Überblick zur Programmiersprache

Haskell 98

LV Funktionale Programmierung

September 2000

Martin A. Godisch
Dr.-Ing. Monika Sturm

Fakultät Informatik
Institut für Theoretische Informatik

Inhaltsverzeichnis

Kapitel 5

Überblick zur Programmiersprache Haskell

Der folgende Überblick soll Studenten unterstützen, sich lehrveranstaltungsbegleitend praktische Fähigkeiten im Umgang mit der Sprache Haskell zu erarbeiten. In der Vorlesung werden zur weiteren Vertiefung entsprechende Literaturempfehlungen gegeben.

5.1 Was ist Haskell?

Haskell ist eine nicht-strikte, reine funktionale Programmiersprache, die nach dem Logiker Haskell Brooks Curry benannt ist. Sie wurde von einem internationalen Kommitee entwickelt, dem neben anderen Simon Peyton Jones, John Hughes, Philip Wadler und Paul Hudak angehören. Die aktuelle Version ist derzeit Haskell 98, welches als eine stabile Variante der zuletzt erschienenen Versionen zu verstehen ist.

Beim Design von Haskell spielte das Bedürfnis nach einer allgemein anerkannten funktionalen Sprache eine entscheidende Rolle. Diese sollte zu Lehrzwecken ebenso geeignet sein, wie zur Forschung und Anwendung bei der Programmierung. Sie sollte die bis dahin bekannten Ideen der Forschung bezüglich funktionaler Sprachen in sich vereinen und als Grundlage für kommende Forschungsarbeit in Sprachdesign und Implementierung dienen.

Die Definition der Sprache Haskell ist niedergeschrieben im *Report on the Programming Language Haskell 98* [1] sowie in den *Standard Libraries for the Haskell 98 Programming Language* [2].

5.2 Ausführung von Haskell-Programmen

Die Ausführung von Haskell-Programmen soll hier zunächst mithilfe des Haskell-Interpreters *Hugs* erläutert werden. Hugs ist eine auf Haskell basierende funktionale Programmierumgebung. Im Fakultätsrechenzentrum ist Hugs sowohl unter

Solaris im Raum 108 wie auch unter Linux im Raum 136 installiert. Zum Start öffne man ein Terminal und gebe das Kommando `hugs` ein. Bei der Benutzung anderer Rechner kann z. B. eine *remote shell* auf dem Server 6 geöffnet werden. Für den Heimgebrauch ist Hugs unter `http://www.haskell.org/hugs/` frei erhältlich.

Einmal gestartet, verhält sich Hugs wie ein Vielzweck-Taschenrechner. Man übergibt dem System ein Kommando und es antwortet entsprechend. Programmieren in Haskell bedeutet nun, Definitionen in Skripten niederzuschreiben. Es gibt vordefinierte *(built-in)* Definitionen, und es gibt solche, die man selbst und andere Programmierer definiert haben. Um die in einem Skript enthaltenen Formeln für Berechnungen nutzen zu können, muß dieses geladen werden. Dies geschieht mit dem Kommando `:load script`. Die entsprechenden Dateien haben üblicherweise die Endung `.hs` *(Haskell Script)* oder `.lhs` *(Literate Haskell Script)*. Alle am Hugs-Prompt verfügbaren Befehle können – solange eindeutig – abgekürzt werden. Demzufolge kann z. B. `:load` auch als `:l` getippt werden.

Um ein bequemes Ändern geladener Skripten zu ermöglichen, läßt sich das zuletzt geladene Skript mit dem Kommando `:edit` editieren. Daraufhin setzt sich Hugs in den Hintergrund und übergibt die Kontrolle an den durch die Umgebungsvariable EDITOR spezifizierten Editor. Andere Dateien können auf gleiche Weise editiert werden, wenn ihr Name dem `:edit` Befehl folgt.

Es ist natürlich möglich, mehr als ein Skript zu laden. Da `:load` ersetzend arbeitet, gibt es das Kommando `:also`, welches die Definitionen eines weiteren Skripts zusätzlich zu den bereits vorhandenen lädt.

Um Hugs zu beenden, verwende man den Befehl `:quit`.

Eine Übersicht zu allen am Prompt verfügbaren Kommandos erhält man bei Eingabe von `:?`. Weitere Informationen zu Hugs sind in der Online-Dokumentation des Interpreters zu finden.

5.3 Funktionsdefinitionen und Syntax

5.3.1 Skripten & Kommentare

An dieser Stelle soll zunächst noch einmal auf den Unterschied zwischen *Haskell Scripts* und *Literate Haskell Scripts* eingegangen werden.

Haskell Scripts sind Textdateien, in denen jede Zeile Haskell-Code ist – mit Ausnahme von Kommentaren. Ein Kommentar wird mit einem doppelten Minus „`--`" eingeleitet und reicht bis zum Ende der Zeile. Ein geschachtelter Kommentar beginnt mit der Zeichenfolge „`{-`" und erstreckt sich bis zu dem zugehörigen „`-}`". Geschachtelte Kommentare können mehrere Zeilen umfassen und – wie der Name schon sagt – beliebig geschachtelt werden. *Haskell Scripts* werden als `.hs` Dateien abgespeichert.

Literate Haskell Scripts dagegen sind Textdateien, in denen jede Zeile Kommentar ist – mit Ausnahme von Code-Zeilen, die durch das Zeichen „`>`" eingeleitet werden und eine Leerzeile Abstand zum umgebenden Text haben müssen. *Literate Haskell Scripts* eignen sich insbesondere für gut dokumentierten Quelltext und werden als `.lhs` Dateien gespeichert.

5.3.2 Der Datentyp Int

Im folgenden soll zunächst ein einfacher Datentyp und die Syntax von Haskell eingeführt werden. Zu diesem Zweck wird ein simples Programm definiert, welches zwei *Integers* addiert und deren Summe liefert.

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

Die erste Zeile bestimmt den Typ der Funktion f. Der doppelte Doppelpunkt „`::`" wird üblicherweise als „*has type*" gelesen. Somit besagt die Definition, daß f zwei Argumente vom Typ *Int* fordert und ein Ergebnis vom Typ *Int* liefert. Die nächste Zeile definiert das Verhalten der Funktion. Wird f mit zwei Argumenten x und y aufgerufen, so ergibt sich der Funktionswert aus $x + y$.

Nachdem die Definition in einem Skript niedergeschrieben wurde, kann dieses in Hugs geladen werden. Falls alles syntaktisch korrekt ist, wird Hugs sich nicht beschweren und den Prompt *Prelude* durch *Main* ersetzen. Nun kann eine Anfrage gestellt werden, woraufhin das Ergebnis ausgegeben wird.

```
Prelude> :l script.hs
Reading file "script.hs":

Hugs session for:
/usr/share/hugs/lib/Prelude.hs
script.hs
Main> f 1 2
3
```

Ein anderer Weg, die Funktion f aufzurufen, wäre die Infix-Notation. Ein Ausdruck der Art $1f2$ ist möglich, jedoch muß der Funktionsname in diesem Fall in sogenannte *back quotes* „`‘`" eingeschlossen werden.

```
Main> 2 ‘f‘ 3
5
```

Das Gegenteil, einen Infix-Operatoren als Präfix zu schreiben, geschieht, indem dieser in runde Klammern gesetzt wird.

```
Main> (+) 4 5
9
```

Mit diesem Wissen ist es nun möglich, die Definition der Funktion f auf komfortablere Weise niederzuschreiben.

```
f :: Int -> Int -> Int
f = (+)
```

Welche weiteren Operatoren lassen sich auf Variablen des Typs *Int* anwenden? Sie sind im folgenden aufgeführt.

```
2 + 3         -- Summe von 2 und 3
2 * 3         -- Produkt von 2 und 3
2 ^ 3         -- 3. Potenz von 2
2 - 3         -- Differenz aus 2 und 3
-2            -- Negation von 2
2 `div` 3     -- ganzzahlige Division
2 `mod` 3     -- Divisionsrest
abs (-2)      -- Betrag von -2
negate (-2)   -- Negation von -2
```

5.3.3 *Guards* und *pattern matching*

Es sollen nun zwei weitere Konstrukte eingeführt werden, die das funktionale Äquivalent zu imperativen *if*-Konstrukten darstellen. Sie werden als *guards* (Wächter) und *pattern matching* (Mustererkennung) bezeichnet.

```
h :: Int -> Int
h n | n == 0    = 0
    | otherwise = n-1
```

Mittels *guards* lassen sich – in Abhängigkeit von einer Bedingung – unterschiedliche Berechnungen durchführen. *Guards* werden von oben nach unten getestet und der dem ersten erfolgreichen *guard* folgende Ausdruck wird ausgewertet. Es besteht keine Notwendigkeit, die Bedingungen disjunkt zu formulieren. Das (optionale) Schlüsselwort *otherwise* fängt alle verbleibenden Fälle und ist deswegen der letzte Eintrag der Liste.

Das folgende Beispiel zeigt eine einfache Form des *pattern matching*. *h* hat hier dieselbe Funktionalität wie zuvor. Auch die Muster des *pattern matching* werden von oben nach unten geprüft und der erste zutreffende Ausdruck ausgewertet. *Pattern matching* wird von größerer Bedeutung sein, wenn es um das Programmieren mit Listen geht, und deshalb in Kapitel 5.7 häufig benutzt werden.

```
h :: Int -> Int
h 0     = 0
h (n+1) = n
```

Ein spezielles Muster beim *pattern matching* ist das Zeichen „_“, welches immer paßt, den gefundenen Ausdruck aber an keine Variable bindet. Von dieser Möglichkeit wird immer dann Gebrauch gemacht, wenn ein Argument berücksichtigt werden muß, welches anschließend auf der rechten Seite der Definitionsgleichung nicht gebraucht wird.

5.3.4 Berechnungsfolge

Die Assoziativität eines Operators erlaubt es, Ausdrücke auszuwerten, in denen derselbe Operator mehrmals vorkommt. Im Fall des Erscheinens verschiedener

Operatoren ist die Bindungsstärke der einzelnen Operatoren ausschlaggebend, d. h. diese wird verglichen und der stärker bindende Operator zuerst berechnet. Eine vollständige Auflistung der Assoziativitäten und Bindungsstärken ist im *Haskell Report* nachlesbar.

Es soll hier trotzdem gesagt werden, daß die Funktionsanwendung die überhaupt stärkste Bindung ist. Dies bedeutet, daß $fn + 1$ z. B. $(fn) + 1$ ist und nicht $f(n+1)$. Auch für Verwirrung mag das „`-`“ sorgen, welches sowohl als Infix- als auch als Präfix-Operator benutzt werden kann. $f - 1$ subtrahiert deshalb 1 vom Ergebnis der Funktion f und ist nicht $f(-1)$, was möglicherweise gewünscht war. Aus diesen Gründen sollte im Zweifelsfall immer geklammert werden!

5.3.5 *Offside rule, where* und *let*

Haskell legt Wert auf das Erscheinungsbild des Codes. Um das Ende einer Definition zu finden, sucht das System nach dem ersten Stück Text, welches gleichermaßen eingerückt ist oder aber links vom aktuellen Zeilenanfang beginnt. Dieses Verhalten ist als *offside rule* bekannt. Weiterhin gilt, daß ein Semikolon „`;`“ zwischen mehreren Definitionen in einer Zeile deren Enden bestimmt.

Jede Gleichung kann von einer Menge von Definitionen gefolgt sein, welche lokal zu ihrer Definition erfolgen. Diese Definitionen werden zu diesem Zweck nach dem Schlüsselwort *where* notiert. Im linken Beispiel ist die Funktion g außerhalb von f nicht mehr zu erreichen.

```
f :: Int -> Int
f x = g (x+1) where
  g = (^2)
```

```
f :: Int -> Int
f x = let
  g = (^2) in g (x+1)
```

Eine weitere Möglichkeit einer lokalen Definition ist die Verwendung des Schlüsselwortes *let*. So definiert das rechte Beispiel die Funktion g innerhalb des Gültigkeitsbereiches von f. Die Funktionalität der beiden Definitionen ist exakt die gleiche.

5.4 Typen, Polymorphismus und Typklassen

5.4.1 Der Datentyp Bool

Hier soll zunächst ein weiterer Datentyp namens *Bool* eingeführt werden. Der Wert einer Variablen des Typs *Bool* kann entweder *True* oder *False* sein. Das folgende Beispiel beantwortet die Frage, ob ein *Integer* größer oder gleich sieben ist.

```
f :: Int -> Bool
f = (>= 7)
```

Welche Operatoren können auf Variablen des Typs *Bool* angewendet werden? Sie sind im folgenden aufgelistet.

```
True && False  -- logisches UND
True || False  -- logisches ODER (inklusive)
not True       -- logische Negation
True == False  -- Gleichheit
True /= False  -- Ungleichheit
```

5.4.2 Der Datentyp Char

Ein weiterer Datentyp ist der Typ *Char. Char* steht für *character* und eine Variable dieses Typs faßt ein einzelnes Zeichen. Zeichen werden in sogenannte *single quotes* „'“ eingeschlossen.

```
c :: Char
c = 'a'
```

Aufgrund des engen Verhältnisses zwischen Zahlen und Zeichen gemäß der ASCII-Tafel gibt es zwei Funktionen, die einen der beiden Typen in den jeweils anderen konvertieren.

```
Main> ord 'a'
97
Main> chr 97
'a'
```

chr und *ord* sind vordefinierte Bibliotheksfunktionen. Sie gehören zum Haskell-Sprachumfang, sind jedoch nicht in der *Standard Prelude* enthalten. Um sie benutzen zu können, ist die *Library Char* einzubinden.

```
import Char (chr, ord)
```

5.4.3 Der Datentyp String

Ein weiterer Verwandter[1] des Typs *Char* ist der Datentyp *String.* Eine Variable des Typs *String* enthält eine – möglicherweise leere – Folge von Zeichen. Diese Folge wird in *double quotes* „"“ eingeschlossen. Einelementige Zeichenketten sind demzufolge nicht mit einzelnen Zeichen zu verwechseln: *'a'* ist vom Typ *Char, "a"* jedoch vom Typ *String.*

```
s :: String
s = "a string example"
```

5.4.4 Die Datentypen Float und Double

Variablen der Typen *Float* und *Double* enthalten Gleitkommazahlen. *Double* ist der Typ höherer Genauigkeit und wird deshalb vorrangig verwendet. Auch über diesen beiden Typen sind verschiedene Funktionen definiert, meist mathematischer Natur.

[1]Dies wird in Kapitel 5.7 erklärt, wie auch die über dem Typen String definierten Operationen.

5.4.5 Tupel

Tupel sind geordnete Paare von – möglicherweise verschiedenen – Datentypen. Im folgenden Beispiel ist dies eine Variable, die sowohl den Namen als auch das Alter einer Person beinhaltet.

```
person :: (String, Int)
person = ("Bill", 24)
```

Haskell stellt einige Funktionen zur Verfügung, die über Tupeln operieren.

5.4.6 Typ-Synonyme

Das Schlüsselwort *type* führt ein Typ-Synonym ein. Hierbei handelt es sich lediglich um einen neuen Namen für einen bereits vorhandenen Typen. Wie alle (nicht-polymorphen) Typen beginnt dieser mit einem Großbuchstaben.

```
type Person = (String, Int)
```

5.4.7 Polymorphismus

Polymorphismus bedeutet „hat viele Gesichter“. Eine polymorphe Funktion akzeptiert Variablen verschiedener Typen als Eingabe. Man stelle sich z. B. eine Projektionsfunktion vor, die von drei Argumenten das erste zurückliefert. Wie ist diese zu definieren, so daß sie mit verschiedenen Typen arbeitet, und nicht für jeden Typen erneut definiert werden muß? Man definiert ihren Typ polymorph.

```
f :: a -> b -> c -> a
f x _ _ = x
```

Der Unterschied wird von Haskell anhand der mit einem Großbuchstaben beginnenden Namen konkreter Typen und der mit Kleinbuchstaben beginnenden Namen polymorpher Typen erkannt.

5.4.8 Überladen

Einen echten Unterschied gibt es zwischen polymorphen und überladenen Funktionen. Eine polymorphe Funktion besitzt *eine* Definition, die für alle Typen gültig ist; eine überladene Funktion dagegen wird für jeden Typen explizit definiert. Ein Beispiel hierfür wäre die Gleichheitsrelation, die für *Integers* anders definiert ist als für Wahrheitswerte, und die es z. B. für Funktionstypen in Haskell nicht gibt.

5.4.9 Typklassen

Es geht nun weiterhin um die Gleichheit `(==)` als Beispiel für eine überladene Funktion. Es wurde bereits erwähnt, daß es verschiedene Typen gibt, die auf Gleichheit getestet werden können, und daß es auch solche gibt, bei denen dies nicht der Fall ist. Bei welchen Typen kann nun der Test auf Gleichheit anwandt werden? Es sind dies alle in der Klasse *Eq* enthaltenen Typen.

```
Main> :t (==)
(==) :: Eq a => a -> a -> Bool
```

Mitglieder einer Typklasse werden Instanzen genannt. Vordefinierte Instanzen der Klasse *Eq* schließen die Typen *Int, Bool, Char, Float, Double* und auch *String* ein. Haskell stellt eine Anzahl vordefinierter Klassen zur Verfügung, so z. B.

- *Eq,* eine Klasse für Gleichheit und Ungleichheit;
- *Ord,* welche eine Ordnung über den enthaltenen Typen definiert;
- *Enum,* eine Klasse, die es gestattet, daß Elemente ihrer Instanzen aufgezählt werden können;
- *Show,* in der sich Elemente eines Typs in eine textuelle Form überführen und damit auf dem Bildschirm ausgeben lassen; *Read,* mit der sie sich aus Zeichenketten herauslesen lassen.

Eine vollständige Auflistung der vordefinierten Typklassen ist im *Haskell Report* zu finden.

Weiterhin ist es auch möglich, eigene Typklassen einschließlich ihrer Instanzen zu definieren.

5.5 Funktionskomposition

5.5.1 Funktionskomposition

Gefragt nach dem Nachfolger eines Zeichens in der ASCII-Tafel, kann z. B. eine Variable des Typs *Char* in den Typen *Int* konvertiert werden, dieser inkrementiert und zurück in ein Zeichen konvertiert werden. Die Implementierung ist recht intuitiv.

```
f :: Char -> Char
f c = chr (succ (ord c))
```

Es werden drei Funktionen komponiert, um eine weitere zu erhalten. Die Funktion *succ* ist vordefiniert und hat den Typen $Enum\ a \Rightarrow a \rightarrow a.$ Da *Char* keine Instanz der Klasse *Enum* ist, ist *succ* nicht direkt auf ein Zeichen anzuwenden. Es folgt eine elegantere Schreibweise obiger Definition.

```
f :: Char -> Char
f = chr . succ . ord
```

Es ist dies ein neuer Operator, der über Funktionen operiert. `(.)` komponiert zwei Funktionen, gesetzt den Fall, der Ergebnistyp der zweiten Funktion – die zuerst angewandt wird – stimmt mit dem Argumenttyp der ersten Funktion überein.

Die allgemeine Form des Komponierens von n Funktionen kann wie folgt niedergeschrieben werden: $(f_n.f_{n-1}.\cdots.f_1)(x) = f_n(f_{n-1}(\cdots f_1(x)\cdots))$. Der Punkt ist ein polymorpher Operator des Typs $(b \to c) \to (a \to b) \to (a \to c)$. Dies bedeutet, er übernimmt zwei Argumente, beides Funktionen. Die zweite Funktion wird zuerst angewandt, somit hat ihr Ergebnistyp mit dem Argumenttyp der ersten Funktion übereinzustimmen. Weiterhin stimmen der Argumenttyp der zweiten Funktion mit dem der resultierenden Funktion überein, wie auch der Ergebnistyp der ersten Funktion mit dem der resultierenden.

5.5.2 Funktionen höherer Ordnung

Funktionen, die andere Funktionen als Argument übernehmen, werden als Funktionen höherer Ordnung *(higher order functions)* oder Funktionale bezeichnet. Die eben eingeführte Funktion `(.)` ist offensichtlich solch eine Funktion höherer Ordnung.

5.6 Rekursion

Anstelle einer formalen Definition der Rekursion soll hier eine kleines Beispiel[2] in die jeweiligen Aspekte und die entsprechende Haskell-Syntax einführen. Sei das Beispielobjekt ein Geschäft mit einem wöchentlichen Umsatz und einer Funktion *sales*, welche (vom Typ $Int \to Int$) zu einer Wochennummer den zugehörigen Umsatz liefert.

5.6.1 Gesamtumsatz

Gefragt nach dem Gesamtumsatz der Periode der Wochen 0 bis n, kann nun folgende Funktion definiert werden, die für ein beliebiges n die Summe $sales\ 0 + sales\ 1 + \ldots + sales\ n$ liefert. Dies wird rekursiv realisiert.

```
totalSales :: Int -> Int
totalSales n | n == 0    = sales 0
             | otherwise = totalSales (n-1) + sales n
```

Es sollte nicht schwerfallen, dies einmal für $n = 2$ nachzuvollziehen. Seien die Umsätze dazu wie in der folgenden Tabelle dargestellt.

Week	0	1	2	...
Sales	7	2	5	...

Es folgt die Berechnung.

[2] Entnommen: S. Thompson, *Haskell – The Craft of Functional Programming,* 1. Auflage

```
totalSales 2
= totalSales 1 + sales 2
= (totalSales 0 + sales 1) + sales 2
= (sales 0 + sales 1) + sales 2
= (7 + 2) + 5
= 14
```

5.6.2 Maximaler Umsatz

Ein andermal wird nach dem maximalen Umsatz innerhalb einer bestimmten Periode – der Wochen 0 bis n – gefragt.

```
maxSales :: Int -> Int
maxSales 0 = sales 0
maxSales n = max (maxSales (n-1)) (sales n)
```

Es folgt die Berechnung.

```
maxSales 2
= max (maxSales 1) (sales 2)
= max (max (maxSales0) (sales 1)) (sales 2)
= max (max (sales 0) (sales 1)) (sales 2)
= max (max 7 2) 5
= 7
```

Hierzu gehört die Frage nach der Woche, in welcher der maximale Umsatz getätigt wurde. Die Definition folgt.

```
maxSalesWeek :: Int -> Int
maxSalesWeek n = salesWeek n (maxSales n)
  where
  salesWeek :: Int -> Int -> Int
  salesWeek n maxS
    | sales n == maxS = n
    | n == 0          = error "Error in maxSales!"
    | otherwise       = salesWeek (n-1) maxS
```

5.6.3 Woche ohne Umatz

Wiederum eine andere Frage mag sich nach dem Vorhandensein einer Woche ohne Umsatz erkundigen. Für Fragen dieser Art gibt es den Ergebnistyp *Bool*.

```
noSales :: Int -> Bool
noSales 0 = sales 0 == 0
noSales n = noSales (n-1) || sales n == 0
```

Folgend die BOOLEsche Berechnung.

```
noSales 2
= noSales 1 || sales 2 == 0
= (noSales 0 || sales 1 == 0) || sales 2 == 0
= ((sales 0 == 0) || sales 1 == 0) || sales 2 == 0
= ((7 == 0) || 2 == 0) || 5 == 0
= (False || False) || False
= False
```

Bleibt schließlich die Frage nach der entsprechenden Woche ...

```
noSalesWeek n
  | sales n == 0 = n
  | n == 0       = error "There is no week without sales!"
  | otherwise    = noSalesWeek (n-1)
```

5.7 Programmieren mit Listen

5.7.1 Einführung

Aus dem vorangegangenen Kapitel sind die Datentypen *Char* und *String* bekannt. An dieser Stelle soll das Verhältnis zwischen diesen beiden ein wenig näher beleuchtet werden. Der Typ *String* ist nicht wirklich ein Datentyp, er ist vielmehr ein Typ-Synonym für eine Liste des Typs *Char.*

Eine Liste ist eine Folge von Werten gleichen Typs. Sie wird kurzerhand dargestellt als eine Folge von Elementen, eingeschlossen in quadratischen Klammern und separiert durch Kommata. So können z. B. Listen des Typs *Int* definiert werden, oder auch des Typs *Char,* was – wie erwähnt – dem Typ *String* gleichbedeutend ist. Eine Liste des Typs *String* ist demzufolge eine Liste von Listen des Typs *Char.*

```
listOfInt :: [Int]
listOfInt = [1,2,3,4]

listOfChar :: [Char]
listOfChar = ['h','e','l','l','o']  -- oder: "hello"

listOfString :: [[Char]]            -- oder: [String]
listOfstring = [['o','n','e'], "two", "three"]
```

5.7.2 Konstruktion von Listen

Jede Liste ist entweder leer, `[]`, oder nicht-leer. Im letzteren Fall läßt sich die Liste in der Form `x:xs` darstellen, wobei `x` das erste Element der Liste ist, genannt *head* (der Kopf), und xs die verbleibende Liste, genannt *tail* (der Schwanz).

In Haskell ist jede Liste intern auf identische rekursive Weise aufgebaut. Sie besteht aus der leeren Liste `[]` und dem Listen-Konstruktor *cons* `(:)`. Letzterer ist eine rechts-assoziative Funktion des Typs $a \to [a] \to [a]$. Durch seine wiederholte Anwendung erhält man eine Liste mit mehreren Elementen.

```
[]      == []
1:[]    == [1]
2:[1]   == [2, 1]
3:[2,1] == 3:2:1:[] == [3, 2, 1]
```

Ein anderer zum Aufbau von Listen geeigneter Operator ist `(++)`, vom Typ $[a] \to [a] \to [a]$, welcher zwei Listen konkateniert. Dieser ist jedoch kein Konstruktor.

```
[1,2] ++ [] ++ [3,4,5] == [1, 2, 3, 4, 5]
```

Listen von Datentypen, die der Typklasse *Enum* angehören, lassen sich in Kurzform durch Angabe eines Bereiches angeben. Innerhalb dieses Bereiches besteht ferner die Möglichkeit der Angabe einer Schrittweite.

```
[2 .. 8]      == [2, 3, 4, 5, 6, 7, 8]
[1, 3 .. 12] == [1, 3, 5, 7, 9, 11]
```

5.7.3 *Pattern matching* über Listen

Muster über Listen können unter anderem wie folgt konstruiert werden.

- Ein nicht-strukturiertes `xs` paßt auf jede Liste.
- `(_:_)` ist ein Konstruktor-Muster, welches auf jede nicht-leere Liste zutrifft, ohne deren Elemente an Variablen zu binden.
- `(x:xs)` paßt ebenfalls auf jede nicht-leere Liste. Der Kopf der Liste wird an die Variable `x` gebunden, der Schwanz an die `xs`.
- `(_:[])` ist identisch mit `[_]` und paßt auf jede Liste mit genau einem Element, ohne dieses an eine Variable zu binden.
- `(x1:x2:xs)` extrahiert die beiden ersten Elemente einer mindestens zweielementigen Liste und bindet sie an die Variablen `x1` und `x2`, sowie die Restliste an `xs`.

5.7.4 *List Comprehension*

Haskell bietet eine Möglichkeit, über einer Auswahl spezifischer Elemente einer Liste anstelle der gesamten Liste zu operieren. Dies ist als *list comprehension* bekannt. Eine mathematische Mengennotierung, etwa $M = \{x \mid x \in G, x \text{ gerade}\}$, kann auf diese Weise direkt in Haskell-Code überführt werden.

```
[x | x <- g, even x == True]
```

Der Senkrechtstrich wird gewöhnlich als „so daß" gelesen. Der Ausdruck $x \leftarrow g$ ist als „Generator" bekannt und bestimmt, daß x die Werte aller Elemente aus der Menge g annimmt, eines nach dem anderen, diejenigen akzeptierend, welche der Bedingung des folgenden *guard* entsprechen – in diesem Fall: alle geraden Zahlen. Es können mehrere *guards* spezifiziert werden, die dann durch Kommata getrennt und damit konjunktiv verknüpft werden. Als ein weiteres Beispiel folgt die Liste aller möglicher geordneter Paare, gebildet aus den Elementen zweier Listen.

```
pairs :: [a] -> [b] -> [(a,b)]
pairs xs ys = [(x,y) | x <- xs, y <- ys]
```

5.7.5 Rekursion über Listen

Die folgende Funktion summiert mit Hilfe des *pattern matching* die Elemente einer Liste auf.

```
sum :: [Int] -> Int
sum []     = 0
sum (x:xs) = x + sum xs
```

Folgend ein weiteres Beispiel, bei dem eine Liste nach einem Auswahlkriterium gefiltert wird. Beide Funktionen sind bereits vordefiniert.

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

Das erste Argument ist hier eine Funktion, die ein Listenelement nimmt und zu diesem entweder „ja" oder „nein" sagt. Das zweite Argument ist eine Liste entsprechenden Typs, wie auch das Ergebnis. Im Fall der nicht-leeren Liste baut *filter* nun die Ergebnisliste auf, indem es hierzu nacheinander all diejenigen Elemente benutzt, die von der übergebenen Funktion akzeptiert werden.

Eine Klasse für sich sind die Faltungsfunktionen. Angenommen, zu einer Liste des Typs *Int* wird deren minimales Element gesucht. Haskell verfügt bereits über die vordefinierte Funktion *min,* die das Minimum zweier *Integers* zurückliefert. Ein korrekter Ansatz wäre nun die wiederholte Anwendung von *min* auf jeweils zwei Elemente der Liste mittels Rekursion. Haskell bietet jedoch einen eleganteren Weg.

```
minElem :: [Int] -> Int
minElem = foldr1 min
```

foldr1 ist vom Typ $(a \to a \to a) \to [a] \to a$. Im obigen Beispiel wurde die Funktion *min* hierauf partiell angewandt. Entstanden ist die Funktion zur Bestimmung des minimalen Listenelements. Zum Beweis folgt die Definition von *foldr1.*

$$\text{foldr1}(f, [x_1, \ldots, x_n]) = \begin{cases} f(x_1, f(x_2, \ldots f(x_{n-1}, x_n) \ldots)) & n \geq 2 \\ x_1 & n = 1 \\ \mathit{Error} & n = 0 \end{cases}$$

foldr1 faltet rechts-assoziativ Listen mit mindestens einem Element. Wird eine Operation benötigt, die auch über der leeren Liste definiert ist, so kann hierzu *foldr* benutzt werden. *foldr* ist vom – etwas allgemeineren – Typ $(a \to b \to a) \to a \to [b] \to a$ und verhält sich wie folgt.

$$\text{foldr}(f, z, [x_1, \ldots, x_n]) = \begin{cases} f(x_1, f(x_2, \ldots f(x_{n-1}, f(x_n, z)) \ldots)) & n \geq 1 \\ z & n = 0 \end{cases}$$

Bei den Funktionen *foldl1* und *foldl* handelt es sich um das jeweilige links-assoziative Äquivalent.

$$\text{foldl}(f, z, [x_1, \ldots, x_n]) = \begin{cases} f(f(\ldots, f(f(z, x_1), x_2) \ldots, x_{n-1}), x_n) & n \geq 1 \\ z & n = 0 \end{cases}$$

5.7.6 Vordefinierte Funktionen über Listen (Auswahl)

Im folgenden ist eine Auswahl an über Listen operierenden Funktionen aufgeführt, die zum Haskell-Sprachumfang gehören. Gegeben sind Name und Typ einer jeden Funktion.

```
(:)        :: a -> [a] -> [a]
(++)       :: [a] -> [a] -> [a]
(!!)       :: [a] -> Int -> a
map        :: (a -> b) -> [a] -> [b]
filter     :: (a -> Bool) -> [a] -> [a]
head       :: [a] -> a
last       :: [a] -> a
tail       :: [a] -> [a]
init       :: [a] -> [a]
length     :: [a] -> Int
null       :: [a] -> Bool
take       :: Int -> [a] -> [a]
drop       :: Int -> [a] -> [a]
takeWhile :: (a -> Bool) -> [a] -> [a]
dropWhile :: (a -> Bool) -> [a] -> [a]
concat     :: [[a]] -> [a]
iterate    :: (a -> a) -> a -> [a]
```

```
repeat    :: a -> [a]
replicate :: Int -> a -> [a]
foldr1    :: (a -> a -> a) -> [a] -> a
foldl1    :: (a -> a -> a) -> [a] -> a
foldr     :: (a -> b -> b) -> b -> [a] -> b
foldl     :: (a -> b -> a) -> a -> [b] -> a
```

5.8 Algebraische Datentypen

Bisher wurden grundlegende Datentypen wie *Int, Bool, Char,* eingeführt, dann Listen und der Typ *String* als eine spezielle Liste. Im folgenden werden nun komplexere Datentypen betrachtet. Das folgende Kapitel sei den algebraischen Datentypen gewidment.

5.8.1 Aufzählbare Typen

Die einfachste Art algebraischer Typen wird durch Aufzählung der Elemente des entsprechenden Typs definiert. So kann z. B. ein Typ *Institute* definiert werden, den Datentyp aller Institute der Fakultät – oder auch ein Typ *Chair,* als Datentyp aller Lehrstühle.

```
data Institute = ThI | TeI | SMT | SyA | KI | AI
data Chair     = AAT | ALG | GdP | IKT | ...
```

Eine Typ-Definition wird durch das Schlüsselwort *data* eingeleitet, gefolgt vom Namen des Typs und dessen Elementen wie oben aufgeführt. Die Elemente sind in diesem einfachen Fall gleichzeitig die (0-ären) Konstruktoren des entsprechenden Typs. Eine über diesen neuen Typen operierende Funktion kann nun beispielsweise wie folgt definiert werden.

```
chairs :: Institute -> [Chair]
chairs ThI = [AAT, ALG, GdP]
chairs SyA = [BS, DB, IKT, RN]
...
```

5.8.2 Produkttypen

Um zwei oder mehr Datentypen miteinander zu kombinieren, bildet man das algebraische Produkt. Um bei Personen den Namen mit dem Alter zu verknüpfen ist also neben der bereits bekannten Tupel-Bildung auch folgendes denkbar.

```
type Name   = String
type Age    = Int
data People = Person Name Age
```

Person ist hier der Konstruktor des Datentyps *People.* Er ist vom Typ *Name* $\rightarrow$ *Age* $\rightarrow$ *People.* Etwas bildlicher kann man ihn sich als Funktion vorstellen, die aus den Typen *Name* und *Age* ein Element des Datentyps *People* generiert.

5.8.3 Alternative Typen

An dieser Stelle soll nun das obige Beispiel wie folgt modifiziert werden: Eine Person ist entweder ein Student oder ein Mitarbeiter (an der Universität). Ersterer hat einen Namen und eine Matrikelnummer, letzter hat einen Namen und gehört einem Institut an. Eine entsprechende Implementierung dieses Sachverhalts in Haskell könnte etwa wie folgt aussehen.

```
type Name   = String
type Matr   = Int
data Inst   = ThI | TeI | SMT | SyA | KI | AI
data People = Student Name Matr | Staffer Name Inst
```

Der alternative Typ ist hier in jedem Fall ein Produkttyp. Welcher von beiden, das hängt davon ab, mit welchem Konstruktor – *Student* oder *Staffer* – er initialisiert wird.

5.8.4 Rekursive Typen

Als Beispiel für rekursive Typen wird im folgenden ein algebraischer Baum betrachtet. Ein Baum ist im Prinzip nichts anderes als ein alternativer Typ, nämlich entweder ein leerer Baum oder ein Baum, bestehend z. B. aus einem mit einem *Integer* beschrifteten Knoten und zwei Unterbäumen gleicher Art.

```
data Tree = Nil | Node Int Tree Tree
```

Ein Beispiel hierfür folgt, wie auch in Abbildung 5.1 dargestellt.

```
myTree :: Tree
myTree = Node 1 (Node 2 (Node 4 Nil Nil) (Node 5 Nil Nil))
                (Node 3 Nil Nil)
```

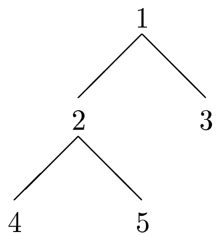

Abbildung 5.1: Ein binärer Baum

Es kann nun leicht eine rekursive Funktionen definiert werden, welche die Tiefe dieses Baumes ermittelt.

```
depth :: Tree -> Int
depth Nil            = 0
depth (Node _ t1 t2) = 1 + max (depth t1) (depth t2)
```

5.8.5 Klasseninstanzen

Wurde ein neuer Typ eingeführt, mag es erforderlich sein, über einen Test auf Gleichheit zweier seiner Elemente zu verfügen, oder über eine Ordnung über seinen Elementen etc. Diese Eigenschaften können vererbt werden.

```
data Institute = ThI | TeI | SMT | SyA | KI | AI
  deriving (Eq, Ord, Enum, Show, Read)
```

Elemente des Datentyps *Institute* lassen sich nun vergleichen, ordnen, anzeigen etc. – wenn auch einige dieser Eigenschaften in diesem Fall nicht sehr sinnvoll sind.

```
Main> ThI == TeI           -- die Klasse Eq benutzend
False
Main> ThI < TeI            -- die Klasse Ord benutzend
True
Main> [SMT .. AI]          -- mittels der Klassen Enum und Show
[SMT, SyA, KI, AI]
```

5.8.6 Polymorphe Typen

Das Thema polymorpher algebraischer Datentypen schließt direkt an das bisherige Wissen über Polymorphismus an. Ein polymorpher binärer Baum würde z. B. wie folgt definiert.

```
data Tree a = Leaf a | Node (Tree a) (Tree a)
```

Eine spezielle Instanz könnte sich nun wieder mit *Integers* befassen.

```
myTree :: Tree Int
myTree = Node (Node (Leaf 4) (Leaf 5)) (Leaf 3)
```

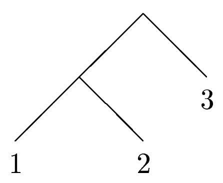

Abbildung 5.2: Instanz eines polymorphen Baumes

5.9 Abstrakte Datentypen

5.9.1 ADTs in Haskell

In diesem Teil sollen kurz die abstrakten Datentypen, kurz ADTs, in Haskell eingeführt werden. Ein abstrakter Datentyp ist ein mehr oder weniger komplexer Mechanismus, der gewisse Funktionalitäten exportiert, gewissermaßen als Interface. Die internen Strukturen des Datentyps werden nicht exportiert und bleiben dem Nutzer daher verborgen. Im folgenden wird als prominentes Beispiel dieser Klasse einen Stack definiert.

5.9.2 Beispiel: Stack

Ein Stack ist ein Datenreservoir, welches gemäß dem LIFO-Prinzip *(last in – first out)* arbeitet. Ein Stack verfügt üblicherweise über drei Zugriffsmethoden, das sind *push,* um ein Datum auf der Spitze des Stacks abzulegen, *pop,* um das Datum auf der Spitze zurückzuholen, und *is_empty,* um zu überprüfen, ob der Stack leer ist. Diese Begriffe werden im folgenden übernommen, um einen Stack zu implementieren.

```
module Stack (Stack, isEmpty, push, pop) where

data Stack a = Empty | Push a (Stack a)

isEmpty :: Stack a -> Bool
isEmpty Empty      = True
isEmpty (Push _ _) = False

push :: a -> Stack a -> Stack a
push x xs = Push x xs

pop :: Stack a -> (a, Stack a)
pop Empty       = error "Empty stack"
pop (Push x xs) = (x, xs)
```

Um dieses Modul in ein Programm einzubinden, ist es z. B. in einer Datei unter dem Namen `Stack.hs` abzuspeichern. Die Definitionen werden anschließend wie folgt durch Import des Moduls in einer beliebigen anderen Datei verfügbar gemacht.

```
import Stack
```

5.10 λ-Notation

Neben der Definition benannter Funktionen ist in Haskell auch eine direkte Implementierung anonymer Funktionen mittels λ-Kalküls vorgesehen. Das λ wird hierbei durch den *backslash* „\“, der folgende Punkt durch den typischen Pfeil „`->`“ ersetzt.

Der Ausdruck $\lambda x.\lambda y. + x\, y$ würde in Haskell somit wie folgt notiert.

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

Ebenso zulässig ist die Kurzform. Mit

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

wird der Definition ein Funktionsname zugewiesen.

Literaturverzeichnis

[1] Simon Peyton Jones und John Hughes (Hrsg.)
Report on the Programming Language Haskell 98
1998
`http://www.haskell.org/definition/`

[2] Simon Peyton Jones und John Hughes (Hrsg.)
Standard Libraries for the Haskell 98 Programming Language
1998
`http://www.haskell.org/definition/`

[3] Simon Thompson
Haskell – The Craft of Functional Programming (2. Auflage)
Addison-Wesley, 1999

[4] Rex Page
Two Dozen Short Lessons in Haskell
School of Computer Science, University of Oklahoma, 1997
`http://www.cs.ou.edu/cs1323h/textbook/haskell.shtml`