TECHNISCHE UNIVERSITÄT CAROLO-WILHELMINA ZU BRAUNSCHWEIG

Übungsskript

Eine Einführung in die funktionale Programmierung mit Haskell

Jan Oliver Ringert

Wintersemester 07/08

Institut für Programmierung und Reaktive Systeme
Dr. Werner Struckmann

Vorwort

Dieses Dokument bietet eine Einführung in die funktionale Programmierung am Beispiel von Haskell. Es orientiert sich hauptsächlich an [2]. Zusätzlich wurde Material aus einer Vorlesung über funktionale Programmierung in Scheme von Dr. W. Struckmann übernommen.

Geeignete Einführungen in Haskell 98 stellen auch [8] und [9] dar. Eine umfassende Referenz findet sich in dem überarbeiteten Haskell 98 Report [4], welcher zur Zeit Grundlage für die Sprache Haskell ist.

Inhaltsverzeichnis

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 [3]. Eine Biographie des Mathematikers Haskell Brooks Curry findet sich in [7].

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 [6] 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 [3] 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 [3] 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 [4]) 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 Datentypen beinhaltet, die z. B. zur Ein-/Ausgabe und zur Behandlung von Listen dienen. Außerdem sind arithmetische Operatoren enthalten.

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 und Funktionen

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.

2.2.1 Typen

Einige Typen, die dem Benutzer zur Verfügung stehen sind `Int, Float, Char, String` und `Bool`. Die Typen `Int` und `Float` gehören zu der Klasse `Num` für deren Datentypen viele Funktionen wie z. B. Addition und Subtraktion definiert sind.

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 Datentypen 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.

2.2.2 Funktionen

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 3.5.1).

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. Der Befehl `:?` listet die möglichen Kommandos auf.

Aufgabe 2.1.

1. Starten Sie die interaktive Shell und probieren Sie ein paar einfache Berechnungen wie in Beispiel 2.1 aus.

2. Laden Sie das Modul `Simple` (in Listing 2.2) in der Shell. Benutzen Sie die Funktionen `inc, sum', exclaim` und `averageOf2` für einige Beispiele.

```haskell
module Simple where

-- Erhöhen eines Integers um 1
inc :: Int -> Int   -- Signatur
inc x = x + 1       -- Implementierung/Berechnungsvorschrift

-- Addieren von zwei Integern
sum' :: Int -> Int -> Int
sum' x y = x + y

-- Berechnen des Durchschnitts von zwei Werten
averageOf2 :: (Fractional a) => a -> a -> a
averageOf2 a b = (a + b)/2

-- Einer Zeichenkette "!" anfügen
exclaim :: String -> String
exclaim a = a ++ "!"
```

Listing 2.2: Datei Simple.hs

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

Die Funktion `inc` 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:

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

Listing 2.3: Alternative zur Definition von `inc`

Aufgabe 2.2.

1. Fügen Sie zur Datei `Simple.hs` die Definition aus Listing 2.3 hinzu und vergleichen Sie die Signaturen von `inc` und `inc'`.

2. Erweitern Sie die Definitionen um die Funktion `double`, welche einen Wert verdoppelt, und eine Funktion `square`, welche einen Wert quadriert.

3. Sind die folgenden Konstrukte korrekt? Welche Typen besitzen sie? Prüfen Sie Ihre Vermutungen in der interaktiven Shell.

   a) `sum' 12`

b) `sum' (square 12)`

c) `sum' inc 12`

2.3 Kontrollstrukturen

Haskell Programme werden aus Modulen zusammengesetzt. 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 2.4: Beispiel eines Kommentars, der ein Modul einleitet

2.3.1 Auswahl

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 Seitenbedingungen können auch wie im zweiten Beispiel in Listing 2.5 realisiert werden.

```
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 2.5: Möglichkeiten zur Auswahl von Werten

2.4 Variablen und Tupel

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 2.6: Globale und lokale Variablen

In Haskell können auch Tupel von Variablen benutzt werden. Dabei kann es sich um zusammengesetzte Tupel aus unterschiedlichen Typen handeln (siehe Listing 2.8). Wenn ein spezielles Tupel immer wieder verwendet wird, kann durch `type` ein Name für das Tupel vergeben werden.

```
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 2.7: Tupel und Typen

Aufgabe 2.3.

1. Programmieren Sie eine Funktion `mittelPunkt`, die zwei `Punkte` als Argumente annimmt und einen dritten ausgibt, der den geringsten möglichen Abstand zu beiden besitzt.

2. Geben Sie die Definition einer Funktion an, die für einen `Punkt` entscheidet, in welchem Quadranten er sich befindet (per Fallunterscheidung, `if-then-else` oder einer geeigneten Kombination).

2.5 Listen

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

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

Hilfreiche Funktionen zum Erstellen von Listen sind der Konstruktor `[]` (genannt `nil`) und die Funktion `(:) :: a -> [a] -> [a]` zum Anhängen 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 2.8: Einfaches Modell einer Bank

Aufgabe 2.4.

1. Schreiben Sie eine Funktion, die eine `Bank` annimmt und eine Liste der Kunden ausgibt.

2. Finden Sie durch Ausprobieren und Betrachten der Signatur heraus, welche Auswirkungen die folgenden Funktionen auf Listen haben: `length`, `head`, `tail`, `sum`, `product`.

3 Rekursion und Listen

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 2.8). 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

Aufgabe 3.1.

1. Geben Sie eine Funktion an, mit der die Fakultät einer ganzen positiven Zahl n berechnet werden kann (den Typen `Integer` anstelle von `Int` verwenden).

2. Geben Sie eine Funktion an, die für die beiden Argumente `c::Char` und `n::Int` eine Zeichenkette der Länge `n` ausgibt, welche nur aus Wiederholungen des Buchstaben `c` besteht.

3. Erweitern Sie die partielle Funktion `natSum` zu einer totalen Funktion über dem Datentypen `Int`. Geben Sie in vorher undefinierten Fällen eine Fehlermeldung durch die Funktion `error` aus.

4. Setzen Sie die iterative Variante des euklidischen Algorithmus zur Bestimmung des größten gemeinsamen Teilers (siehe Listing 3.2) in eine funktionale Implementierung in Haskell um.

```
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 Datentypen, 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

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

Listing 3.4: Mustervergleich unter Verwendung von Konstruktoren

Aufgabe 3.2.

1. Geben Sie eine Funktion `listProduct :: Num a => [a] -> a` an, die alle Elemente der Liste multipliziert und das Ergebnis zurück liefert.
2. Modifizieren Sie die Funktion so, dass für eine leere Eingabe `0` das Ergebnis ist.
3. Erstellen Sie eine Funktion, die eine Liste von Listen des gleichen Typs zu einer Liste zusammenfasst.

3.3 Typklassen

Typklassen legen in Haskell Funktionen fest, die jeder Datentyp, welcher eine Instanz der Klasse ist, anbieten muss. 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`*):

Beispiel 3.2.

```haskell
Prelude> :i Num
class (Eq a, Show a) => Num a where
  (+) :: a -> a -> a
  (*) :: a -> a -> a
  (-) :: a -> a -> a
  negate :: a -> a
  abs :: a -> a
  signum :: a -> a
  fromInteger :: Integer -> a
        -- Defined in GHC.Num
instance Num Double -- Defined in GHC.Float
instance Num Float -- Defined in GHC.Float
instance Num Int -- Defined in GHC.Num
instance Num Integer -- Defined in GHC.Num
```

Listing 3.5: Informationen über Typklassen in der interaktiven Shell

Es ist möglich festzulegen, welcher Typklasse eine Typenvariable (siehe Abschnitt 3.4.1) in der Signatur einer Funktion angehört. Diese Festlegung müssen vor der Verwendung der Variablen getroffen werden:

Beispiel 3.3.

```
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) then
5                                 True else
6                                 contains y xs
```

Listing 3.6: Verwendung der Typklasse `Eq`

In der obigen Definition müssen alle in den Argumenten zugelassenen Typen die beiden Funktionen `(==) :: a -> a -> Bool` und `(/=) :: a -> a -> Bool` unterstützen, mit denen Elemente auf Gleichheit bzw. Ungleichheit geprüft werden können.

Aufgabe 3.3. Implementieren Sie die Funktion `isSorted`, die für eine Liste feststellt, ob ihre Elemente aufsteigend sortiert sind. Verwenden Sie hierzu die Typklasse `Ord`.

3.4 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.

3.4.1 Parametrische Polymorphie in Haskell

Die verwendeten Funktionen `head, tail, fst, ...` sind nicht auf einen Datentypen festgelegt, sondern enthalten in der Signatur Typenvariablen (beginnend mit Kleinbuchstaben). Die konkrete Instanz (der Datentyp) wird erst bei der Anwendung der Funktion festgelegt (Instantiierung der Typenvariablen).

Beispiel 3.4.

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

Listing 3.7: Infix-Funktion `(++)` zum Konkatenieren von Listen beliebigen Typs

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

3.4.2 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 verwendet. So ermöglichen z. B. alle Typen der Typklasse `Eq` 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.

3.5 Beispiel zum Arbeiten mit Listen

3.5.1 let-in-Ausdruck

Es existiert in Haskell 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 folgendermaßen 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`, 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 = let s = (sum xs)
3                  w = (length xs)
4                       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 [5]).

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

3.5.2 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

Aufgabe 3.4.

1. Geben Sie eine Funktion, die bei einer gegebenen Preisliste den Preis für ein Produkts liefert.
2. Implementieren Sie eine Funktion, die aus einer Einkaufsliste und einer Preisliste eine Liste mit Tupeln (Preis pro Stück, Menge) erstellt.
3. Entwickeln Sie eine Funktion, die aus einer wie im vorherigen Schritt erzeugten Liste eine Endsumme berechnet.
4. Fügen Sie alle entwickelten Funktionen in der Definition der Funktion `kosten :: PreisListe -> EinkaufsListe -> Cent` zusammen, sodass die Endsumme eines Einkaufs berechnet wird.

3.5.3 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.

Aufgabe 3.5.

1. Implementieren Sie die Funktion `insertSorted`.
2. Geben sie eine Funktion `iSort` an, die eine Liste nach dem obigen Verfahren unter Benutzung von `insertSorted` sortiert.

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

Listing 3.15: Verwendung der Funktion `map`

Aufgabe 3.6.

1. Implementieren Sie eine weitere Funktion höherer Ordnung mit dem Namen `filter'` und der Signatur `filter' :: (a -> Bool) -> [a] -> [a]`. In der Ausgabeliste sollen nur noch die Elemente vorkommen, für die das übergebene Prädikat `True` ist.

2. Geben Sie eine Implementierung für die Funktion `dropWhile'` an, die vom Anfang einer Liste alle Elemente entfernen, solange das übergebene Prädikat zutrifft. Oder implementieren Sie die Funktion `takeWhile'`, welche das Anfangsstück einer Liste liefert, für dessen Elemente das Prädikat zutrifft. Die Signatur ist wie oben jeweils `:: (a -> Bool) -> [a] -> [a]`.

3.7 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`.

Aufgabe 3.7. Geben Sie die Funktionen `doubleList` und `squareList` an, welche alle Elemente einer Liste verdoppeln bzw. quadrieren. Benutzen Sie für die tatsächlichen Rechenoperationen anonyme Funktionen und für den Rest `map`.

3.8 Listenkomprehension

Die Listenkomprehension ist eine Methode, um Listen aus anderen Listen zu erzeugen. Der 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 haben die Form x `<-` e wobei x ein Muster des Typen `T` ist und der Ausdruck e den Typen `[T]` besitzt.
- Prädikate sind 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.

Aufgabe 3.8. Geben Sie mit Hilfe der Listenkomprehension eine Funktion an, die aus zwei Listen mit Zahlen eine Liste mit Paaren erstellt. Es sollen nur solche Paare vorkommen, bei denen die Zahl aus der ersten Liste durch die aus der zweiten teilbar ist.

Es kann die Funktion `mod :: Integral a => a -> a -> a` verwendet werden, welche den Rest bei einer Division liefert.

4 Ein-/Ausgabe, Datentypen und Module

4.1 Interaktion mit dem Benutzer

Die bisher verwendeten Funktionen bekamen ihre Parameter direkt in der interaktiven Shell übergeben. Der Benutzer eines Programms sollte jedoch nicht dazu gezwungen werden, die interaktive Shell benutzen zu müssen und seine Eingaben durch die Definition von Variablen vorzunehmen. Da Haskell eine reine funktionale Programmiersprache ist, sind Ein- und Ausgaben als Interaktion mit dem Benutzer gar nicht ohne weiteres möglich.

Das I/O-System von Haskell ist wie der Rest der Sprache rein funktional. Es wird eine Monade benutzt, um I/O-Funktionen in Haskell zu integrieren. Die I/O Monade „vermittelt“ zwischen Werten (wie bisher bekannt) und Aktionen, welche die eigentlichen I/O-Operationen sind.

4.1.1 I/O-Funktionen

In Haskell besitzen alle Ein- und Ausgabefunktionen den Typen `IO a` wobei `a` eine Typvariable ist. Diese I/O-Funktionen können Daten einlesen oder ausgeben. Der Typ für den die Variable `a` steht, ist der Typ des Wertes, der von einer Funktion zurück gegeben wird. Zur Ausgabe von Text stellt `Prelude` die folgenden Funktionen bereit.

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

Listing 4.1: 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 4.2: Einige Eingabefunktionen des Prelude Moduls

Die meisten der obigen Funktionen sind von ihrem Namen und ihrer Signatur her verständlich. Kurze Erklärungen finden sich in [4]. 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`: Ähnlich der Funktion `read :: (Read a) => String -> a` (siehe Abschnitt 4.1.3), wird eine Zeichenkette gelesen und versucht, das Gelesene zu parsen. Wenn z. B. eine Zahl aus einer Zeichenkette gelesen werden soll, schreibt man `readIO „123“ :: IO Integer`. Da `Integer` eine Instanz der Klasse `Read` ist, kann diese Funktion verwendet werden. Der Unterschied zwischen `read` und `readIO` ist, dass `readIO` einen Parserfehler an die IO Monade signalisiert und nicht das Programm beendet. Um an den geparsten Wert vom Typ `Integer` zu kommen, siehe Abschnitt 4.2.1.

4.1.2 Datei-Ein-/Ausgabe

Manchmal ist es sinnvoll, die Möglichkeit zu haben, in Dateien zu schreiben und auch wieder daraus zu lesen. In Haskell wird dies über die folgen Funktionen für das Lesen von Zeichenketten gelöst.

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

Listing 4.3: Funktionen zum Arbeiten mit Dateien

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 Datentypen in Dateien geschrieben werden sollen, müssen diese vorher umgewandelt werden. Zum Umwandeln kann die in Abschnitt 4.1.3 vorgestellte Funktion `show` verwendet werden.

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 4.1.

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 4.4: Lesen aus einer Datei

4.1.3 Werte darstellen und Lesen

Das Schreiben von Daten in Dateien ist besonders dann sinnvoll, wenn beliebige darstellbare Werte aus einem Programm auch in Dateien gespeichert 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 werden. Die Umwandlung geschieht in der Shell durch die Funktion `show` des jeweiligen Datentypen, welche in der Klasse `Show` definiert ist. Fast alle Typen in `Prelude` sind Instanzen der Klasse `Show`. Ausnahmen sind nur die Funktionstypen und alle `IO`-Typen. `Prelude` bietet eine Unterstützung für eigene Datentypen, sodass es sehr leicht ist, diese zu Instanzen der Klasse `Show` zu machen und sie z. B. ausgeben zu lassen.

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 Klasse `Read`, von der ebenfalls alle Datentypen aus `Prelude` mit der obigen Einschränkung Instanzen sind.

4.2 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 4.2.

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

Listing 4.5: 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.

4.2.1 Ablaufsteuerung und Auswertungsreihenfolge

Die Auswertungsreihenfolge normaler Funktionen in Haskell ist lediglich durch Datenabhängigkeiten bestimmt. So kann z. B. eine Zuweisung erst dann ausgeführt werden, wenn die rechte Seite ausgewertet worden ist. Für I/O-Operationen gilt diese Freiheit jedoch nicht, da die Reihenfolge der Aus- und Eingabe von Daten sehr wichtig sein kann. Wenn Aktionen miteinander verbunden werden sollen, kann die `do`-Notation verwendet werden.

Beispiel 4.3.

```
1 do
2    putStr "Das␣Ergebnis␣von␣1␣+␣1␣ist␣"
3    print (1 + 1)
```

Listing 4.6: Sequentielle Abarbeitung der Ausgabe mit Hilfe der `do`-Notation

Die Anweisungen, welche dem `do` eingerückt folgen, können einfache oder zusammengesetzte I/O-Operationen sein. Um das Ergebnis einer Aktion zu erhalten, verwendet man die Notation `v <- aktion`. Die Variable `v` ist vom Typ `typ`, wenn der Ausdruck `aktion` den Typen `IO typ` besitzt. Es besteht ein wichtiger Unterschied zwischen dem Typ der IO-Monade z. B. `IO String` und dem einfachen Typen `String`. So kann z. B. die Rückgabe der Funktion `getLine :: IO String` nicht direkt an die Funktion `putStr :: String -> IO ()` übergeben werden, da die Typen sich unterscheiden. Es ist durch `( <- )` eine zwischenzeitliche Bindung an eine Variable notwendig.

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

Listing 4.7: Ausgeben einer eingelesenen Zeile

Der Typ und Wert der Rückgabe einer Funktion, die über einen `do`-Ausdruck definiert wird, entsprechen dem der letzten ausgeführten Anweisung, solange der Wert nicht explizit vorher zurückgegeben wird. Es ist möglich, den Rückgabewert über die Funktion `return :: (Monad m) => a -> m a` zu bestimmen.

Beispiel 4.4.

```
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 4.8: Definition der Funktion `getLine`

Aufgabe 4.1. Implementieren Sie die Funktion `ask :: String -> IO String`, welche die im ersten Parameter übergebene Frage ausgibt und eine eingelesene Zeile als Antwort liefert.

4.2.2 Weiteres zur Ein-/Ausgabe

Bei der Verwendung des `do`-Ausdrucks muss darauf geachtet werden, dass die folgenden Zeilen alle eingerückt sind. Wenn eine bedingte Anweisungen verwendet wird, welche im `then` oder `else` Teil mehrere Ein-/Ausgabeaktionen nutzt, muss der jeweilige Block mit einem neuen `do`-Ausdruck eingeleitet werden. Rekursion lässt sich weiterhin wie bekannt verwenden, es muss bei der Auswertung von Werten jedoch darauf geachtet werden, dass die Typen (`IO ...` oder nur `...`) übereinstimmen.

Beispiel 4.5.

```
1 upperLine :: IO ()
2 upperLine = do
3         line <- getLine
```

```
4         if line == ""
5           then do       -- Basisfall
6             putStrLn "upperLine beendet"
7             return ()
8           else do       -- Rekursionsschritt
9             putStrLn (allUpper line)
10            upperLine
```

Listing 4.9: Weiteres Anwenden des `do`- Ausdrucks in Blöcken

Es ist wichtig zwischen der Ausgabe und der Rückgabe von Funktionen zu unterscheiden. Im obigen Beispiel ist die Rückgabe der Funktion ein leeres Tupel (Unit), wobei die Ausgabe aus mindestens einer Text-Zeile besteht.

Viele Datentypen können direkt über die Funktion `readLn :: (Read a) => IO a` eingelesen werden. Es wird in der Signatur gefordert, dass die einlesbaren Typen Instanzen der Klasse `Read` sind. So können z. B. Werte vom Typ `Int`, `String` oder `Float` mit der selben Funktion eingelesen werden. Über Typinferenz stellt der Compiler beim Übersetzen fest, welche konkrete Funktion verwendet werden muss.

Aufgabe 4.2.

1. Geben Sie eine Funktion, die eine Liste aus den Paaren `(String, Int)` in eine Datei speichert, nach deren Pfad per `ask` gefragt wird.

2. Schreiben Sie eine Funktion, mit der die gespeicherten Daten wieder gelesen werden können.

4.3 Benutzerdefinierte Datentypen

Die Bibliothek `Prelude` versorgt den Programmierer mit einigen grundlegenden Datentypen und Klassen. Wenn die Anwendungen komplexer werden, möchte man jedoch durch Abkürzungen oder eigene zusammengesetzte Datentypen 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.

4.3.1 Parametrische Typsynonyme

Zwei Datentypen, 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 4.10: 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 Datentypen 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 4.11: 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.

4.3.2 Algebraische Sicht der Typdefinition

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

Algebraisch gesehen hat die Datentypdeklaration die Form:

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

Dabei ist cx ein Kontext wie z. B. `Eq a`. T ist ein „type constructor“, der wie in Abschnitt 4.3.1 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.

4.3.3 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 4.6. Es soll ein einfacher Datentyp `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 4.12: Definition des Aufzählungstypen `Wochentag`

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

Beispiel 4.7. 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 4.13: 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 4.8.

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

Listing 4.14: 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.

4.3.4 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 4.9.

```
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 4.15: Definition eines Datentypen für geometrische Figuren

Aufgabe 4.3. Geben Sie eine Funktion, die für eine konkrete `Figur` den Mittelpunkt dieser berechnet.

4.3.5 Rekursive Datentypen

Wie schon bekannt von der Definition von Funktionen kann man den zu definierenden Datentypen auch auf der rechten Seite selbst verwenden, da als Parameter der „data constructors“ Datentypen 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 4.10.

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

Listing 4.16: Definition eines Datentypen 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 4.17: Berechnung der Mächtigkeit von Mengen verschiedenen Typs

Aufgabe 4.4. Geben Sie eine Funktion, die für ein Element vom Typ `T` feststellt, ob es in einer Menge vom Typ `Set T` enthalten ist.

4.3.6 Typklassen für eigene Datentypen

Damit das Arbeiten mit neu definierten Datentypen 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 Datentyp kann mit dem Schlüsselwort `deriving` diese Klassen ableiten.

Beispiel 4.11.

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

Listing 4.18: Ableiten der Klassen `Eq` und `Ord`

Die Ordnung der einzelnen „data constructor“ richtet sich nach der Reihenfolge, in der sie in der Datentypdefinition auftauchen. So gilt hier `Montag < Donnerstag`. Die Klassen `Ord` und `Eq` können für alle benutzerdefinierten Datentypen abgeleitet werden, welche aus Datentypen 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 Datentyp zum wieder Einlesen die Funktion `read` benötigt. Wie im vorherigen Listing erweitert man zum Ableiten der Klasse einfach die Liste nach dem Schlüsselwort `deriving`.

Beispiel 4.12.

```
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 4.19: Ableiten der Klassen `Show` und `Read`

Es müssen wieder alle Typen die Klasse 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 Klasse `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 Klasse `Enum` abgeleitet wird, sind Konstrukte wie `[Montag..Samstag]` möglich. Dieses Konstrukt erzeugt eine Liste mit allen Tagen von `Montag` bis `Samstag` als Elemente.

Aufgabe 4.5.

1. Definieren Sie einen parametrischen Datentypen `Vielleicht a`, der entweder `Nichts` ist oder ein Wert vom Typ `a` enthält.
2. Schreiben Sie ein Prädikat zur Prüfung, ob `Vielleicht a` vielleicht `Nichts` ist.
3. Geben Sie eine Funktion, mit der der Wert von `Vielleicht a` bestimmt werden kann. Falls es `Nichts` ist, benutzen Sie `error`.

Aufgabe 4.6.

```
1 -- Eine Assoziationsliste mit Schluessel und Wert
2 type AssocList key value = [(key, value)]
3
4 lookUp :: Eq a => AssocList a b -> a -> b
5 lookUp [] key = error "Schluessel nicht gefunden."
6 lookUp ((k, v):as) key = if (k == key)
7                          then v
8                          else lookUp as key
```

Listing 4.20: Finden von Elementen in Assoziationslisten

Ändern Sie die gegebene Definiton von `lookUp` so ab, dass auch `Nichts` zurückgegeben werden kann.

4.4 Module

Bei größeren Softwareprojekten kann es schnell zu einer Menge von Code, Funktionen und Datentypen kommen. Fast überall ist eine Trennung in Module oder Klassen 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

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 [4] zu finden.

4.4.1 Export

Jedes Modul kann seine Funktionen und Datenstrukturen anderen zur Verfügung stellen. Wenn der Entwickler keine Angaben macht, dann werden alle Funktionen, Datenstrukturen und Klassen (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 4.13.

```
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 4.21: 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 4.14.

```
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 4.22: Nur Teile der „data constructor“ exportieren

In diesem Beispiel wird gezeigt, dass es unterschiedliche Möglichkeiten gibt, Datentypen 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 Datentypen `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.

4.4.2 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 4.15.

```
1 module A
2 where
3 import qualified Set
4 import Supermarkt (Cent, Produkt)
5 import Prelude hiding (print, error)
6
7 ...
```

Listing 4.23: 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.

4.4.3 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 Klasse 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.

5 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.

5.1 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.

Aufgabe 5.1.

1. Definieren Sie einen parametrischen, rekursiven Datentypen für einen binären Baum, der Werte verschiedener Datentypen in den Knoten halten kann.
2. Erstellen Sie per Hand sortierte Bäume, die die Werte „AUTO“, „BOOT“, „HAUS“, „KAMEL“ und „TIER“ enthalten. Ein Baum soll dabei die minimale Tiefe und einer die maximale Tiefe besitzen.

3. Geben Sie ein Prädikat an, ob ein gegebenes Element in dem Baum vorhanden ist (Klasse `Ord` und `Eq` benötigt).

4. Entwickeln Sie eine Funktion, die Elemente sortiert in einen Baum einfügt.

5.2 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 5.1: 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 [8].

5.3 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.

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

Listing 5.2: 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.

5.3.1 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.

Aufgabe 5.2.

1. Geben Sie eine Funktion an, die überprüft, ob ein Wort in einem gegebenen `Trie` vorhanden ist.
2. Entwickeln Sie eine Funktion, die Zeichenketten in einen `Trie` einfügt.

5.3.2 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.

Aufgabe 5.3. Geben Sie eine Funktion zur Bestimmung der Menge der gespeicherten Wörter in einem `Trie` an.

Aufgabe 5.4. Schreiben Sie eine Funktion zum Löschen von Einträgen aus einem `Trie`.

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

[3] 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

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

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

[6] 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

[7] O'Connor, J. J.; Robertson, E. F.: Haskell Brooks Curry. In: *MacTutor History of Mathematics* (2004). http://www-history.mcs.st-andrews.ac.uk/Biographies/Curry.html

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

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