

Gucken wir uns nun den Typ von dem Lambda-Ausdruck an.

```
1 \line -> putStr line :: String -> IO ()
```

Da **`putStr`** einen String erwartet, erwartet der Ausdruck ebenfalls einen String. `(>>=)` verknüpft nun die linke Seite vom Typ **`IO String`** mit dem Lambda-Ausdruck.

Das heißt es entsteht ein IO-Objekt vom Typ **`IO`** `()` (dem Rückgabetyp des Lambda-Ausdrucks). Dieses reicht zur Laufzeit das Ergebnis von **`getLine`** an den Lambda-Ausdruck weiter. Dieser konstruiert daraufhin mit **`putStr`** die Ausgabe-Aktion.

Der Operator `(>>=)` entspricht damit genau der Eingangs gesuchten Verknüpfung zwischen Aktionen durch Funktionen. Der Operator `(>>)` stellt in dem Modell aus Abschnitt 6.1 eine Verknüpfung mit der Identitätsfunktion zur Verfügung.

7 Anwendung: Parsen mit Parsec

Im Folgenden soll ein Parser für CSV-Dateien vorgestellt werden, der das Parser-Framework Parsec [6]. Auf der zugehörigen Internetseite findet sich auch die vollständige Dokumentation.

```
1  module CSVParser (parseCsv) where
2  import Text.ParserCombinators.Parsec
3
4
5  type CsvLine = [String]
6
7  parseCsv :: Char -> String -> (Either ParseError [CsvLine])
8  parseCsv d input = parse (csvlines d) "CSVParserError" input
9
10 csvlines :: Char -> GenParser Char () [CsvLine]
11 csvlines d = (csvvalues d) ‘sepBy‘ eol
12
13 csvvalues :: Char -> GenParser Char () CsvLine
14 csvvalues d = do
15         val <- csvvalue d
16         rest <- csvvaluesrest d
17         return (val:rest)
18
19 csvvaluesrest :: Char -> GenParser Char () CsvLine
20 csvvaluesrest d =
21     (char d >> csvvalues d)
22     <|> return []
23
24 csvvalue :: Char -> GenParser Char () String
25 csvvalue d = many1 (noneOf (d:"\n\r"))
26
27 eol :: GenParser Char () String
28 eol =
29         try (string "\r\n")
30         <|> try (string "\n\r")
31         <|> string "\r"
32         <|> string "\n"
```

Listing 7.1: CSV-Parser

Zeile 2: Import von Parsec

Zeile 7-8: Funktion um die Benutzung zu erleichtern. Hier wird die Funktion parse von Parsec benutzt. Diese übernimmt als Parameter einen Parser, einen zusätzlichen Fehlertext und die Eingabe.

Der eigentliche Parser besteht aus den einzelnen Funktionen, die von Zeile 10 bis 32 definiert werden und die Elemente einer CSV-Datei wiederspiegeln.

Ein CSV-Dokument besteht aus mehreren durch Zeilenumbrüche (`eol`) getrennte Zeilen. Dies ist durch die Funktion `csvlines` dargestellt (Zeile 10 - 11). `sepBy` ist eine Funktion von Parsec, die zwei Parser übernimmt. Sie erzeugt einen Parser, der eine Liste von den Elementen des ersten Parsers enthält. Das Trennzeichen (2. Argument) wird verworfen.

Eine Zeile besteht wiederum aus mindestens einem Feld. Die Felder werden durch das Trennzeichen (Parameter `d`) getrennt. `csvvalues` parst zuerst einen Wert (`csvvalue`), der wiederum wieder mindestens ein Zeichen enthält. Danach wird versucht ein Trennzeichen zu lesen und wenn dieses gefunden wurde, werden weitere Felder gelesen (`csvvaluerest`).

Die Notation ist analog zu der der Ein-/Ausgabe: Ausdrücke werden mit Hilfe von **`do`** verknüpft und Zuweisungen werden mit `<-` ausgedrückt. Entsprechend werden die von den Ausdrücken erzeugten Parser nach dem **`do`** sequenziell ausgeführt. Wenn einer dieser Parser fehlschlägt, bricht der aufrufende Parser ab.

Zeile 15: Einlesen eines Wertes bis zum Trennzeichen d oder einem Zeilenumbruch

Zeile 16: Einlesen der restlichen Werte, diese werden in Zeile 17 mit `val` verknüpft zurückgegeben.

Zeile 21 - 22: Es wird versucht ein Trennzeichen einzulesen. Wenn dies gefunden wird, dann werden weitere Werte eingelesen. Sollte das Trennzeichen nicht gefunden werden, wird als Rest eine leere Liste zurückgegeben.

Der Operator `<|>` definiert Alternativen, die ausgewertet werden, wenn der Parser auf der linken Seite kein Zeichen verbraucht hat. Das bedeutet, dass der Parser bereits beim ersten Zeichen abgebrochen hat. Sollte er mehr als ein Zeichen vor dem Abbruch analysiert haben, sind diese Zeichen in den nachfolgenden Parsern nicht mehr verfügbar, sie sind verbraucht. Wenn man dies verhindern möchte, muss der Parser in ein **`try`** eingebettet werden. Durch das **`try`** wird somit ein *Lookahead* ermöglicht. Diese Besonderheit wird bei `eol` deutlich.

Zeile 25: Es wird mindestens ein Zeichen gelesen, das nicht das Trennzeichen, „\n“ oder „\r“ ist.

Zeile 28-32: Hier wird versucht ein Zeilenende zu lesen. Da die ersten beiden Alternativen aus mehreren Zeichen bestehen, wird `try` verwendet.

Die von den Funktionen erzeugten Parser haben die Signatur `GenParser Char ()a`. Dabei ist `Char` der Typ der zu parsenden Zeichen. `()` kann vorerst ignoriert werden (hier könnte ein eigener Zustand gespeichert werden). Das wichtigste ist `a`. Hier wird der Typ des Rückgabewertes des Parsers festgelegt.

8 Anwendungen: Bäume

Viele Anwendungen, welche Datenstrukturen verwenden, benötigen auch Zugriffe auf enthaltene Daten. Diese Zugriffe sind je nach der Art der Daten und ihrer Anordnung stark unterschiedlich. Ein Zugriff auf ein bestimmtes Element in einer unsortierten Assoziationsliste hat eine Laufzeitkomplexität von $O(n)$. In einer sortierten Liste kann man mit der Binärsuche (Telefonbuchsuche) eine Laufzeitkomplexität in $O(log\ n)$ erreichen. Leider lässt sich dieses Verfahren nicht mit linearen Listen verwenden, wie sie in Haskell typischerweise benutzt werden. Hier müssen zum Ansteuern eines bestimmten Elementes in der Liste normalerweise alle vorherigen durchlaufen werden.

8.1 Einfache Bäume zum Speichern von Daten

Eine Datenstruktur, welche oft zum Organisieren und Ablegen von Daten benutzt wird, sind Bäume. Die Idee dieser rekursiven Datenstruktur ist, dass ein Wurzelknoten und innere Knoten existieren, welche jeweils mehrere Kinderknoten als Nachfolger besitzen können. Neben den beiden genannten Knotenarten existieren noch die so genannten Blätter, welche keine Nachfolger mehr besitzen.

Ein Baum, dessen Knoten maximal zwei Nachfolger besitzen, wird binär genannt. Binärbäume können nach ihren Elementen sortiert werden. Typischerweise enthält das linke Kind eines Knoten einen Wert, der kleiner ist als der im Knoten gespeicherte und das rechte Kind einen, der größer ist. Ein wichtiges Merkmal von Bäumen ist ihre Tiefe. Die Tiefe ist für jeden Knoten definiert und die Tiefe des Baumes ist die Tiefe des Wurzelknotens.

- Die Tiefe eines Blattes ist 0.
- Die Tiefe eines Knotens ist um eins höher als die maximale Tiefe seines rechten und linken Teilbaums.

8.2 AVL-Bäume

Neben der Zeit, ein Element zu finden, ist es auch wichtig, die Komplexität des Einfügens von neuen Elementen oder des Löschens von bestehenden zu betrachten. Bezogen auf diese Operationen sind AVL-Bäume sehr effiziente Datenstrukturen.

```
1  -- Einfaches Typsynonym für die Balance des Baums
2  type Balance = Int
3
4  -- AVL-Baum mit der jeweiligen Balance (-1, 0 oder 1)
5  data AVLBaum a = Knoten Balance a (AVLBaum a) (AVLBaum a)
6                 | Blatt
7                 deriving (Show)
```

Listing 8.1: Definition des Datentypen eines AVL-Baums

Es gibt zwei Eigenschaften, die ein binärer Baum besitzen muss, damit er ein AVL-Baum ist:

1. Die Tiefe des rechten und des linken Teilbaums unterscheiden sich höchstens um eins.

2. Beide Teilbäume sind AVL-Bäume.

Diese Eigenschaften können beim Einfügen und Löschen durch verschiedene Rotationen von Teilbäumen wiederhergestellt werden. Eine gute Erklärung dieser Operationen mit theoretischen Betrachtungen und einer Implementierung in Haskell findet sich in [9].

8.3 Trie-Strukturen

Für Anwendungen wie das Speichern von vielen Zeichenketten, sind andere Datenstrukturen als binäre Bäume sinnvoller. In einem AVL-Baum müsste man alle Wörter einzeln in den Knoten speichern. Die Wörter lassen sich jedoch auch durch ihre Position in einem Baum bestimmen, wenn jede Kante einen Buchstaben enthält. Soll ein Wort gefunden werden, wird den Buchstaben gefolgt, bis dies entweder nicht mehr möglich ist, oder ein Knoten gefunden wurde, der das Ende eines Wortes definiert. Ob an einem gegebenen Knoten ein Wort endet, wird über einen Wert vom Typ `Bool` angegeben.

```
1  -- Datentypdefinition für einen Trie
2  data Trie = TrieNode Bool [(Char,Trie)]
3          deriving (Show)
```

Listing 8.2: Definition des Datentypen eines Tries

In der obigen Datentypdefinition fällt auf, dass alle Knoten gleich sind und es keine expliziten Blätter mehr gibt. Blätter lassen sich jedoch dadurch erkennen, dass sie eine leere Liste von Kinderknoten haben und der Knotenwert `True` ist.

8.3.1 Suchen und Einfügen von Wörtern

Das Suchen von Wörtern in dem Baum ist ein Absteigen entlang der Kante mit dem nächsten Buchstaben aus dem Wort. Es werden also für ein Wort mit n Buchstaben n Knoten nach der Wurzel besucht. Es gibt zwei Möglichkeiten für ein negatives Ergebnis der Suche:

- Von einem Knoten aus geht keine Kante mit einem entsprechenden Buchstaben aus dem Wort weiter.
- Die Suche endet in einem Knoten, der den Wert `False` enthält. Das gesuchte Wort ist also kein vollständiges Wort.

Das Einfügen von Wörtern in eine Trie-Struktur ist ähnlich zu dem in einem binären Suchbaum. Es findet keine Balancierung des Baumes statt, selbst wenn dieser entarten sollte. Eine Balancierung ist nicht möglich, da in dem Trie die Reihenfolge der Knoten den Inhalt bestimmt. Daraus resultiert, dass das Einfügen eines Wortes so viele Schritte erfordert, wie die Länge des Wortes beträgt. Das Einfügen ist aufgeteilt in zwei Schritte:

1. Suche nach dem wachsenden Prefix des Wortes im Baum, bis das Wort gefunden ist, oder ein Rest des Wortes übrig bleibt.
   a) **Wenn kein Rest übrig ist:** Setze den Wert des bestimmten Knoten auf `True`, um anzuzeigen, dass dort ein Wort endet.
   b) **Sonst:** Füge den Rest des Wortes als Kindknoten hinter dem aktuellen Knoten ein, bis das Wort in einem Knoten `TrieNode True []` endet.

8.3.2 Löschen von Einträgen

Wenn ein Eintrag aus einem `Trie` entfernt werden soll, reicht es unter Umständen nicht, nur das letzte Blatt mit seiner Kante zu entfernen. Ein `Trie`, der nur die beiden Worte „`Arm`“ und „`Armenhaus`“ enthält, würde dann noch Knoten mit Kanten für `e`, `n`, `h`, `a`, `u` und `s` enthalten. Es wäre zwar kein Teilwort daraus zu finden, da alle Knoten in der Kette `False` enthalten, es sollte jedoch etwas besser „aufgeräumt“ werden. Ein `Trie` darf nur Blätter mit dem Wert `True` enthalten.

Literatur

[1] *The Hugs 98 User's Guide.* : *The Hugs 98 User's Guide.* http://cvs.haskell.org/Hugs/pages/users_guide/haskell98.html

[2] Barendregt, Henk: Lambda Calculi with Types. Version: 1992. citeseer.ist.psu.edu/barendregt92lambda.html. In: *Handbook of Logic in Computer Science, Volumes 1 (Background: Mathematical Structures) and 2 (Background: Computational Structures), Abramsky & Gabbay & Maibaum (Eds.), Clarendon* Bd. 2. 1992

[3] Chakravarty, Manuel M.; Keller, Grabriele C.: *Einführung in die Programmierung mit Haskell.* 1. München: Pearson Education Deutschland GmbH, 2004

[4] Hudak, Paul; Hughes, John; Jones, Simon P.; Wadler, Philip: A history of Haskell: being lazy with class. In: *HOPL III: Proceedings of the third ACM SIGPLAN conference on History of programming languages.* New York, NY, USA: ACM, 2007. – ISBN 978–1–59593–766–X, S. 12–1–12–55

[5] Jones, Simon P.: *Haskell 98 Language and Libraries - The Revised Report*, 2003. http://haskell.org/definition/haskell98-report.pdf

[6] Leijen, Daan: *Parsec.* http://legacy.cs.uu.nl/daan/parsec.html. Version: 2005. – [Online; accessed 01-February-2010]

[7] Matthes, Ralph: *Kurzeinführung in Haskell.* http://www.tcs.informatik.uni-muenchen.de/lehre/WS02-03/VLII/haskellintro/, 2002

[8] McCarthy, John: History of LISP. In: *SIGPLAN Not.* 13 (1978), Nr. 8, S. 217–223. http://dx.doi.org/http://doi.acm.org/10.1145/960118.808387. – DOI http://doi.acm.org/10.1145/960118.808387. – ISSN 0362–1340

[9] O'Donnell, John; Hall, Cordelia; Page, Rex: *Discrete Mathematics Using a Computer.* 2. Auflage. London: Springer Verlag, 2006

[10] O'Sullivan, Bryan; Goerzen, John; Stewart, Don: *Real World Haskell.* 1. Auflage. Sebastopol, CA, USA: O'Reilly Media Inc., 2008

[11] Paul Hudak, John P.; Fasel, Joseph: *A Gentle Introduction to Haskell.* http://www.haskell.org/tutorial/index.html, 2000

[12] Pepper, Prof. Dr. P.: *Funktionale Programmierung in Opal, ML, Haskell und Gofer*. 1. Auflage. Berlin: Springer, 1999

[13] Wikibooks: *Haskell*. `http://en.wikibooks.org/wiki/Haskell`, 2007