

Eine Einführung in Haskell

Dirk Braun

dirk.braun@fu-berlin.de

Version: 1.0.1

10. April 2009

Inhaltsverzeichnis

1	Einfache Funktionen	4
1.1	Parameter	4
1.2	Signaturen	5
1.3	Notizen	6
2	Datentypen	7
2.1	Listen	7
2.1.1	Listen konkatenieren	7
2.1.2	Selbstvervollständigende Listen	8
2.2	Tupel	8
2.3	Polymorphe Datentypen	9
3	Rekursionen	10
3.1	Guards	10
3.2	Guter Stil für Rekursionen	11
3.3	Fibonaccizahlen	11
4	Pattern-Matching und Lazy-Evaluation	12
4.1	Pattern-Matching bei Rekursionen	12
4.2	Lazy-Evaluation auf Listen	13
4.3	Lazy Evaluation auf unendliche Listen	14
5	List Comprehensions	15
5.1	Beispiel Das Sieb des Eratosthenes	15
5.2	Rekursive List Comprehensions	15
5.3	Sortieren	16
6	Eigene Datentypen	17
6.1	Lineare Datentypen	17
6.2	Rekursive Datentypen	18
7	Operationen auf Listen	20
7.1	map	20
7.2	zip	20
7.3	Faltung von Listen - foldl und foldr	20
7.4	Dynamische Programmierung	21

Einleitung

Dieser Text eine Sammlung der wichtigsten Konzepte und Programmiermittel für Haskell. Dabei werden sowohl die Datentypen in Haskell angesprochen, als auch die Paradigmen, die Haskell zu einer beliebten Lehrsprache für die Funktionale Programmierung machen.

Vorraussetzung

Vorraussetzung für diesen Text ist ein funktionierendes Haskell System. So sind sowohl als Interpreter **HUGS** oder der **Glasgow Haskell Compiler (GHC)** verfügbar. Jedoch gebe ich keine Unterstützung für bei Problemen mit diesen System.

Weitere Informationen

Ein weiterer Text über den *Bootstrap des GHC unter Mac OS X* ist ebenfalls auf www.26thmeusoc.com verfügbar.

1 Einfache Funktionen

Beginnen wir mit dem definieren einer simplen Funktion. Sie gibt uns nur die Worte „Hallo Welt“ aus.

Listing 1.1: chap1/hallowelt.hs

```
1 hallo = "Hallo Welt"
```

Laden wir die Datei nun in einen Haskell Interpreter, passiert noch nicht viel. Geben wir jedoch den Namen der Funktion (hallo) ein, so erhalten wir endlich die gewünschte Ausgabe. Es passiert hierbei jedoch nicht viel interessantes. Schreiben wir doch einfach mal eine Funktion, die zwei vorbereitete Zahlen addiert:

Listing 1.2: chap1/add.hs

```
1 addi = 2 + 3
```

Nachdem nun diese Funktion aufgerufen wird, erhalten wir das gewünschte Ergebnis 5. Im Gegensatz zum Beispiel 1.1 stehen um das Ergebnis jedoch keine Anführungszeichen. Der Grund liegt in den verschiedenen Datentypen der Ergebnisse. Zeichenketten (auch bekannt als String) werden immer in Anführungszeichen geschrieben, während Zahlenwerte (Integer) normalerweise nicht von Zeichen umschlossen werden.

1.1 Parameter

Bisher sind unsere Funktionen eher nutzlos, denn sie ergeben ja bei jeder Anwendung das gleiche Ergebnis aus. Viel interessanter wäre es daher, einer Funktionen einen oder mehrere Parameter zu übergeben, um dann damit etwas zu erledigen. Bauen wir also eine Funktion, die einen String annimmt, und diesen wieder ausgibt. Diese Art von Funktion nennt man *Identitätsfunktion*.

Listing 1.3: chap1/id.hs

```
1 ident x = x
```

Nun starten wir die Funktion indem wir, nach dem Laden der Datei, folgenden Aufruf tätigen:

```
> ident "Hi there"  
"Hi there"
```

Bei der Definition dieser Funktion haben wir uns jedoch nicht auf einen bestimmten Typ festgelegt. Wir können beispielsweise auch eine Zahl eingeben:

```
> ident 3
3
```

Warum das funktioniert, erklären wir gleich.

Interessanter wäre es aber doch, wenn wir etwas wirklich nützliches erledigen könnten wie etwa eine Addition. Daher definieren wir einfach eine Funktion, die dies für uns macht:

Listing 1.4: chap1/addition.hs

```
1 addi x y = x + y
```

Zum addieren der beiden Zahlen geben wir also beispielsweise folgendes ein:

```
> addi 4 9
13
```

1.2 Signaturen

In vielen Sprachen wird eine Funktion dadurch definiert, dass man die Typen für die Rückgabe sowie Parameter direkt festlegt. In C sieht das ungefähr so aus:

```
1 (int) addi (int a, int b)
```

Die Funktion nimmt also 2 Integer Werte an, und gibt einen zurück. In Haskell wird dies in einer optionalen Zeile, der sogenannten Signatur, erledigt. Dazu fügt man einfach eine Zeile hinzu, die folgenden Aufbau erfüllt:

```
<Funktionsname> :: <Parameter1>-><Parameter2>-><Rückgabewert>
```

Diese Zeile wird jedoch nicht benötigt, und dient in erster Linie eher dazu, Entwickler weitere Informationen über die Typen zu geben, die einer Funktion übergeben werden und zurückerhalten werden und ist daher in erster Linie freiwillig (später sehen wir einige Beispiele in dem das anders ist). Haskell versucht aber ansonsten selbst eine Signatur zu bilden. Um sie sich anzusehen gibt man im Interpreter folgendes an:

```
> :type <Funktionsname>
<Funktionsname> :: <Signatur>
```

Im Falle der Additionsfunktion sieht die Signatur also so aus:

```
1 addi :: Int -> Int -> Int
```

Die Hallo World Funktion gibt nur einen Wert zurück. Daher ergibt sich die entsprechende Signatur:

```
1 hallo :: String
```

1.3 Notizen

Um seinen Quelltext übersichtlicher zu gestalten, kann man auch in Haskell Notizen einführen. Einzeilige Notizen werden durch zwei Bindestriche eingeführt. Notizen die über mehrere Zeilen laufen sollen, werden jedoch von den folgenden Symbolen umschlossen: `{- ...-}`. Beispielhaft sehen wir uns den nächsten Quelltext mit den Beispielen an:

Listing 1.5: chap1/notizen.hs

```
1 {- Wirklich lange Notizen
2 hui, sogar ueber zwei Zeilen, wenn ich wollte ginge es noch
   laenger! -}
3 funk1 = "Eine Funktion"
4 -- Oh, nur eine Zeile, die naechste ist wieder ein Aufruf
5 funk2 = "Noch eine Funktion"
```

2 Datentypen

Als nächstes sehen wir uns die in Haskell vordefinierten Datentypen an. Zwei davon kennen wir ja bereits: *String* für Zeichenketten und *Int* für Zahlenwerte.

Name	Beispielhafte Werte	Bemerkungen
Bool	True , False	Boolesche Werte
Char	'a' , 'c' , ...	Einstellige Zeichen
String	"Hallo", "Viele Buchstaben"	Zeichenkette
Int	1, 27, 2147483647	Auf 32-bit begrenzte Zahlenwerte
Integer	1, 88, 18927978	Unendlich große Zahlenwerte

Wir haben also einiges an Auswahl mit der wir bereits einiges an Programmen entwickeln können.

2.1 Listen

Ein beliebtes Konzept in Programmiersprachen sind die sogenannten Arrays. Das sind Ansammlungen von Daten, die zusammengehören, aber aus verschiedenen Elementen bestehen und alle mit einmal übergeben werden, um sie zu verarbeiten. In Haskell gibt es dazu ein ähnliches Konstrukt: Listen.

Diese werden konstruiert, indem man einige, durch Kommas getrennte Werte in eckige Klammern ([und]) schreibt. Eine Liste der Zahlen von 1 bis 3 kann also folgendermaßen erstellt werden:

```
1 lbis3 = [1,2,3]
```

Dies kann sowohl dann sehr nützlich sein, wenn wir die Ergebnisse über die Zeit beobachten wollen, als auch dann, wenn eine unbestimmte Zahl von Werten übergeben werden soll.

2.1.1 Listen konkatenieren

Wir können in Haskell 2 Listen zusammenlegen. Diesen Vorgang nennt man **konkatenieren**. Es gibt zwei Möglichkeiten dafür:

1. Eine Liste wird an eine weitere Liste angehängen. Dies geschieht mit Hilfe des '++' Operators.

Listing 2.1: chap2/concat1.hs

```
1 listeconcat a b = a ++ b
2 beispiel = [1,2] ++ [3,4,5]
```

2. Ein Element wird an den Anfang der Liste gehangen:

Listing 2.2: chap2/concat2.hs

```
1 liste a b = a : b
2 beispiel = 1 : [2,3,4]
```

Zu beachten ist jedoch, dass bei besonders lange Listen, die erste Methode länger benötigt, als die zweite, da sie erst die gesamte Liste durchlaufen muss, bevor die Elemente angehängen werden können.

2.1.2 Selbstvervollständigende Listen

Stellen wir uns mal vor, wir wollen eine Liste der Zahlen von 0 bis 1000 erstellen. Mit unseren aktuellen Mitteln, haben wir eigentlich nur die Möglichkeit, dass wir alle Zahlen von 0 bis 1000 manuell in die Liste eingeben. Haskell hat dafür jedoch eine Automatisierung parat. Diese gilt ebenfalls für Buchstaben. Automatisierte Listen sehen folgendermaßen aus:

```
1 1bis1000 = [1..1000]
2 0bis32 = [0..32]
3 5bis10 = [5..10]
4 AbisZ = ['a'..'z']
```

Listen dürfen unendlich groß sein, das heißt, die Anzahl der enthaltenen Elemente ist also unerheblich. Listen können in Haskell sogar verschachtelt werden:

```
1 liste = [[1,2,3],[2,4,6],[1..10]]
```

Das einzige was nicht erlaubt ist, dass zwei verschiedene Datentypen in einer Liste zusammen auftauchen. Wir haben übringens bereits im ersten Kapitel mit einer Liste gearbeitet, ohne dass wir es bisher so genau wussten: Ein String ist nicht mehr als eine Liste von Char's! Es gilt daher:

```
['H','a','l','l','o'] = "Hallo"
```

2.2 Tupel

Um uns genau dieses Vermischen zweier Datentypen zu ermöglichen, können wir uns mit Tupeln helfen. Dazu definieren wir uns einfach einen solchen in unserem Skript. Dafür benötigen wir den **type**-Befehl. Ein Tupel mit dem Namen *UnserTupel* der aus einem String und einem Int Wert besteht wird folgendermaßen gebildet und angewendet:

Listing 2.3: chap2/tupeldef.hs

```
1 type UnserTupel = (String, Int)
2 unserTupelListe :: [UnserTupel]
3 unserTupelListe = [("Drei", 3), ("Zwei", 2), ("Einhundert", 100)]
```

2.3 Polymorphe Datentypen

Nicht immer ist jedoch der Datentyp so genau gegeben. Ein Beispiel ist die Identitätsfunktion aus dem Beispiel 1.3. Wie wir bereits festgestellt haben, können wir dort sowohl Strings und Int Werte eingeben, ohne einen Fehler zu erhalten. Wir haben also einen polymorphen (also vielfältigen) Datentyp in dieser Funktion. Um diese Eigenschaft beibehalten zu können, dürfen wir in der Signatur keinen Datentyp wie Int, String oder float angeben. Stattdessen schreiben wir dort einen Platzhalter hin.

Listing 2.4: chap2/id.hs

```
1 ident :: a -> a
2 ident x = x
```

Der Rückgabewert der Funktion `ident` ist also genau der gleiche, wie der Eingabetyp des ersten Parameters. Wollten wir beispielsweise eine Signatur für eine Funktion `z` konstruieren, deren zweiter Parameter einen anderen Typ als der erste hat, so verwenden dort einen anderen Platzhalter.

```
1 z :: a -> b -> a
```

3 Rekursionen

Kommen wir nun zu einem der wichtigsten Programmier-techniken, gerade in Haskell: Rekursive Funktion. In Haskell ist es nicht möglich, wie in den anderen Programmiersprachen, einfach seine Aufrufe nacheinander aufzurufen. Viel häufiger rufen sich die meisten Funktionen selbst auf. Stellen wir uns doch einfach mal vor, wir wollen, eine Funktion schreiben, der ein Zahlenwert übergeben wird, und dieser Wert mit allen vorhergehenden Zahlen addiert wird.

Listing 3.1: chap3/rekursion1.hs

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

Das Problem an dieser Funktion zeigt sich jedoch sobald wir die Funktion laufen lassen: Wir haben eine Endlosschleife konstruiert, sollten wir die 0 nämlich irgendwann passiert haben, so gehen wir weiter in den negativen Bereich der Zahlen. Wir müssen die Funktion also irgendwann abbrechen lassen.

3.1 Guards

Wollen wir einen Wert überprüfen, so benutzen wir in Haskell normalerweise Guards. Diese gleichen dem *if-then-else* Gebilden aus den meisten anderen Sprachen. Dazu lassen wir die Instruktionen aus der ersten Zeile weg zu erst weg, springen in die nächste Zeile, setzen einen Tabulator und schreiben nach einem senkrechten Strich (|) unsere Prüfung und danach die Instruktionen. Für alle Fälle, die auf keinen der Guards zutreffen, setzen wir den *otherwise* Operator. Es ist jedoch wichtig, dass die Reihenfolge stimmt. Der *otherwise* Operator muss als letztes in der Funktion stehen, da er ansonsten immer ausgeführt wird.

Listing 3.2: chap3/rekursion2.hs

```
1 summ :: Int -> Int
2 summ x
3     | (x == 1) = 1
4     | otherwise = x + summ (x-1)
```

In Zeile 2 des Skriptes unter 3.2 sehen wir die Abbruchbedingung der Rekursion (auch bekannt unter dem Namen Rekursionsanker). Diese wird nur dann aufgerufen, wenn unsere Zählervariable x den Wert 1 angenommen hat. Ansonsten wird der jeweilige Wert zum Vorgänger addiert. Vorstellen kann man sich die Rekursion folgendermaßen:

```
> summ 3
3 + (summ (3-1)) = 3 + (summ 2)
3 + 2 + (summ (2-1)) = 3 + 2 + (summ 1)
3 + 2 + 1
5 + 1
6
```

3.2 Guter Stil für Rekursionen

Wie in jeder Sprache gibt es sowohl guten, als auch schlechten Stil beim Coden. Rekursionen sollten immer **endrekursiv** sein. Endrekursive Funktionen zeichnen sich dadurch aus, dass der rekursive Aufruf am Ende der Funktion steht, und somit als letztes ausgeführt wird. Ein solches Konstrukt können wir im Codebeispiel 3.2 ansehen.

Rekursionen mit mehreren Parametern können auch sehr schnell unübersichtlich werden. Daher sollte man dringend versuchen, den Parametern statt einfacher Buchstaben einen aussagekräftigen Namen zu geben. Die einzige Ausnahme dafür sollten Zähler werden, und sollten die üblichen Bezeichnungen wie n oder i besitzen.

3.3 Fibonaccizahlen

Eine weitere, beliebte Rekursion sind die Fibonacci Zahlen. Dabei wird die n -te Fibonacci Zahl dadurch berechnet, indem man die Fibonacci Zahlen $(n-1)$ und $(n-2)$ addiert. Schreiben wir uns nun eine Funktion, die die n -te Fibonacci Zahl berechnet. Wir brauchen aber nicht nur einen sondern 2 Rekursionsaker, für den Fall, dass die 3 Zahl berechnet werden soll.

Listing 3.3: chap3/fibon.hs

```
1 fibon x
2     | x == 1 = 1
3     | x == 2 = 2
4     | otherwise = fibon (x-1) + fibon (x-2)
```

4 Pattern-Matching und Lazy-Evaluation

Ein weiteres, wichtiges Konzept in Haskell ist das sogenannte Pattern-Matching. Dabei wird in der Funktionsdefinition selbst, der Zustand von einer oder mehrerer Variablen, direkt gecodet. Wir können dafür die folgende Definition verwenden, um eine Boolesche Variable zu untersuchen und einen entsprechenden Output geben zu können.

Listing 4.1: chap4/truefalse.hs

```
1 -- Unser bisheriger Ansatz
2 truefalseALT :: Bool->String
3 truefalseALT x
4     | x == True = "Wert war True"
5     | x == False = "Wert war False"
6
7 -- Der neue Ansatz unter Benutzung von Lazy Evaluation
8 truefalseNEU :: Bool->String
9 truefalseNEU True = "Der Wert war True"
10 truefalseNEU x = "Irgendein anderer Wert"
```

Genausogut wäre es auch möglich, in Zeile 10 statt der Variablen „x“ den *False* Wert einzugeben. In vielen Fällen benötigen wir jedoch noch eine Funktion, für Werte, auf die Pattern-Matching vorher zutrifft, also eine Art *else* Wert.

4.1 Pattern-Matching bei Rekursionen

Mit Hilfe der Pattern-Matching können wir auch unsere Rekursionen leichter gestalten. Nehmen wir uns beispielsweise die Summenfunktion. Anstelle der Guards benutzen für den Rekursionsanker einfach eine weitere Zeile bei der Funktionsdefinition:

Listing 4.2: chap4/summ.hs

```
1 summ :: Int -> Int
2 summ 1 = 1
3 summ x = x + summ (x-1)
```

Eine Funktion kann mehrere Pattern-Matching Zeilen besitzen, das einzig wichtige ist, dass der undefinierte Fall als letztes eingetragen wird. Sehen wir uns beispielsweise die Funktion für die Fibonacci Zahlen nochmal an. Sowohl die Erste als auch die Zweite Zahl ergeben 1. Die nächsten Zahlen können wir einfach weiter berechnen. Es ist wichtig, dass

wir beide Zahlen definieren, da wir ja unter Umständen die dritte Zahl berechnet haben wollen. Wir definieren also die folgende Funktion:

Listing 4.3: chap4/fibon.hs

```
1 fibon 1 = 1
2 fibon 2 = 1
3 fibon x = fibon (x-1) + fibon (x-2)
```

4.2 Lazy-Evaluation auf Listen

Wir können auf Listen, wie etwa einem String, uns das jeweils erste Element holen und verarbeiten.

Ein schönes Beispiel hierfür kann eine Funktion sein, die uns das kleinste Element einer Integer Liste zurückgibt.

Listing 4.4: chap4/elemente.hs

```
1 -- Aufruf fuer unsere Funktion
2 smallestElement :: [Int]->Int
3 -- Wir muessen eine Hilfsfunktion aufrufen, da wir uns
   ansonsten unser kleinstes Element nicht merken koennen. Das
   erste Element der Liste wir fuers erste, als das kleinete
   Element angesehen, und ab dem zweiten Element
   weitergeprueft.
4 smallestElement (x:xs) = smallestElement1 xs x
5
6 -- Unsere Hilfsfunktion, damit wir unser kleinstes Element
   merken koennen
7 smallestElement1 :: [Int]->Int->Int
8
9 -- Rekursionsanker. Diese Zeile wird aufgerufen, wenn die
   Zahlenliste leer ist. Wir geben den kleinsten Wert zurueck
10 smallestElement1 [] kleinstenWert = kleinstenWert
11
12 -- Da ist noch mindestens ein weiterer Wert in der Liste, wir
   untersuchen ihn, und rufen diese Funktion rekursiv mit dem
   Rest der Liste auf.
13 smallestElement1 (x:xs) kleinstenWert
14     | x < kleinstenWert = smallestElement1 xs x
15     | otherwise = smallestElement1 xs kleinstenWert
```

Sehen wir uns die wichtigsten Zeilen nochmal näher an. In Zeile 4 und 13 sehen wir das Konstrukt $(x:xs)$. Das bedeutet, dass wir das erste Element der Liste nehmen und als Variable x speichern, und den Rest der Liste in xs halten (sozusagen der Sukzessor, also Nachfolger, von x). xs kann auch eine leere Liste enthalten! Diese würde jeweils in den

Zeilen 2 und 10 abgefangen werden. Wir stellen also fest, dass eine Liste äquivalent zur mathematischen Menge ist. Somit gilt für die leere Liste:

$$[] = \emptyset$$

4.3 Lazy Evaluation auf unendliche Listen

Wir lernten ja bereits dass es in Haskell unendliche Zahlenlisten gibt. Stellen wir uns mal vor, dass wir genau diese in einer Funktion verarbeiten wollen. Die meisten Programmiersprachen würde eine solche Funktion erst dann ausführen, sobald alle Parameter abgeschlossen wurden. Haskell jedoch wartet dank der Lazy Evaluation nur solange bis es arbeiten kann.

Mit Hilfe der `interact` Funktion ist es möglich, eine Eingabe in eine Funktion zugeben. Dazu darf sie nur einen String Parameter annehmen und muss einen solchen wieder ausgeben. Wir eine Funktion jedoch mit „`interact`“ gestartet, beginnt die Auswertung nie. Der Grund liegt darin, dass `interact` einen unendlichen Input an die jeweilige Funktion weitergeben will. Damit wir jedoch nach einer Eingabetaste weiterarbeiten können, müssten wir mit den Funktionen `lines` und `unlines` arbeiten. `lines` teilt einen String in eine Liste von String auf, bei dem jedes Element eine eigene Zeile ist. `unlines` macht das Gegenteil und muss somit vor dem Ende der Funktion aufgerufen werden. `lines` startet unsere Funktion, da die Lazy Evaluation nach der Eingabetaste beginnen kann.

5 List Comprehensions

Wir haben ja bereits festgestellt, dass man sich eine Liste eigentlich eher wie eine mathematische Menge vorstellen kann. Wer sich nun etwas mit mathematischen Mengen auskennt, kennt auch die Mathematische Darstellung für Mengen, deren Elemente eine bestimmte Eigenschaft haben, beispielsweise alle Elemente, die zwar in der Menge \mathbb{A} aber nicht in der Menge \mathbb{B} enthalten sind:

$$\{ x \mid x \in \mathbb{A} \wedge x \notin \mathbb{B} \}$$

Ein ähnliches Konstrukt gibt es auch in Haskell, für das bilden von Listen. Die sogenannten List Comprehensions sind eine beliebte Art und Weise, um mit wenig Aufwand, Listen zu bilden. Ein einfaches Beispiel wäre beispielsweise, eine Liste von 1 bis 1000, in denen nur gerade Zahlen stehen.

Listing 5.1: chap5/even.hs

```
1 evenlist = [x | x <- [1..1000], even x]
```

Die List Comprehension ist also der Mathematischen Darstellung nicht unähnlich.

5.1 Beispiel Das Sieb des Eratosthenes

Nun können wir beispielsweise alle Primzahlen berechnen lassen. Dazu verwenden wir den Algorithmus des „Sieb des Eratosthenes“. Dabei sind Primzahlen alle Zahlen, die nicht durch eine vorhergehende Primzahl teilbar sind.

Listing 5.2: chap5/eratosthenes.hs

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

Wir haben für diese Funktion jedoch keinen Rekursionanker programmiert. Da unsere eingegebene Zahlenliste aber unendlich ist, und eh niemals enden wird, benötigen wir diesen gar nicht.

5.2 Rekursive List Comprehensions

Ein weiterer Anwendungsbereich für List Comprehensions liegt im Bilden von Permutationen. Dabei wird eine List Comprehension rekursiv aufgerufen, und als Parameter jeweils das gerade verwendete Elemente gelöscht. Dazu laden wir zu Beginn der Datei das Data.List Paket für die delete Funktion.

Listing 5.3: chap5/permut.hs

```
1 kombi 0 x=[x]
2 kombi n x=[(y:ys) | y<-x, ys<- kombi (n-1) (delete y x)]
```

Testweise können wir die Funktion mit folgendem Aufruf starten:

```
> kombi 9 [1..9]
[1,2,3,4,5,6,7,8,9], [1,2,3,4,5,6,7,9,8], [1,2,3,4,5,6,8,7,9]...
```

5.3 Sortieren

Weiterhin kann mit Hilfe von List Comprehensions auch sehr schnell Sortieralgorithmen realisieren. So etwa der Insertionsort Algorithmus für Zahlenwerte:

Listing 5.4: chap5/insert.hs

```
1 insert' x xs = [ y | y <- xs , y < x ] ++ [ x ] ++ [ y | y <-
                xs , y >= x ]
```

6 Eigene Datentypen

Stellen wir uns mal vor, wir wollen einen eigenen Datentyp entwickeln, weil wir damit die Informationen einfacher verarbeiten können. Somit kann beispielweise das Pattern-Matching aus Kapitel 4 sehr einfach verwendet werden.

6.1 Lineare Datentypen

Stellen wir uns vor, wir erstellen, einen Datentypen mit denen wir die Jahreszeiten codieren wollen. Dies geht relativ schnell:

Listing 6.1: chap6/jahreszeiten.hs

```
1 data Jahreszeiten = Winter | Fruehling | Sommer | Herbst
```

Dazu müssen wir jedoch einige, wichtige Klassen laden. Allen voran Show, sodass wir eine Ausgabe erstellen können.

Listing 6.2: chap6/jahreszeitenfull.hs

```
1 data Jahreszeiten = Winter | Fruehling | Sommer | Herbst
2   deriving (Show)
```

Es gibt 4 Klassen, die für die Erstellung von Datentypen von Bedeutung sind:

Name	Bezeichnung	Bedeutung
Show	Show	Ausgabe von Informationen
Eq	Equivalence	Vergleich von Werten auf Gleichheit
Ord	Order	Vergleich von Werten ob kleiner oder groesser
Enum	Enumerate	Aufzählen von Werten

Wenn die Datei geladen ist, und wir die Ausgabe in unserem Interpreter testen, erhalten wir folgende Ergebnisse:

```
> Winter
Winter
> :type Winter
Winter :: Jahreszeiten
```

Als nächstes sehen wir uns, ein für die Programmierung wichtiges Konstrukt an.

6.2 Rekursive Datentypen

Einen einfachen Binärbaum kann man sich folgendermaßen vorstellen.

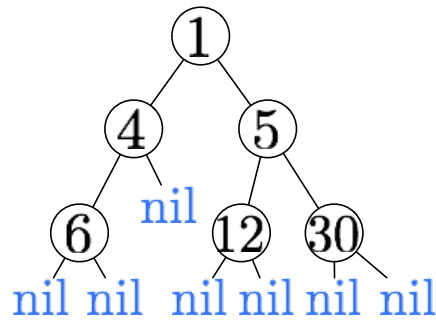


Abbildung 1: Ein einfacher Binärbaum

Man sieht also, dass jeder Knoten nur zwei Verbindungen hat, und die untersten Knoten nur noch auf ein *nil* und somit auf den Abschluss zeigen. Außerdem halten die Knoten jeweils einen Integer Wert. Daher erstellen wir den Datentyp `BBaum` folgendermaßen:

Listing 6.3: chap6/bbaum.hs

```

1 data BBaum = Knoten Integer BBaum BBaum | Nil
2     deriving (Show, Eq)

```

Das ist alles, was wir benötigen um einen Binärbaum, in Haskell zu definieren. Man sieht darin, dass wir in der Definition des Typs `BBaum` ebenfalls den `BBaum` verwenden. Das bedeutet, dass dort erneut ein `BBaum` steht (in dem wiederum entweder 2 `BBäume` stehen oder *nil*)

Die äußersten Knoten, also die Knoten, deren untergeordneten Knoten *nil* sind, werden gemeinhin als Blätter gekennzeichnet. Unsere erste Funktion wird daher prüfen, ob ein gegebener Knoten *nil* ist. Dazu verwenden wir erneut die *Lazy Evaluation*.

Listing 6.4: chap6/nil.hs

```

1 data BBaum = Knoten Integer BBaum BBaum | Nil
2     deriving (Show, Eq)
3
4 isNil :: BBaum -> Bool
5 isNil Nil = True
6 isNil _ = False

```

Wie man sieht, können wir direkt sehen, dass der Wert entweder `x` oder `Nil` ist.

Als nächstes durchsuchen wir den Baum auf der Suche nach einem Wert.

Listing 6.5: chap6/search.hs

```
1 data BBAum = Knoten Integer BBAum BBAum | Nil
2     deriving (Show, Eq)
3
4 searchBaum :: Int -> BBAum -> Bool
5 searchBaum _ Nil = False — Rekursionsanker
6 searchBaum i (Knoten x lBaum rBaum)
7     | x == i = True
8     | (searchBaum i lBaum) == False = (searchBaum i rBaum)
9     | otherwise = True
```

Haben wir einen *nil* Wert erreicht, so geben wir an dieser Stelle False zurück. In der nächsten Zeile sehen wir, dass wir den Wert des aktuellen Knotens untersuchen. Sollte er dem Suchwert entsprechen geben wir True zurück, ansonsten sehen wir uns den linken Teilbaum an. Finden wir darin nicht den gesuchten Wert, so untersuchen wir den rechten Teilbaum. Im „otherwise“ Fall, gehen wir davon aus, dass im rechten Teilbaum der Wert gefunden wurde, weshalb wir True zurückgeben.

7 Operationen auf Listen

Besonders häufig verwenden wir Listen. Daher gibt es auch einige Funktionen die Operationen auf Listen durchführen. Daher wollen wir uns diese Themen noch etwas näher ansehen.

7.1 map

map ist von allen Funktionen die leichteste. Dabei wird einfach nur auf jedes Element der Liste eine vorgegebene Funktion ausgeführt. Das Ergebnis ist eine neue Liste.

```
> map double [1,2,3]
[2,4,6]
> map reverse "abcd"
"dcba"
```

7.2 zip

Die *zip* Funktion nimmt 2 Listen an, und verbindet die jeweiligen Elemente in Tupel miteinander. Hat eine Liste mehr Elemente als die anderen, so werden die überzähligen Elemente ignoriert.

```
> zip [1,2,3] ['A','B','C','D','E']
[(1,'A'),(2,'B'),(3,'C')]
```

Eine Mischung zwischen *map* und *zip* wäre *zipWith*.

7.3 Faltung von Listen - foldl und foldr

Mit Hilfe von Faltungen werden Listenwerte miteinander ausgeführt. Ein Beispiel: Die Funktion *sum* nimmt eine Integer-Liste und addiert alle Werte miteinander.

```
> sum [1..5]
15
```

sum faltet also die Liste von Links nach rechts. Und addiert die Werte von links nach rechts. Wir können also die Fakultätsfunktion ähnlich definieren:

Listing 7.1: chap7/fakul

```
1 fakul n = foldl (*) 1 [1..n]
```

Genauso können wir auch Listen von rechts nach links falten.

7.4 Dynamische Programmierung

Betrachten wir nochmal unser Programm für die Fibonacci Zahlen:

Listing 7.2: chap3/fibon.hs

```
1 fibon x
2     | x == 1 = 1
3     | x == 2 = 1
4     | otherwise = fibon (x-1) + fibon (x-2)
```

Wir sehen, dass wir jede Zahl erneut berechnen müssen. Leichter wäre es doch aber, diese Werte abgespeichert zu halten. Die kann unseren Code bereits beschleunigen.

Dank `map` können wir zumindest schon eine Liste mit den Fibonacci Zahlen erstellen.

Listing 7.3: chap7/fibomap.hs

```
1 fibon x
2     | x == 1 = 1
3     | x == 2 = 1
4     | otherwise = fibon (x-1) + fibon (x-2)
5
6 fibomap = map fibon [1..]
```

Endlich können wir eine unendliche Liste mit Fibonacci Zahlen erstellen. Nach einigen Zahlen nimmt die Geschwindigkeit aber rapide ab. Leichter wäre es daher immer nur die jeweils letzten Zahlen wählen zu müssen um diese zu addieren.

Listing 7.4: chap7/fibopatternmatch.hs

```
1 fibon = 0 : 1 : zipWith (+) fibon (tail fibon)
```

Der Aufruf beweist, dass die Berechnung nun bei weitem schneller geht. Dieses Prozedere nennt man **memoisierung**, da wir uns die Werte nun merken und verwenden, statt jedesmal neu zu berechnen. Außerdem ist unser Code bereits viel kürzer geworden.

Literaturverzeichnis

- [1] Graham Hutton. *Programming in Haskell*. Cambridge University Press, 2008.
- [2] Simon Thompson. *Haskell: The Craft of Functional Programming 2nd Edition*. Addison Wesley, 1999.