

Haskell - kurz und schmerzlos

Martin L sch

2. Februar 2004

Inhaltsverzeichnis

I	für den Anfang ...	4
1	Grundlagen	5
1.1	Kommentare	5
1.2	Eingabe und Aufruf	5
2	Grundlegender Programmaufbau	6
2.1	Das einfachste Programm	6
2.2	Funktion mit Parameter	6
2.3	Verwendung von Parametern zur Berechnung	7
2.4	Problemlösung mittels Rekursion	7
2.5	Größere Programme mittels Hilfsfunktionen	8
3	Listen	9
3.1	grundlegende Syntax	9
3.2	Zugriff auf das vorderste Element	9
3.3	Verketteten von Listen	10
4	Erweitertes Pattern-Matching	11
II	zum Weitermachen ...	12
5	die Datentypen von Haskell	13
5.1	ganzzahlige Typen	13
5.2	Fließkomma-Typen	13
5.3	Zeichen-Typen	13
5.4	Boolesche Werte	14
6	Relationen und Operatoren	15
6.1	Operatoren für Zahlentypen	15
6.2	Operatoren für Boolesche Werte	15
6.3	Relationen	15
7	Tupel	16
III	wenn's noch nicht reicht ...	17
8	Lesbarkeit erhöhen	18
8.1	where-Ausdrücke	18
8.2	let-Ausdrücke	18
8.3	eigene Infix-Operatoren	19

9	eigene Datentypen	20
9.1	Typ-Synonyme	20
9.2	”echt-eigene” Datentypen	20
9.3	eigene Datentypen durch Aufzählen	21
9.4	rekursive Datentypen	21
10	Funktionen höherer Ordnung	23
10.1	map	23
10.2	foldl	23
10.3	foldr	24
10.4	filter	25
10.5	zip	25
10.6	zipWith	26
11	Typklassen	27
11.1	Was sind Typklassen überhaupt?	27
11.2	Definition eigener Typklassen	27
11.3	direkte Verwendung von Typ-Klassen	28
11.4	indirekte Verwendung vordefinierter Typklassen	30
11.4.1	Show	31
11.4.2	Eq	31
11.4.3	Ord	31
11.5	Anwendung von Typklassen	31
IV	das Beste zum Schluß ...	33
12	Lambda-Abstraktionen	34
13	Listen generieren	35
13.1	Aufzählungen	35
13.2	Listengeneratoren	36
13.3	iterierte Funktionen	36
13.4	Element-Wiederholungen	37
A	Listen-Operationen	38
B	Funktionen höherer Ordnung	39
C	vordefinierte Typ-Klassen	40
C.1	Enum	40
C.2	Eq	40
C.3	Num	40
C.4	Ord	40
C.5	Show	40

Vorwort

Hallo alle miteinander!

Um von Anfang an allen Mißverständnissen vorzubeugen: dieser Text (der nebenbei keinen Anspruch auf Vollständigkeit oder Korrektheit erhebt, gefundene Fehler bitte an dukemarty@gmx.de mailen ...) soll keine formale Einführung in die funktionale Programmiersprache Haskell darstellen (dieser Programmpunkt wird in der Info1-Vorlesung wohl zur genüge abgehandelt werden).

Stattdessen möchte ich, ausgehend von Problemen, die sich bei der Anwendung von Haskell ergeben können, die (für die Praxis, sprich Übungsblätter) wichtigen Dinge vermitteln, die man wissen sollte - das ganze grob unterteilt in Anfänger-, Fortgeschrittenen- und zwei Profi-Teile.

Ich habe dabei versucht, den nötigen Stoff möglichst knapp, ja soweit möglich auf einer Seite abzuhandeln, und dabei möglichst viele Beispiele einfließen zu lassen.

Die Abbildungen im Kapitel über Funktionen höherer Ordnung verdanken wir Markus Westphal, einem meiner geschätzten Info1-Tutorenkollegen (Danke nochmal an dieser Stelle :-).

Damit wäre wohl alles wichtige gesagt, denke ich, also bleibt nur noch: viel Spaß und Erfolg, zu dem dieses kleine Skript hoffentlich seinen Teil beitragen wird.

Karlsruhe, den 2. Februar 2004

Martin Lösch

Teil I
für den Anfang ...

Kapitel 1

Grundlagen

Worüber man sich zuerst einmal klar werden sollte, besonders wenn man schon einmal in einer "normalen" Sprache programmiert hat, ist folgendes: Haskell unterscheidet sich als funktionale Programmiersprache in einigen wichtigen konzeptionellen Punkten ganz erheblich von einer imperativen Programmiersprache. Während man beim imperativen Programmieren einem Rechner Befehle erteilt (daher der Name), wie und welche Speicherstellen manipuliert werden sollen, ordnet man beim funktionalen Programmieren Werten (die aus Ausdrücken berechnet werden) Namen zu, und zwar einmalig; das heißt insbesondere, daß keine Variablen existieren, deren Wert geändert werden könnte. Weiterhin existieren als Folge dieses Konzeptes üblicherweise auch keine Schleifen (die stattdessen durch Rekursion nachgebildet werden müssen).

In diesem Kapitel wollen wir uns kurz einige allgemeinere Konstrukte und mögliche Probleme in Haskell ansehen, die noch gar nicht direkt mit der Programmierung zu tun haben.

1.1 Kommentare

Wie in jeder anderen Sprache auch, besteht in Haskell die Möglichkeit, *Kommentare* in den Programm-Quelltext einzufügen, um anderen das Nachvollziehen des Programmablaufs leichter zu machen (und auch dem Programmierer selbst, sollte er in die Verlegenheit kommen, sich sein eigenes Programm nach längerer Zeit noch einmal ansehen zu müssen).

In Haskell werden Kommentare, die eine ganze Zeile umfassen, durch `--` eingeleitet; sie dürfen an beliebiger Stelle stehen, ohne den Programmablauf in irgendeiner Art und Weise zu beeinflussen.

Soll ein ganzer Kommentar-Block eingefügt werden (oder umgekehrt ein Block auskommentiert werden), so wird dieser Block in `{- und -}` eingeschlossen. (ein solcher Kommentar kann auch in der Mitte einer Zeile, z.B. nach einer Anweisung, beginnen).

1.2 Eingabe und Aufruf

Bei der Eingabe von Zahlen als Parameter ist zum Einen darauf zu achten, daß Kommazahlen mit Dezimalpunkt statt Dezimalkomma unterteilt werden, und zum anderen darauf, daß negative Zahlen (mit vorangestelltem Minus) in Klammern eingeschlossen werden müssen, weil sonst die Zugehörigkeit des Minus zu der Zahl nicht erkannt wird.

Wenn wir also einer Funktion *hallo* (was darunter zu verstehen ist, wird im nächsten Kapitel erklärt) eine `-13` als Parameter übergeben wollten, müßten wir schreiben:

```
hallo (-13)
```

Kapitel 2

Grundlegender Programmaufbau

Ein Haskell-Programm besteht im Prinzip nur aus verschiedenen Funktionen, die sich wechselseitig und ggf. auch rekursiv aufrufen und dadurch das gewünschte Ergebnis generieren. Die Ausgabe besteht üblicherweise aus dem Wert, der für die aufgerufene Funktion berechnet wurde.

2.1 Das einfachste Programm

Das führt uns zu einem möglichst einfachen Programm, dessen einzige *Funktion* immer einen Konstanten Wert liefert:

```
test = "Hallo Welt!"
```

Dabei kümmern wir uns vorerst einmal nicht darum, inwieweit Haskell Strings (also Zeichenketten) definiert und unterstützt.

2.2 Funktion mit Parameter

Wenn wir nun einen Schritt weitergehen wollen, könnten wir der Funktion einen *Parameter* "spendieren"; vorher sollte man aber den Interpreter darüber informieren, von welchem Typ dieser Parameter sein soll (d.h. aus welchem Wertebereich die Werte, die man übergibt, stammen müssen). Das nennt man die *Signatur* der Funktion, die allgemein folgenden Aufbau hat:

```
Funktionsname :: ParamTyp1 -> [... ParamTypk ->] ErgebnisTyp
```

Wenn wir das in unser Beispiel-Programm einfügen, und zwischen drei möglichen Parametern (1, 2 oder was anderes) unterscheiden, sieht das dann so aus:

```
test :: Int -> String
test 1 = "Hallo Welt!"
test 2 = "Hallo Karlsruhe!"
test a = "Hallo Aliens!"
```

Ob eine der Definitionen verwendet werden kann, entscheidet der Interpreter mittels *Mustererkennung*, d.h. er prüft, ob der aktuelle Aufruf das gleiche Muster aufweist wie die Definition (also ist die 2. Zeile dann ausführbar, wenn der Parameter den Wert 1 hat, die 3. Zeile beim Wert 2).

Das "a" beim 3. Fall ist ein sogenannter *formaler Parameter*, d.h. daß dieser Fall unabhängig vom aktuellen Wert des Parameters benutzt werden kann; der formale Parameter kann dann auf der rechten Seite des Gleichheitszeichens wie eine Konstante verwendet werden.

Beim Ausprobieren des Programmes ist vor allem auf 2 Punkte zu achten: zum Einen wird zwischen Groß- und Kleinschreibung unterschieden, also ist "Int" richtig, "int" aber nicht; zum Anderen ist die Reihenfolge der verschiedenen zu unterscheidenden Fälle relevant: um das zu zeigen, ergänzt man das obige Programm um folgenden, beinahe identischen Ausschnitt:

```
test2:: Int -> String
test2 a = "Hallo Aliens!"
test2 1 = "Hallo Welt!"
test2 2 = "Hallo Karlsruhe!"
```

Nun wird unabhängig vom Wert des Parameters immer der Wert der 2. Zeile zurückgegeben, weil der Interpreter die Liste der möglichen Muster von oben nach unten durchläuft, und schon bei der 2. Zeile auf ein zutreffendes Muster stößt; die weiter unten stehenden Fälle werden dann überhaupt nicht mehr betrachtet.

2.3 Verwendung von Parametern zur Berechnung

Wie wird nun etwas mit den Parametern berechnet? Nun, am einfachsten verwenden wir für das folgende Programm die Grundrechenarten, die mithilfe der üblichen Symbole benutzt werden können:

```
quadrat:: Int -> Int
quadrat 0 = 0
quadrat a = a * a
```

Dabei wurde in diesem Programm der Fall $a=0$ nur deshalb herausgenommen, um zu zeigen, daß man auch in einem solchen Fall den Parameter für Berechnungen verwenden kann; das Programm tut also das Gleiche wie das folgende:

```
quadrat2:: Int -> Int
quadrat2 a = a*a
```

Hat eine Funktion mehrere Parameter, für die aktuelle Berechnung ist aber nur ein Teil der Parameter entscheidend, so kann statt der übrigen Parameter ein Unterstrich `_` eingetragen werden, um eben diesen Sachverhalt zu verdeutlichen.

2.4 Problemlösung mittels Rekursion

Wie löst man nun aber ein Problem, dessen Größe vorher noch nicht bekannt ist, wenn man nicht einfach irgendwelche Anweisungen wiederholen kann? Nun, die Antwort lautet: mittels *Rekursion*, also indem sich eine Funktion selbst nochmal aufruft; dabei muß man natürlich darauf achten, daß die Rekursion auch irgendwann wieder abbricht, üblicherweise indem man einen der Parameter der Funktion beim Selbstaufwurf verändert, sodaß irgendwann ein anderer Fall der Funktionsdefinition ausgewählt wird.

Betrachten wir dazu einmal folgendes Programm, das dazu dienen soll, die Fakultät einer Zahl X zu berechnen:

```
fakultaet:: Int -> Int -> Int
fakultaet 0 0 = 0
fakultaet a 0 = a
fakultaet a b = fakultaet (a*b) (b-1)
```

berechnet die Fakultät einer Zahl x , also den Wert $1 \cdot 2 \cdot \dots \cdot x$

Der Aufruf, um die Fakultät von X zu berechnen, lautet: `fakultaet 1 X`

Dabei wird in jedem Schritt der übergebene Parameter um eins erniedrigt, aber vorher noch an das bisher errechnete (Teil-)Ergebnis dranmultipliziert.

2.5 Größere Programme mittels Hilfsfunktionen

Mit dem, was wir bisher gelernt haben, stößt man auf 2 große Fragen:

1. wie soll man mit einer solchen Funktion eine größere Funktionalität für den Benutzer des Programms anbieten können
2. das obige Programm zur Berechnung der Fakultät war nicht gerade benutzerfreundlich, da der Benutzer nicht nur "seinen eigenen" Parameter (nämlich X) angeben, sondern sich auch daran erinnern muß, noch eine zusätzliche 1 einzugeben

Beide Probleme kann man auf einen Schlag lösen:

In einem Haskell-Programm können mehrere verschiedene Funktionen definiert werden, die sich gegenseitig aufrufen; und zur Lösung der obigen Probleme verwendet man einfach einige der Funktionen als Hilfsfunktionen für die Funktion, die vom Benutzer letztendlich aufgerufen werden soll. In Abwandlung und Ergänzung des obigen Programmes zur Berechnung der Fakultät einer Zahl sieht das dann wie folgt aus:

```
fakhilf:: Int -> Int -> Int
fakhilf a 0 = a
fakhilf a b = fakhilf (a*b) (b-1)

fakultaet:: Int -> Int
fakultaet 0 = 0
fakultaet a = fakhilf 1 a
```

Der Aufruf zur Berechnung von $X!$ lautet nun einfach noch: `fakultaet X`.

Die Funktion `fakultaet`, die wir jetzt benutzen, ruft einfach die Hilfsfunktion, unsere ehemalige Fakultätsfunktion, mit dem zusätzlich benötigten Parameter 1 auf.

Nachdem wir uns bisher mit den Grundlagen eines Haskell-Programmes beschäftigt haben, wenden wir uns im nächsten Kapitel einem weiteren wichtigen Baustein von Haskell zu, den man benötigt, um größere Datenmengen (von variabler Größe) bearbeiten zu können.

Kapitel 3

Listen

3.1 grundlegende Syntax

Eine *Liste* von Elementen besteht im Grunde (wenn man sie z.B. als Parameter übergeben will) aus einer Reihe von Daten gleichen Typs, die jeweils durch ein Komma getrennt und in eckige Klammern eingeschlossen sind.

```
[1,5,34,45,12,423,4]
```

In der Signatur einer Funktion schließt man den Typ, von dem die einzelnen Listen-Elemente sein sollen, in eckige Klammern ein (auch wenn es dadurch implizit schon klar sein sollte, sei es doch noch einmal ausdrücklich erwähnt: natürlich müssen alle Elemente einer Liste den gleichen Typ haben):

```
Funktion1:: [ParamTyp] -> ErgebnisTyp
```

Die Funktion Funktion1 errechnet also aus einer Liste, die Elemente vom Typ ParamTyp enthält, ein Ergebnis vom Typ ErgebnisTyp.

Die *leere Liste*, die durch [] definiert wird, wird häufig in Fallunterscheidungen zum Beenden der Rekursion eingesetzt; ein Beispiel zur Notation:

```
testlist:: [Int] -> String
testlist [] = "die Liste ist leer"
testlist x = "diese Liste ist NICHT leer!"
```

Diese Funktion übernimmt also eine Liste als Parameter, und liefert eine Meldung darüber, ob sie leer ist (erster Fall), oder ob sie nicht leer ist (was eigentlich gar nicht direkt überprüft, sondern einfach angenommen wird, da im zweiten Fall der erste offensichtlich nicht zutreffend war).

3.2 Zugriff auf das vorderste Element

Möchte man auf die einzelnen Elemente der Liste zugreifen, um sie für Berechnungen heranzuziehen, so kann man nur ein (oder mehrere) Elemente vom Anfang der Liste wegnehmen; zur Notation dieses Zugriffs wird ein Doppelpunkt eingesetzt; genauso wird ein Element an den Anfang einer Liste angehängt, wie im folgenden Beispiel zu sehen:

```

giberstes:: [Int] -> Int
giberstes [] = 0
giberstes (x:xs) = x

gibrest:: [Int] -> [Int]
gibrest [] = []
gibrest (x:xs) = xs

haengan:: [Int] -> Int -> [Int]
haengan b c = (c:b)

```

giberstes liefert das 1. Element der übergebenen Liste, gibrest löscht gewissermaßen das 1. Element einer Liste (bzw. gibt nur die übrigen Elemente zurück); haengan fügt den übergebenen Wert an den Anfang der übergebenen Liste ein

An diesem Programm sollen 2 Dinge gezeigt werden: zum Einen die zusätzliche Unterscheidung auf die leere Liste bei den beiden ersten Funktionen; sie ist deshalb nötig, weil es sonst zu Fehlermeldungen kommt, daß kein Element x (bei giberstes) bzw. keine Liste xs (bei gibrest) vorhanden ist. Zum anderen ist darauf zu achten, daß bei der Fallunterscheidung links des Gleichheitszeichens die Aufteilung der Liste $x:xs$ in normale Klammern eingeschlossen ist, damit sie vom Interpretier als ein Ausdruck (nämlich eine Liste) erkannt werden.

Das erworbene Wissen kann man nun z.B. einsetzen, um in einer Liste von Integern jede einzelne Zahl zu quadrieren:

```

quadlist:: [Int] -> [Int]
quadlist [] = []
quadlist (x:xs) = (x*x):(quadlist xs)

```

Ein anderes mögliches Beispiel wäre die Berechnung eines Polynomwertes an der Stelle x , wobei die Koeffizienten des Polynoms als Liste im Programm abgelegt werden (der Aufbau der Berechnung ist an das Horner-Schema angelehnt):

```

koeff:: [Int]
koeff = [1,2,3]

hornhlp:: [Int] -> Int -> Int -> Int
hornhlp [] _ erg = erg
hornhlp (x:xs) a erg = hornhlp xs a ((erg*a)+x)

horner:: Int -> Int
horner a = hornhlp koeff a 0

```

die (quadratische) Funktion wird durch die Hilfsfunktion koeff definiert, die die Koeffizienten der Funktion als Liste angibt

3.3 Verketteten von Listen

Um zwei oder mehr Listen zu *verketteten*, wird die ++-Funktion verwendet (wobei die Wortwahl "Funktion" statt "Operator" darauf hinweisen soll, daß ++ Teil der Funktionsbibliothek von Haskell ist, und nicht zum Grundgerüst der Sprache gehört). Das kann dann wie folgt aussehen:

```
["Apfel", "Birne"] ++ ["Melone", "Feige", "Kirsche"]
```

Beim Verketteten sollte man aber, ebenso wie beim vorne Anfügen eines Elementes mittels : daran denken, daß in Wirklichkeit nicht die vorhandene Liste verändert, sondern eine vollständig neue Liste erstellt wird, die (je nach dem) ein zusätzliches Element vorne bzw. die Elemente beider Ursprungslisten enthält.

Kapitel 4

Erweitertes Pattern-Matching

Wie schon relativ zu Anfang erwähnt, wird durch *Mustererkennung* bei einem Aufruf bestimmt, welche der Funktionsdefinitionen genau gemeint ist. Die oben vorgestellte Methode, daß auf der linken Seite des definierenden Gleichheitszeichens unterschiedliche Parameter auftreten, ist allerdings nicht für alle möglichen Probleme ausreichend flexibel. Zum Beispiel könnte der Fall auftreten, daß man darauf reagieren möchte, wenn einer der Parameter innerhalb eines bestimmten Wertebereichs liegt. In diesen Fällen ist die Notation des folgenden Beispiels sehr nützlich:

```
test:: Int -> String
test a | a<10 = "klein"
      | a>10 = "gross"
      | otherwise = "sehr seltsam"
```

Die Schreibweise funktioniert also so: am Anfang der Zeile wie üblich die Funktionsdefinition, allerdings alle Parameter als formale Parameter; dann folgt ein senkrechter Strich (auf die Einrückung in den weiteren Zeilen achten!), und dahinter eine Bedingung, die erfüllt sein muß, damit der hinter dem folgenden Gleichheitszeichen stehende Rumpf ausgeführt werden kann. Mit "otherwise" können schließlich noch alle möglichen, weiter oben nicht abgefangenen Fälle behandelt werden. Natürlich wird auch hier wieder von oben nach unten die erste passende Definition gesucht und verwendet.

Die einzige Einschränkung, der die Bedingung nach dem "|" genügen muß, ist, daß sie einen booleschen Wert darstellt. Deshalb sind auch zusammengesetzte Bedingungen erlaubt, wie sie durch boolesche Operatoren ermöglicht werden (siehe Kapitel 6). Das könnte dann etwa so aussehen:

```
groessennamen:: Int -> String
groessennamen a | (a>100) && (a<160) = "klein"
                | (a>=160) && (a<175) = "mittelgroß"
                | (a>=175) && (a<190) = "wirklich groß"
                | (a>=190) && (a<220) = "riesig"
                | True = "also, das ist eigentlich keine Größe"
```

Hier wird anhand einer übergebenen Zahl (die eine Größe in cm darstellen soll) entschieden, in welchem Intervall diese Zahl liegt, und somit, ob man diese Größe als klein, groß oder etwas anderes einordnen kann.

Teil II
zum Weitermachen ...

Kapitel 5

die Datentypen von Haskell

5.1 ganzzahlige Typen

An ganzzahligen Datentypen sind zwei, einander sehr ähnliche Typen zu nennen:

Int der in den bisherigen Beispielen verwendete Datentyp, der in der Größe beschränkt ist, aber der Wertebereich entspricht mindestens einer vorzeichenbehafteten 29-bit Binärzahl

Integer eine in der Größe unbeschränkte oder auch "mathematische" Ganzzahl dar, wobei auch dieser Unbeschränktheit natürlich gewisse, allerdings meist unerreichte Grenzen gesetzt sind

5.2 Fließkomma-Typen

Die beiden Fließkomma-Typen, die von Haskell standardmäßig zur Verfügung gestellt werden, sind `Float` und `Double`, die sich wieder in der Größe des darstellbaren Wertebereichs unterscheiden; intern werden beide Zahlentypen in Exponent-Darstellung gespeichert.

5.3 Zeichen-Typen

Haskell bietet einen Datentyp zum Aufnehmen eines einzelnen Zeichens: `Char`. Ein einzelnes Zeichen wird dabei in einzelne Hochkommata eingeschlossen, was wie folgt aussehen kann:

```
zeichen :: Char
zeichen = 'a'

zeichenio :: Char -> Char
zeichenio x = x
```

Ein `String`, wie er in den ersten Beispielen verwendet wurde, stellt nun nichts anderes als eine Liste von `Char`'s dar, wobei noch eine verkürzte Schreibweise zur Eingabe verwendet werden kann, indem man die einzelnen Zeichen direkt aneinander schreibt, und das ganze in die üblichen doppelten Hochkommata einschließt. Mit diesen Strings kann man dann wie mit einer Liste arbeiten, z.B. also eine Verkettung, in diesem Fall oft *Konkatenation* genannt:

```
text :: String
text = "Hallo"

textio :: String -> String
textio a = text ++ a
```

Neben der hier gezeigten Konkatenation unterstützen Strings auch noch den Vergleich auf Gleichheit und Ungleichheit mittels "==" bzw. "/=":

```
stringtest:: String -> String -> Bool
stringtest a b = a == b

stringuntest:: String -> String -> Bool
stringuntest a b = a /= b
```

5.4 Boolesche Werte

Mit Hilfe der Möglichkeiten, eigene Datentypen zu erstellen (die weiter hinten in Kapitel XXX noch vorgestellt werden) wird von Haskell schon ein Datentyp zur Aufnahme von Wahrheitswerten (oder *Booleschen Werten*) zur Verfügung gestellt, der nur die beiden Werte "True" oder "False" annehmen kann. Die beiden geraden erwähnten Vergleiche "==" und "/=" liefern z.B. Ergebnisse von diesem Typ.

Der zugehörige Datentyp wird mit Bool angegeben, wie schon im letzten Beispiel gesehen.

Kapitel 6

Relationen und Operatoren

Um Berechnungen mit und Vergleiche auf den verschiedenen, eben vorgestellten Datentypen durchzuführen, sollte es in Haskell Operatoren (zur Verknüpfung von Werten) und Relationen (zum Vergleichen) geben; dies ist natürlich wie in anderen Programmiersprachen auch der Fall.

6.1 Operatoren für Zahlentypen

Für Zahlentypen existieren folgende Operatoren:

+	Addition
-	Subtraktion
*	Multiplikation
/	(bei Int/Integer: ganzzahlige) Division
%	Modulo (Rest der ganzzahligen Division)
^	Exponentiation

6.2 Operatoren für Boolesche Werte

Für Boolesche Werte existieren die folgenden Operatoren:

&&	logische UND
	logisches ODER

6.3 Relationen

Für den Vergleich von Werten des gleichen Typs existieren folgende Relationen:

==	Gleichheit
!=	Ungleichheit
>	links größer als rechts
<	links kleiner als rechts
>=	links größer oder gleich rechts
<=	links kleiner oder gleich rechts

Kapitel 7

Tupel

Ein *Tupel* verbindet mehrere Typen (die unterschiedlich oder gleich sein können) zu einem neuen *Aggregattyp*, der z.B. verwendet werden kann, um mehrere Werte, die verschiedenen Typs sind, als Ergebnis zurück zu liefern. Definiert wird ein Tupel (z.B. in einer Signatur) wie folgt:

```
(Typ1,Typ2,...)
```

Entsprechend dieser Schreibweise wird auch ein Tupel als Parameter geschrieben, indem die Werte der einzelnen Teile in runden Klammern, getrennt durch Kommata, stehen.

In einem Programm, das zu einem übergebenen String den 1. Buchstaben und die Länge des gesamten Strings zurück liefert, könnte die Verwendung wie folgt aussehen:

```
-- liefert den 1. Buchstaben eines Strings und seine Länge
fstlaenge:: String -> (Char,Int)
fstlaenge (x:xs) = (x, laenge 1 xs)

-- addiert die Länge der Parameter-Liste zum 1. Parameter
laenge:: Int -> [a] -> Int
laenge b [] = b
laenge b (x:xs) = laenge (b+1) xs
```

Die Verwendung gleicher Typen wäre z.B. dann angebracht, wenn man irgendwelche Koordinaten verwendet (etwa bei der Umrechnung von kartesischen zu Winkel-Koordinaten).

Teil III

wenn's noch nicht reicht ...

Kapitel 8

Lesbarkeit erhöhen

So langsam stoßen wir in Bereiche vor, in denen Funktionsaufrufe immer länger und unübersichtlicher werden können; um dieses Problem in den Griff zu bekommen, kann man einerseits die Funktionen in kleinere Teilfunktionen zerlegen, denen man möglichst deutliche Namen gibt. Oder aber man schöpft die Möglichkeiten, die von Haskell geboten werden, weiter aus, in dem man `where`- oder `let`-Ausdrücke verwendet, wie sie in den nächsten beiden Abschnitten erläutert werden.

Unabhängig davon, welchen der beiden Befehle man wählt (auf den semantischen Unterschied möchte ich hier nicht eingehen), bieten beide die Möglichkeit, einen Teil der Berechnungen aus der Funktion auszugliedern, ihm einen Namen zu geben, und diesen in der Funktion zu verwenden. Dieses Ausgliedern findet aber direkt bei der Funktion selbst statt, und nicht zwei Bildschirmseiten entfernt, wie es mit echten Teilfunktionen der Fall wäre:

8.1 `where`-Ausdrücke

`where`-Ausdrücke werden der eigentlichen Funktion nachgestellt, wie im folgenden Beispiel zu sehen ist (das hoffentlich einige an Physik \rightarrow Elektrodynamik erinnert, wenn nicht - einfach den Sinn des Ganzen ignorieren):

```
coulombkraft:: Float -> Float -> Float -> Float
coulombkraft q1 q2 r = 1/ vier_pi_eps_0 * ladungsprodukt / abstandsquadrat
    where vier_pi_eps_0 = 9 * 10^9
          ladungsprodukt = q1 * q2
          abstandsquadrat = r^2
```

Man beachte wiederum die Einrückung der `where`-Klauseln!

8.2 `let`-Ausdrücke

Dasselbe Beispiel, diesesmal aber mit Hilfe von `let` realisiert:

```
coulombkraft:: Float -> Float -> Float -> Float
coulombkraft q1 q2 r = let vier_pi_eps_0 = 9 * 10^9
                        ladungsprodukt = q1 * q2
                        abstandsquadrat = r^2
                    in 1/ vier_pi_eps_0 * ladungsprodukt / abstandsquadrat
```

8.3 eigene Infix-Operatoren

Wenn man eigene Funktionen schreibt, werden immer nur die Operanden verwendet, die hinter dem Funktionsnamen stehen; das ist aber nicht immer erwünscht – besonders bei zweistelligen Funktionen, die eine Operator-ähnliche Funktionalität aufweisen, wäre eine Infix-Schreibweise erheblich angenehmer. Nun, nichts leichter als das: Um aus einer zweistelligen Funktion einen Infix-Operator zu machen, schließt man den Funktionsnamen einfach in ```-Akzente ein:

```
-- Skalarprodukt zweier Vektoren berechnen
skp:: (Float,Float) -> (Float,Float) -> Float
skp (a1,a2) (b1,b2) = a1*b1 + a2*b2

-- Länge eines Vektors berechnen
laenge:: (Float,Float) -> Float
laenge a = sqrt (a `skp` a)
```

berechnet die Länge eines 2-dim. Vektors mit der bekannten Formel $|a| = \sqrt{a \bullet a}$

Kapitel 9

eigene Datentypen

Bei eigenen Datentypen sollte man zuerst zwischen zwei unterschiedlichen Phänomenen unterscheiden: zum Einen einem "echten" neuen Datentyp (der in Haskell durch Angabe eines Konstruktors definiert wird, s.u.) und einem neuen Datentyp, der nur einen neuen Namen besitzt, aber eigentlich schon vorher existierte. So etwas bezeichnet man als Typ-Synonym. Unabhängig von dieser Unterscheidung muß der Name eines eigenen Datentyps aber auf jeden Fall mit einem Großbuchstaben beginnen, das sollte man dabei immer im Hinterkopf behalten.

9.1 Typ-Synonyme

Wenn man für einen schon vorhandenen Typ ein neues *Synonym* einführen möchte, um z.B. die Verwendung klarer zu machen, so bedient man sich des *type*-Befehls in der folgenden Art und Weise:

```
type neuerTyp = bekannterTyp
```

Dabei kann der bekannte Typ entweder ein "einfacher", schon in Haskell vorhandener Typ sein, oder aber einer der komplizierteren, wie z.B. ein Tupel oder eine Liste:

```
type Zahl = Float
type Koordinaten = (Float,Float)

-- Abstand zweier Punkte voneinander bestimmen
abstand:: Koordinaten -> Koordinaten -> Zahl
abstand (a,b) (c,d) = sqrt((a-c)^2 + (b-d)^2)
```

Hier werden Typ-Synonyme verwendet, um neue Koordinatentypen "Zahl" und "Koordinaten" zu definieren, die im Programm verwendet werden, um den Abstand zweier Euklidischer Koordinaten zu bestimmen.

In analoger Weise, wie hier ein Koordinaten-Typ definiert wurde, kann natürlich auch eine Liste verwendet werden, um einen anderen gerade benötigten Typ zu definieren.

9.2 "echt-eigene" Datentypen

Um einen Datentyp zu definieren, der nicht nur eine Umbenennung eines schon vorhandenen Datentyps darstellt, ist es nötig, dem Haskell-Interpreter anzugeben, wie er die möglichen Werte des Typs generieren kann (damit er prüfen kann, ob ein gegebener Wert auch wirklich von diesem Typ ist). Dazu ist es nötig, bei der Definition des Typs einen sogenannten *Typ-Konstruktor* anzugeben,

was dann wie folgt aussieht:

```
data neuerTyp = Konstruktor [|alternativKonstr1|alternativKonstr2|...]
```

Man gibt also hinter dem Schlüsselwort *data* den Namen des neuen Typs an, und hinter dem folgenden Gleichheitszeichen die möglichen Konstruktoren für diesen Typ (ggf. getrennt durch einen senkrechten Strich).

Die zwei Arten von Konstruktoren, die in Haskell möglich sind, werden in den folgenden Abschnitten dargestellt.

9.3 eigene Datentypen durch Aufzählen

Aufzählen ist der einfachste Fall eines Konstruktors: hierbei werden einfach alle möglichen (jeweils "konstanten") Werte, die der Datentyp haben kann, angegeben (aufgezählt), und nur ein Wert, der auch in dieser Liste auftaucht, wird als gültig betrachtet. Dazu zunächst ein einfaches Beispiel:

```
data Boolesch = Wahr | Falsch

boolund :: Boolesch -> Boolesch -> String
boolund Wahr Wahr = "wahr"
boolund a b = "falsch"
```

Hier haben wir uns also unseren eigenen Booleschen Datentyp definiert, analog zum vorgegebenen `Bool`. Was man sich aus diesem Beispiel auf jeden Fall merken sollte: konstante Konstruktoren sind immer groß zu schreiben (wie hier "Wahr" und "Falsch").

Auf diese Art könnte man in einem komplexeren Beispiel einen Typ definieren, der durch mehrere Konstruktoren verschiedene Farbwerte aufnehmen kann (und die Funktion mischt jeweils zwei Farben):

```
data Farbe = Rot | Gelb | Blau | Gruen | Lila | Orange | Weiss | Schwarz

mischen :: Farbe -> Farbe -> String
mischen Rot Gelb = "Orange"
mischen Gelb Rot = "Orange"
mischen Rot Blau = "Lila"
mischen Blau Rot = "Lila"
mischen Blau Gelb = "Grün"
mischen Gelb Blau = "Grün"
mischen a b = "weiß ich leider auch nicht"
```

Durch Aufzählen ist der Größe eines Datentyps praktisch aber eine starke Größenbegrenzung auferlegt. Wie man auf einfache Art einen größeren Wertebereich verwenden kann, wollen wir uns im nächsten Abschnitt ansehen.

9.4 rekursive Datentypen

Und schließlich kann man zur Definition eines eigenen Datentyps auf *Rekursion* zurückgreifen, d.h. mind. einer der Typ-Konstruktoren, aber auf keinen Fall alle, enthält wiederum einen Verweis auf den neuen Typ selbst. Auf diese Art ist es u.a. möglich, einen Datentyp zu definieren, der einen binären Baum aufnehmen kann, in dessen Knoten jeweils eine Integer-Zahl eingetragen ist:

```
data Baum = Blatt Int | Zweig Baum Baum

-- Auflisten der Elemente eines Baumes
listen:: Baum -> [Int]
listen (Blatt a) = [a]
listen (Zweig a b) = (listen a) ++ (listen b)
```

Dieses Programm liefert für die Eingabe

```
listen (Zweig (Zweig (Blatt 2) (Blatt 1)) (Blatt 3))
```

die Ausgabe

```
[2,1,3]
```

also genau die Präfix-Darstellung der Zahlen, die in dem Baum gespeichert waren.

Kapitel 10

Funktionen höherer Ordnung

Funktionen höherer Ordnung, auch als Funktionen auf Funktionen bezeichnet, ermöglichen es, Funktionen als Parameter zu übergeben, und diese Funktionen zur Berechnung eines Wertes heranzuziehen; dadurch wird dem Programmierer ein mächtiges Werkzeug an die Hand gegeben.

Einige Funktionen höherer Ordnung sind schon von Haskell vordefiniert, deshalb sollen (zur Illustration) zuerst einmal diese vorgestellt werden.

10.1 map

map wendet die übergebene Funktion auf jedes einzelne Element der Liste an, und bildet aus den so berechneten Elementen die Ergebnisliste, die zurückgegeben wird:

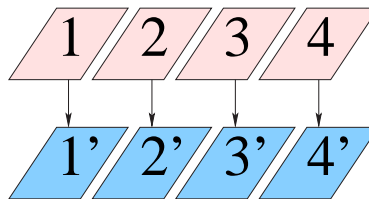


Abbildung 10.1: Funktion von map

Der Aufruf der map-Funktion (Signatur: $(a \rightarrow b) \rightarrow [a] \rightarrow [b]$) erfolgt so:

```
map (Berechnung) Liste
```

Diese Funktion kann man z.B. verwenden, um alle Elemente einer Integer-Liste zu verdoppeln:

```
-- alle Elemente verdoppeln
doppel :: [Int] -> [Int]
doppel liste = map (2*) liste
```

Anzumerken ist noch, daß die Berechnung auch beliebig komplizierter sein kann, solange sie auf die einzelnen Elemente der Liste angewendet werden kann.

10.2 foldl

foldl berechnet aus einer Liste von Elementen ein Ergebnis-Element, indem die Elemente der Liste nacheinander (beginnend mit einem zusätzlich angegebenen Anfangswert) mittels Operation verknüpft werden, die als Parameter übergeben wird; dabei wird mit den Elementen am Anfang

der Liste begonnen, und die Berechnung arbeitet sich nach hinten durch:

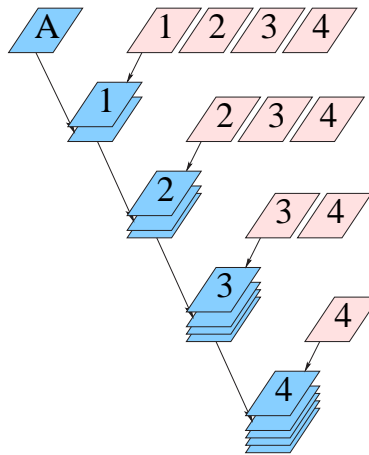


Abbildung 10.2: Funktion von foldl

Der Aufruf der foldl-Funktion (Signatur: $(a \rightarrow b \rightarrow a) \rightarrow a \rightarrow [b] \rightarrow a$) erfolgt so:

```
foldl (Operation) Anfangswert Liste
```

Die Operation muß sich dabei nicht unbedingt auf die vordefinierten beschränken, sondern man kann auch selbstdefinierte Funktionen verwenden - die beiden zu verknüpfenden Werte werden an diese Funktion als Parameter weitergereicht (und zwar der Anfangswert bzw. die berechneten Zwischenwerte als 1. Parameter, das aktuelle Listenelement als 2. Parameter). Genau von dieser Möglichkeit wird im folgenden Beispiel Gebrauch gemacht, das eine alternative Formulierung für das Horner-Schema (siehe auch Kapitel 3.2) bietet:

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

hornerhilf :: Int -> Int -> Int -> Int
hornerhilf x a b = a*x + b

horner :: Int -> Int
horner x = foldl (hornerhilf x) 0 koef
```

die (quadratische) Funktion wird durch die Hilfsfunktion koef definiert, die die Koeffizienten der Funktion als Liste angibt

Der binäre Operator, der angegeben wird, muß im übrigen nicht homogen sein, d.h. der Typ des linken Operanden muß nicht unbedingt mit dem des rechten Operanden übereinstimmen - nur das Ergebnis muß vom gleichen Typ sein wie der linke Operand.

10.3 foldr

foldr arbeitet analog zu *foldl*, nur daß die Liste nicht von vorne nach hinten durchgearbeitet wird, sondern mit dem letzten Element beginnend nach vorne durchlaufen wird.

Als Beispiel könnte man das eben schon betrachtete Horner-Schema verwenden, wenn man die Koeffizienten in umgekehrter Reihenfolge ordnet (da die Liste umgekehrt durchlaufen wird), in der Hilfsfunktion die Verwendung der beiden Parameter *a* und *b* vertauscht (weil bei *foldr* der berechnete Zwischenwert und das aktuelle Element in umgekehrter Reihenfolge wie bei *foldl* übergeben werden - was insbesondere bei Operatoren, die nicht homogen sind (s.o.) sehr schnell

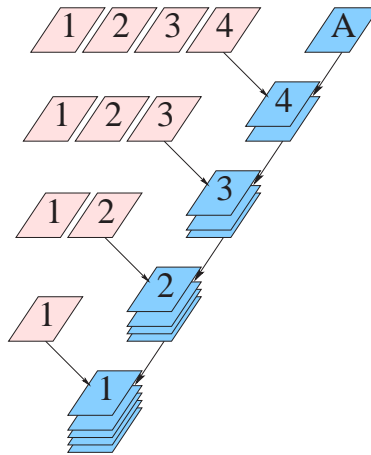


Abbildung 10.3: Funktion von foldr

zu Typfehlern führen kann), und natürlich foldl durch foldr ersetzt:

```

koeff :: [Int]
koeff = [3,2,1]

hornerhilf :: Int -> Int -> Int -> Int
hornerhilf x b a = a*x + b

horner :: Int -> Int
horner x = foldr (hornerhilf x) 0 koeff
    
```

die (quadratische) Funktion wird durch die Hilfsfunktion koeff definiert, die die Koeffizienten der Funktion als Liste angibt

10.4 filter

filter wendet eine angegebene Bool'sche Funktion auf jedes Element der übergebenen Liste an, und prüft das Ergebnis; ist es True, so wird das Element an die zu erstellende Ergebnisliste angehängt, bei False nicht.

Dies könnte man verwenden, um aus einer Liste von Zahlen alle negativen Werte zu entfernen:

```

ohnenegative :: [Int] -> [Int]
ohnenegative a = filter (>=0) a
    
```

10.5 zip

zip übernimmt zwei Listen als Parameter, und bildet die Ergebnisliste dadurch, daß jeweils ein Element aus jeder der beiden Listen zu einem Tupel zusammengefügt werden, und ans Ende der Ergebnisliste kommen. Der Aufruf der zip-Funktion (Signatur: [a] -> [b] -> [(a,b)]) erfolgt so:

```
zip Liste1 Liste2
```

Ein Beispiel für die Verwendung dieser Funktion ist in Kapitel 12 (im Zusammenhang mit Lambda-Abstraktionen) zu finden.

Öfter wird man wohl die folgende, mit zip verwandte Funktion verwenden:

10.6 zipWith

zipWith übernimmt zwei Listen als Parameter, sowie eine Funktion, die auf den Typen der Listen arbeiten kann, und errechnet eine neue Liste, indem jeweils das nächstaktuelle Element jeder der beiden Listen mit der angegebenen Operation verknüpft wird, und an den Anfang der Ergebnisliste gehängt wird:

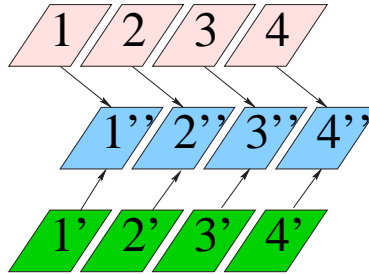


Abbildung 10.4: Funktion von zip

Der Aufruf der zipWith-Funktion (Signatur: $(a \rightarrow b \rightarrow c) \rightarrow [a] \rightarrow [b] \rightarrow [c]$) erfolgt so:

```
zipWith (Operation) Liste1 Liste2
```

Mit dieser Funktion (und etwas zusätzlicher Hilfe von foldl) ist es sehr einfach, das Standard-Skalarprodukt beliebig langer Integer-Listen zu berechnen:

```
-- Skalarprodukt beliebiger Listen
skp :: [Int] -> [Int] -> Int
skp a b = foldl (+) 0 (zipWith (*) a b)
```

Kapitel 11

Typklassen

Beim Erstellen von Haskell-Programmen mit vielen selbst-definierten Typen wird man vielleicht (oder auch nicht?!) darauf stoßen, daß viele Funktionen gleichen Namens (gleicher Signatur und evtl. sogar gleicher Implementierung) für jeden der selbst-definierten Typen implementiert wurden (z.B. Ausgabe eines Wertes von diesen Typen); da ein derartiger Aufwand natürlich unerträglich ist, sollte man Abhilfe schaffen: und eben diese Abhilfe ist in Form von Typklassen das Thema dieses Kapitels.

11.1 Was sind Typklassen überhaupt?

Typklassen kapseln (d.h. fassen zusammen) Signaturen, die für eine (wie der Name schon sagt) Klasse von Datentypen typisch sind; das können z.B. Ausgabefunktionen sein, die aus dem "internen Format" einen String erzeugen, aber auch viele Operatoren fallen in diese Klasse: z.B. ist es ja möglich, mit jedem der vielen numerischen Datentypen (Int, Integer, Float, ...) (+) durchzuführen, ebenso wie die anderen Rechenarten und Vergleiche wie (j). Das alles funktioniert deshalb, weil den entsprechenden Datentypen die passenden Typklassen zugeordnet werden, und der Programmierer des Datentyps muß dann die entsprechenden Funktionen implementieren.

Darüber hinaus können aber auch Teile dieser Funktionen schon implementiert werden; wenn man z.B. sowohl (==) als auch (j) schon implementiert hat, ist es nur noch ein kleiner Schritt zum (j=) zu kommen, und vor allem ist dieser Schritt nur davon abhängig, daß die beiden anderen Operatoren funktionieren. Also könnte man bei einer Typklasse, die die 3 Operatoren (==), (j), (j=) umfaßt, die Implementierung von (j=) sogar schon vornehmen, ohne zu wissen, wie der zugehörige Datentyp aussieht.

All das wird durch Typklassen ermöglicht; darüber hinaus bieten Typklassen auch noch das Konzept der Vererbung an, d.h. eine Typklasse A kann von einer oder mehreren anderen (B, C) *erben*, d.h. ohne die Signaturen der geerbten Klassen noch einmal angeben zu müssen, muß der Datentyp, dem A zugeordnet wird, auch die Signaturen von B und C mit Leben füllen

Doch genug der langen und langweiligen Vorreden, befassen wir uns lieber einmal damit, wie eine Typklasse überhaupt definiert wird, bevor wir uns dann die Verwendung ansehen.

11.2 Definition eigener Typklassen

Wie definiert man eine Typklasse? Nun, ganz einfach: eine Typklasse benötigt natürlich zuerst einmal einen Namen, danach folgt ein Platzhaltername, der in den Signaturen verwendet werden kann, und dann noch eine (beliebige) Anzahl von Signaturen (die eingerückt werden, damit Haskell erkennen kann, wo die Definition der Klasse zuende ist), was dann also etwa so aussieht:

```
class Name Platzhalter where
    Liste von Signaturen
```

Wenn dabei in den Signaturen der Typ, dem die Klasse zugewiesen wurde, verwendet werden soll, benutzt man einfach den Platzhalternamen, etwa so wie im folgenden Beispiel:

```
class Mischmal m where
    mischen :: m -> m -> m
    ismischbar :: m -> m -> Bool
    analyse :: m -> [m]
    anzeigen :: m -> String
```

Wie weiter oben schon erwähnt, dürfen die Funktionen auch gleich implementiert werden, falls gewünscht - was natürlich nur dann möglich ist, wenn die Implementierung keine speziellen Eigenschaften irgendwelcher Datentypen verwendet. Auf diese Art könnte man die obige Klasse z.B. noch um eine Anzeige-Funktion für Listen ergänzen:

```
class Mischmal m where
    mischen :: m -> m -> m
    ismischbar :: m -> m -> Bool
    analyse :: m -> [m]
    anzeigen :: m -> String
    anzeigenL :: [m] -> [String]
    anzeigenL [] = []
    anzeigenL (x:xs) = (anzeigen x):(anzeigenL xs)
```

Hier wird also eine Klasse definiert, zu der drei Funktionen gehören: eine Funktion `mischen`, die zwei Elemente vom entsprechenden Datentyp annimmt und ein solches Element wieder zurückliefert, eine Funktion die stattdessen einen Booleschen Wahrheitswert zurückgibt, und eine die aus einem Element eine Liste von entsprechenden Elementen erzeugen kann.

Nun wollen wir uns darum kümmern, was diese ganze Definiererei überhaupt bringen soll.

11.3 direkte Verwendung von Typ-Klassen

Wenn man einen Datentyp als zu einer Klasse zugehörig bezeichnen will, nennt man das auch "der Datentyp instanziiert die Klasse", und entsprechend heißt auch der entsprechende Befehl:

```
instance Klasse Datentyp where
    Implementierung der Signaturen
```

Man beachte dabei wieder die Einrückung der Implementierungen, um das Ende der Instanzierung zu erkennen.

Das wollen wir uns nun mit unserer oben definierten Klasse, und zwei eigenen Datentypen ansehen (für die bessere Lesbarkeit fehlt die Klassendefinition im folgenden Kasten):

```

data Farbe = Rot | Gelb | Blau | Gruen | Lila | Orange | Weiss | Schwarz

instance Mischmal Farbe where

    mischen Rot Gelb = Orange
    mischen Gelb Rot = Orange
    mischen Rot Blau = Lila
    mischen Blau Rot = Lila
    mischen Blau Gelb = Gruen
    mischen Gelb Blau = Gruen
    mischen a b      = Schwarz

    ismischbar Rot Gelb = True
    ismischbar Gelb Rot = True
    ismischbar Rot Blau = True
    ismischbar Blau Rot = True
    ismischbar Blau Gelb = True
    ismischbar Gelb Blau = True
    ismischbar a b      = False

    analyse Orange = [Gelb, Rot]
    analyse Lila   = [Rot, Blau]
    analyse Gruen  = [Gelb, Blau]
    analyse a      = []

    anzeigen Rot    = "rot"
    anzeigen Gelb   = "gelb"
    anzeigen Blau   = "blau"
    anzeigen Gruen  = "grün"
    anzeigen Lila   = "lila"
    anzeigen Orange = "orange"
    anzeigen Weiss  = "weiß"
    anzeigen Schwarz = "schwarz"

```

Man könnte die Implementierung nun z.B. mit der Eingabe

```
anzeigenL (analyse Orange)
```

testen, und wirklich, die Ausgabe ist:

```
["gelb", "rot"]
```

wie es zu erwarten war.

Hier gleich noch ein zweites Beispiel zum Vergleich; das "deriving Eq" gleich zu Anfang kann vorerst überlesen werden, dazu werden wir im nächsten Abschnitt kommen. Vorerst nur soviel: es ermöglicht den "=="-Vergleich bei der `ismischbar`-Funktion:

```

data Alk = Wodka | Bacardi | Gin | Lemon | Cola | OSaft | WodkaLemon |
        CubaLibre | Gin0 | Bac0 | Ungeniessbar deriving Eq

instance Mischmal Alk where

    mischen Wodka Lemon = WodkaLemon
    mischen Lemon Wodka = WodkaLemon
    mischen Bacardi Cola = CubaLibre
    mischen Cola Bacardi = CubaLibre
    mischen Bacardi OSaft = Bac0
    mischen OSaft Bacardi = Bac0
    mischen Gin OSaft = Gin0
    mischen OSaft Gin = Gin0
    mischen a b = Ungeniessbar

    ismischbar a b | mischen a b == Ungeniessbar = False
                   | otherwise = True

    analyse WodkaLemon = [Wodka, Lemon]
    analyse CubaLibre = [Bacardi, Cola]
    analyse Gin0 = [Gin, OSaft]
    analyse Bac0 = [Bacardi, OSaft]
    analyse Ungeniessbar = []
    analyse a = [a]

    anzeigen Wodka = "Wodka"
    anzeigen Bacardi = "Bacardi"
    anzeigen Gin = "Gin"
    anzeigen Lemon = "Bitter Lemon"
    anzeigen Cola = "Cola"
    anzeigen OSaft = "Orangensaft"
    anzeigen WodkaLemon = "Wodka-Lemon"
    anzeigen CubaLibre = "Cuba Libre"
    anzeigen Gin0 = "Gin-Orange"
    anzeigen Bac0 = "Bacardi-Orange"
    anzeigen Ungeniessbar = "erst ab 3 Promille geniessbar"

```

11.4 indirekte Verwendung vordefinierter Typklassen

Auch Haskell liefert schon einige besonders wichtige und häufig verwendete Typklassen mit, die man auch für eigene Datentypen verwenden kann. Dazu gehört u.a. die Typklasse `Eq`, die im letzten Beispiel schon verwendet wurde: sie bietet den Vergleich auf Gleichheit zwischen 2 Elementen des Datentyps, der die `Eq`-Klasse instanziiert.

Wie man aber auch am obigen Beispiel sehen kann, wird nicht unsere bisher übliche Art der Instanziierung (mittels `instance`) verwendet, sondern ein neuer Befehl:

```
Datentyp-Definition deriving (Typklassen)
```

Dieser Befehl läßt sich nur mit den von Haskell vordefinierten Typklassen verwenden, für die in diesem Fall eine Art "Standard-Implementierung" vorgenommen wird (bei `(==)` z.B. die Prüfung, ob die Konstruktoren der beiden Elemente identisch, d.h. zeichenweise gleich sind - was etwa bei Brüchen, die noch nicht vollständig gekürzt sind, nicht unbedingt das gewünschte Ergebnis liefert).

Bei einfachen Datentypen ist es auf diese Art also ziemlich einfach, eine relativ große Funktionalität zu gewinnen, ohne allzu großen Aufwand in die Implementierung stecken zu müssen; die am häufigsten verwendeten Klassen dürften die folgenden sein:

11.4.1 Show

Haskell verwendet zum Anzeigen des "letzten" Ergebnisses eines Aufrufs implizit einen Aufruf der show-Funktion, die aus einem beliebigen Datentyp einen String erzeugt - "deriving Show" stellt also die einfachste Art da, Haskell zum Anzeigen eigener Datentypen zu bewegen.

11.4.2 Eq

Eq liefert die Möglichkeit, zwei Elemente gleichen Typs mittels (==) zu vergleichen - wie oben beschrieben.

Durch den Einsatz von Eq (was wir eben schon getan haben), und zusätzlich noch Show, läßt sich das obige Beispiel (unter leichter Veränderung der Funktion) noch weiter vereinfachen:

```
data Alk = Wodka | Bacardi | Gin | Lemon | Cola | OSaft | WodkaLemon |
        CubaLibre | Gin0 | Bac0 | Ungeniessbar deriving (Eq, Show)

instance Mischmal Alk where

    mischen Wodka Lemon = WodkaLemon
    mischen Lemon Wodka = WodkaLemon
    mischen Bacardi Cola = CubaLibre
    mischen Cola Bacardi = CubaLibre
    mischen Bacardi OSaft = Bac0
    mischen OSaft Bacardi = Bac0
    mischen Gin OSaft = Gin0
    mischen OSaft Gin = Gin0
    mischen a b = Ungeniessbar

    ismischbar a b | mischen a b == Ungeniessbar = False
                   | otherwise = True

    analyse WodkaLemon = [Wodka, Lemon]
    analyse CubaLibre = [Bacardi, Cola]
    analyse Gin0 = [Gin, OSaft]
    analyse Bac0 = [Bacardi, OSaft]
    analyse Ungeniessbar = []
    analyse a = [a]

    anzeigen a = show a
```

11.4.3 Ord

Die Klasse Ord liefert (<), (<=), (>) und (>=) - bei endlichen Datentypen dadurch, daß die Elemente nach Reihenfolge ihres Auftauchens in der Typdefinition geordnet werden.

11.5 Anwendung von Typklassen

Kommen wir nun zu einem weiteren Vorteil von Typklassen: sie ermöglichen es, erheblich allgemeinere Funktionen zu schreiben, als wenn man immer feste Datentypen in der Signatur angibt

- schließlich reicht es manchmal schon aus zu wissen, daß sich die Elemente dieses Datentyps vergleichen lassen - mithin also die in der Klasse Eq oder Ord definierten Funktionen besitzt.

Eben dies ist möglich, indem man in der Signatur mittels "=>" angibt, daß der folgende (Platzhalter-)Datentyp von einer bestimmten Klasse ist, und zwar in folgender Art und Weise:

```
Funktion:: (Klasse Platzhalter, ...) => Parameter
```

Wie so oft in Haskell, kann man entweder nur eine Klasse mit Platzhalter angeben (dann auch ohne die Klammern), oder eine Liste von Klassen mit (gleichen oder unterschiedlichen) Platzhaltern, die von der eigentlichen Signatur gefolgt werden (in der die Platzhalter dann wie normale Datentypen verwendet werden können).

Als Beispiel wollen wir uns eine Funktion ansehen, die das Mischen unserer obigen Klasse nur durchführt, wenn es auch zu einem sinnvollen Ergebnis führt, ansonsten aber eine Liste mit den "Zutaten" ausgibt:

```
savemisch:: Mischmal m => m -> m -> [m]
savemisch a b | ismischbar a b = [mischen a b]
              | otherwise      = [a,b]
```

Diese Funktion kann man sowohl mit unseren Farben, als auch mit unseren Getränken ausprobieren, trotzdem funktioniert sie wie gewünscht.

Teil IV

das Beste zum Schluß ...

Kapitel 12

Lambda-Abstraktionen

Wer auch jetzt noch nicht genug hat: in Haskell können sogar Lambda-Ausdrücke für Berechnungen benutzt werden, die Notation erfolgt (sehr stark) angelehnt an die Lambda-Schreibweisen, wie man sie aus der Info-Vorlesung kennt:

```
add :: Int -> Int -> Int
add = \x y -> x+y
```

ist also gleichbedeutend mit der "normalen" Haskell-Funktion:

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

Die in dem Lambda-Ausdruck gebundene(n) Variable(n) steht also nach einem `\`, ggf. getrennt durch ein Leerzeichen, falls mehr als eine Variable gebunden werden soll; der Rumpf der Lambda-Abstraktion steht nach einem Pfeil der Form `->`.

Derartige Lambda-Funktionen sind z.B. besonders sinnvoll als *anonyme Funktionen* bei Funktionen höherer Ordnung einsetzbar, um besonders mächtige Berechnungen anzustellen:

```
skp :: [Int] -> [Int] -> Int
skp a b = foldl (\ s (x,y) -> s + (x*y)) 0 (zip a b)
```

Oder in anderer (hoffentlich verständlicherer) Form:

```
skp :: [Int] -> [Int] -> Int
skp a b = foldl (+) 0 prodliste
  where liste = zip a b
        paarprod = \ (x,y) -> x*y
        prodliste = map paarprod liste
```

Beide Programme berechnen das Standard-Skalarprodukt, indem die Elemente der beiden Listen paarweise multipliziert, und die Produkte addiert werden. Dazu werden die beiden Listen mittels `zip` zu einer Tupel-Liste zusammengefügt, deren Elemente dann von einer Lambda-Funktion multipliziert werden; die dadurch entstehende Liste wird dann einfach noch mittels `foldl` aufaddiert, et voilà: Wir 'aben das Standard-Skalar-Produkt.

Kapitel 13

Listen generieren

Haskell stellt auch einige Möglichkeiten bereit, Listen (sowohl endliche als auch (theoretisch) unendlich lange) automatisch erstellen zu lassen, wobei bestimmte Bedingungen für die Elemente, die in der Liste stehen sollen, berücksichtigt werden. In diesem Kapitel wollen wir uns diese verschiedenen Generierungsmöglichkeiten aus der Nähe ansehen.

13.1 Aufzählungen

Aufzählungen werden ähnlich wie in der Mathematik gebildet: wenn für die Elemente der zu generierenden Liste jeweils eindeutige Nachfolger existieren, kann man so eine endliche Liste bilden:

```
[erstes Element...letztes Element]
```

Diese Möglichkeit kann man z.B. gut verwenden, um längere Listen als Testeingaben für irgendwelche Programme zu verwenden!

Auch unendliche Listen können auf diese Art generiert werden, indem man einfach die 2. Grenze ausläßt:

```
[erstes Element...]
```

In diesem Fall sollte es aber natürlich möglich sein, die Liste irgendwie wieder auf eine handliche Länge herunterzubrechen; zu eben diesem Zweck wird dann der *take*-Befehl verwendet, der eine Zahl n und eine Liste als Parameter übernimmt, und die ersten n Elemente der Liste als Ergebnis liefert:

```
take :: Int -> [a] -> [a]
```

Hier nun die beiden Aufzählungsmöglichkeiten in einem Beispiel, beide liefern jeweils die ersten 13 natürlichen Zahlen:

```
get13a = [1..13]
get13b = take 13 [1..]
```

Im übrigen lassen sich Buchstaben genauso wie Zahlen aufzählen!

13.2 Listengeneratoren

Ein Listengenerator arbeitet ganz genau wie oben beschrieben: man gibt Haskell die Anweisung, eine Liste zu bilden aus allen Elementen, die einer Anzahl von Bedingungen genügen; die Syntax sieht dabei folgendermaßen aus:

```
[var | var < -Grundmenge für var, Bedingungen für var]
```

Sollen dabei mehrere Bedingungen erfüllt werden, so müssen diese durch Komma getrennt sein. Mit Hilfe dieses Konstruktes könnte man z.B. eine Liste der Primzahlen $\leq n$ berechnet werden:

```
primzahlen:: Int -> [Int]
primzahlen n = [x|x<-[2..n],prim x]
               where prim x = teiler x == []
                     teiler n = [x | x<-[2..(n-1)],(mod n x)==0]
```

Das Programm arbeitet ohne Betrachten von Randfällen, und zwar wie folgt:

1. geliefert wird eine Teilliste der Zahlen von 2 bis n, und zwar nur die Elemente, die prim sind;
2. prim sind die Elemente, deren Teilmengen leer ist,
3. und die Teilmengen von k ist die Teilmenge der Zahlen von 2 bis k-1, die k mit Rest 0 teilen.

13.3 iterierte Funktionen

Schließlich gibt es die Möglichkeit, eine Liste dadurch zu generieren, daß eine vorgegebene Funktion wiederholt auf einen Wert angewandt wird, und jede weitere Funktionsanwendung liefert ein weiteres Element der Liste; dieses Vorgehen wird durch die *iterate*-Anweisung möglich, die folgende Syntax aufweist:

```
iterate Funktion Startwert
```

Da dabei wiederum eine im Prinzip unendliche Liste erzeugt wird, muß die benötigte Zahl an Elementen durch den Einsatz von *take* entnommen werden:

```
ittest:: [Float]
ittest = iterate (/(-1.20)) 100

itn:: Int -> [Float]
itn n = take n ittest
```

Glieder einer alternierenden
Nullfolge berechnen

Wenn man statt einfacher Funktionsdefinitionen (wie oben) Lambda-Funktionen einsetzt, ist es natürlich möglich, deutlich kompliziertere Berechnungen zu realisieren; als Beispiel soll die folgende Generierung von Fibonacci-Zahlen dienen:

```
fib:: [(Integer,Integer)]
fib = iterate (\(x,y) -> (y,(x+y))) (1,1)

fibshow:: Int -> [Integer]
fibshow n = map (fst) (take n fib)
```

Die generierte Liste besteht also aus 2-Tupeln, wobei jeweils das 2. Glied eines Elementes der Liste gleich dem 1. Glied des nächsten Elements ist; ein neues Element wird also dadurch

berechnet, daß die Summe der beiden Glieder seines VorgängerS berechnet und als neues Element an die Ergebnisliste gehängt wird.

Für die Ausgabe werden nun einfach n Elemente vom Anfang der Liste abgegriffen, und von jedem Element nur noch das 1. Glied ausgegeben.

13.4 Element-Wiederholungen

Eine letzte Möglichkeit zur Erzeugung von Listen besteht darin, ein bestimmtes (vorgegebenes) Element mehrfach (in endlicher Anzahl oder unendlich oft) zu wiederholen; das wird mit den Befehlen *repeat* und *replicate* verwirklicht:

```
repeat :: a -> [a]
replicate :: Int -> a -> [a]
```

Die beiden Befehle liefern folgende Ergebnisse:

```
halorepeat :: Int -> [String]
halorepeat i = take i (repeat "Hallo")

halorepl :: Int -> [String]
halorepl i = replicate i "Hallo"
```

Beide Funktionen liefern i -mal "Hallo", wobei bei *repeat* natürlich (wie bei unendlichen Listen üblich) der *take*-Befehl zum Einsatz kommt. (im übrigen ist *replicate* intern ähnlich wie *halorepeat* implementiert)

Anhang A

Listen-Operationen

Hier eine vollständigere Liste der Operationen, die für Listen in der Prelude schon vordefiniert sind:

```
(:) :: a -> [a] -> [a]
    Element vorne an eine Liste anhängen

(++) :: [a] -> [a] -> [a]
    zwei Listen aneinander hängen

 (!! ) :: [a] -> Int -> a
    auf bestimmtes Element indiziert zugreifen (Zählung ab 0!)
```

```
concat :: [[a]] -> [a]
    Liste von Listen zu einer Liste zusammenfassen

drop :: Int -> [a] -> [a]
    die ersten n Elemente der Liste entfernen

elem :: Eq a => a -> [a] -> Bool
    prüfen, ob ein angegebenes Element in der Liste enthalten ist

head :: [a] -> a
    erstes Element der Liste liefern

init :: [a] -> [a]
    letztes Element der Liste entfernen

last :: [a] -> a
    letztes Element der Liste liefern

length :: [a] -> Int
    Länge der Liste liefern (Anzahl der Elemente)

null :: [a] -> Bool
    prüfen, ob die Liste leer ist

replicate :: Int -> a -> [a]
    gegebenes Element n-mal replizieren

reverse :: [a] -> [a]
    Reihenfolge der Elemente einer Liste umkehren

tail :: [a] -> [a]
    erstes Element entfernen

take :: Int -> [a] -> [a]
    die ersten n Elemente liefern
```

Anhang B

Funktionen höherer Ordnung

`dropwhile :: (a -> Bool) -> [a] -> [a]`

vorn vorne beginnend so lange Elemente der Liste auslassen, solange die Bedingungs-
funktion *true* ergibt, dann die Restliste zurückgeben

`filter :: (a -> Bool) -> [a] -> [a]`

Ergebnisliste besteht aus den Elementen, für die die Bedingungsfunktion *true* ergibt

`foldl :: (a -> b -> a) -> a -> [b] -> a`

mit zweistelliger Funktion (beginnend mit dem übergebenen Startwert) linksassoziativ
die Liste zusammenrechnen

`foldr :: (a -> b -> b) -> b -> [a] -> b`

mit zweistelliger Funktion (beginnend mit dem übergebenen Startwert) rechtsassoziativ
die Liste zusammenrechnen

`groupBy :: (a -> a -> Bool) -> [a] -> [[a]]`

teilt eine Liste in Teillisten, innerhalb denen die Boolesche Funktion *true* liefert

`map :: (a -> b) -> [a] -> [b]`

übergebene Funktion auf jedes einzelne Listenelement anwenden

`takewhile :: (a -> Bool) -> [a] -> [a]`

Ergebnisliste besteht aus dem Anfangselementen der Parameterliste, soweit die
Bedingungsfunktion *true* ergeben hat

`until :: (a -> Bool) -> (a -> a) -> a -> a`

Funktion solange (wiederholt) auf den Startwert anwenden, bis die Bedingungsfunktion
true liefert

`zip :: [a] -> [b] -> [(a,b)]`

aus 2 Listen eine 2-Tupel-Liste machen

`zipWith :: (a -> b -> c) -> [a] -> [b] -> [c]`

aus 2 Listen mittels zweistelliger Funktion (die paarweise angewendet wird) eine
Ergebnisliste berechnen

Anhang C

vordefinierte Typ-Klassen

In diesem Anhang sollen die wichtigsten vordefinierten Typklassen vorgestellt werden, und welche Funktionalität sie jeweils anbieten können.

C.1 Enum

C.2 Eq

C.3 Num

C.4 Ord

C.5 Show

Literaturverzeichnis

- [1] SIMON PEYTON JONES (EDITOR) *Haskell 98 Language and Libraries - The Revised Report*
Die vollständige Dokumentation zu Haskell von den Machern - komplett mit vollständigem Abdruck und Erklärung der Prelude.hs, als PDF verfügbar über www.haskell.org.
- [2] PAUL HUDAK, JOHN PETERSON *A Gentle Introduction to Haskell 98*, University of California, 1999
Das (zumindest in Karlsruhe) weitverbreitetste Machwerk zum Thema Haskell, IMHO aber sehr wirr was die Reihenfolge der Erklärungen angeht, außerdem äußerst unvollständig.

Index

- !!, 38
- ++, 38
- ;, 38
- =;, 32

- Aggregattyp, 16
- Aufzählen, 21
- Aufzählungen, 35

- Boolesche Werte, 14

- Char, 13
- concat, 38

- data, 21
- Dezimalpunkt, 5
- Double, 13
- drop, 38
- dropwhile, 39

- elem, 38
- erben, 27

- False, 14
- filter, 25, 39
- Float, 13
- foldl, 23, 39
- foldr, 24, 39
- fst, 36
- Funktion, 6
 - anonyme, 34
 - höherer Ordnung, 23, 34
 - dropwhile, 39
 - filter, 25, 39
 - foldl, 23, 39
 - foldr, 24, 39
 - groupby, 39
 - map, 23, 39
 - takewhile, 39
 - until, 39
 - zip, 25, 39
 - zipWith, 26, 39

- Großschreibung, 7
- groupby, 39

- head, 38

- Hilfsfunktionen, 8
- homogen, 24
- homogener Operator, 24
- Horner-Schema, 10, 24

- imperative Programmierung, 5
- Infix-Operator, 19
- init, 38
- Instanz, 28
- Int, 13
- Integer, 13
- iterate, 36

- Kapselung, 27
- Kleinschreibung, 7
- Kommazahlen, 5
- Kommentar, 5
 - Block, 5
 - Zeile, 5
- Konkatenation, 13
- Konstruktor, 20
 - Aufzählen, 21
 - Rekursion, 21
 - Schreibweise, 21

- Lambda-Ausdruck, 34
- last, 38
- length, 38
- let, 18
- Liste
 - !!, 38
 - ++, 38
 - ;, 38
 - Anfang, 9
 - concat, 38
 - Definition, 9
 - drop, 38
 - elem, 38
 - head, 38
 - init, 38
 - last, 38
 - leer, 9
 - length, 38
 - null, 38
 - repeat, 37

- replicate, 37, 38
 - reverse, 38
 - tail, 38
 - take, 38
 - verketten, 10
 - Zugriff, 9
- map, 23, 39
- Mustererkennung, 6, 11
- null, 38
- otherwise, 11
- Parameter, 6
 - formaler, 6
- Pattern Matching, 11
- Rekursion, 7
- repeat, 37
- replicate, 37, 38
- reverse, 38
- Signatur, 6
- Skalarprodukt, 19, 26, 34
- String, 13
- Synonym, 20
- tail, 38
- take, 35, 36, 38
- takewhile, 39
- True, 14
- Tupel, 16
- Typ-Konstruktor, 20
- type, 20
- Typklassen, 27
 - Definition, 27
 - deriving, 30
 - Eq, 31
 - Implementierung, 28
 - Instanziierung, 28
 - Ord, 31
 - Show, 31
 - Signaturen, 27
 - vordefinierte, 30
- Unterstrich, 7
- until, 39
- Vererbung, 27
- Wahrheitswerte, 14
- where, 18
- zip, 25, 39
- zipWith, 26, 39