

Aufgaben zu Haskell

Manuel Holtgrewe

30. April 2006

Wir geben nun `hallowelt` (also den Namen unsere Funktion ein) und erhalten folgende Ausgabe:

```
*Main> hallowelt
"Hallo Welt"
*Main>
```

Da ist nun unser „Hallo Welt“. Aber wir haben doch gar kein `print` oder irgendetwas in der Richtung aufgerufen? Nein, aber die Funktion `hallowelt` gibt eine Zeichenkette zurück und der Sprachinterpret GHCi weiß, wie er das zu behandeln hat.

Wir werden in den folgenden Beispielen also auch nicht einführen, wie man in Haskell etwas ausgibt, sondern benutzen die eingebaute Ausgabe des Interpreters.

3 Der Interpreter

Zum GHC bzw. GHCi sollten wir vielleicht noch etwas mehr wissen:

- GHC bedeutet „Glasgow Haskell Compiler“ und ist der Name eines Compilers für Haskell aber gleichzeitig der Name einer Haskell „Distribution“ also all dem, was man für das Ausführen von Haskell-Programmen benötigt. Zu dem zählen unter anderem Bibliotheken.
- Den Compiler rufen wir mit `ghc <Dateiname>` auf. Wir werden ihn jedoch in dieser kurzen Einführung nicht benutzen (man muss dann noch ein paar Dinge mehr beachten, wie eine `main` Funktion bereit stellen).
- Das Paket enthält auch einen Haskell-Interpreter. Dieser heißt GHCi, also GHC Interpreter. Wir rufen ihn mit `ghci <Dateiname>` auf. Der Interpreter liest dann unsere Haskell-Datei ein und wir können in ihm die Funktionen aufrufe, wie wir das bei „Hallo Welt“ oben gemacht haben. **Wir können jedoch keine neuen Funktionen definieren.**

Gerade der letzte Punkt wirft eine Frage auf: Müssen wir den Interpreter immer wieder beenden und neu starten um Änderungen an unseren Programmen zu laden? Und wie beenden wir den `ghci` überhaupt? Wer nicht einfach das Konsolenfenster geschlossen hat, der wird immer noch seine GHCI-Sitzung von oben offen haben.

Haskell ist zwar eine sehr akademische Sprache, aber so weltfremd ist die Implementierung nun auch wieder nicht. Im Interpreter können wir am Prompt (also hinter dem „`>`“) neben Funktionsaufrufen auch ein paar Befehle aufrufen:

- `:reload`, auch verfügbar als `:r`, lädt die gerade geladene Datei neu.

- `:load dateiname.hs`, auch verfügbar als `:l dateiname.hs`, lädt die Datei „dateiname.hs“.
- `:quit`, auch verfügbar als `:q` beendet den Interpreter wieder.
- `:help`, auch verfügbar als `:?` zeigt uns die anderen Befehle an, die wir hier vergessen haben.

4 Die äußere Form

Wir sollten vielleicht auch noch einmal kurz über die äußere Form von Haskell-Programmen sprechen: Haskell unterstützt zwei „Paradigmen“ der Programmdokumentation.

Zum einen können wir unsere Programme ganz normal mit Kommentaren dokumentieren (wie wir das bei „Hallo Welt“ oben schon gemacht haben). Ein Beispiel ist also:

```
-- Das Quadrat - Schon in der Antike war es der Grund für
-- Mord und Totschlag.
sqr n = n * n
```

Wir speichern diese Dateien mit der Endung `.hs` ab.

Wir können jedoch auch `literate programming` verwenden. Dabei räumt man seiner Programmdokumentation einen höheren Stellenwert ein, als dem Programm selbst. Man dokumentiert also nicht seinen Quellcode sondern füllt seine Dokumentation mit dem Quellcode selbst.

Diese Technik wurde unter anderem vom großen Knuth in seinem $\text{T}_{\text{E}}\text{X}$ benutzt. Er schrieb dafür seinen eigenen Pascal-Dialekt WEB. Auch unter Haskell-Programmierern ist *literate programming* beliebt. Haskell-Quellcode, der im *literate programming* Stil geschrieben ist, muss in Dateien gespeichert werden, die auf `.lhs` enden.

Dabei gibt es zwei wieder Möglichkeiten, *literate programming* zu betreiben: Die erste ist dual zur normalen Programmierung: Statt Kommentare zu kennzeichnen, kennzeichnen wir Programmzeilen aus. Programmzeilen müssen mit einem `⌘` beginnen und mindestens eine Leerzeile Abstand zu Kommentarzeilen haben.

```
Das Quadrat - Schon in der Antike war es der Grund für
Mord und Totschlag.
```

```
> sqr = n * n
```

Die andere Möglichkeit ist, es Quellcode in $\text{L}_{\text{A}}\text{T}_{\text{E}}\text{X}$ code einzubetten. Dabei wird dann alles zwischen `\begin{code}` und `\backslash end{code}` als Haskell-Quellcode interpretiert. Beispiel:

```

\documentstyle{article}

\begin{document}

\section{Das Quadrat}

Das Quadrat – Schon in der Antike war es der Grund für
Mord und Totschlag.

\begin{code}
sqr = n * n
\end{code}

\end{document}

```

Ich kann es nur jedem ans Herz legen, \LaTeX zu lernen. Damit wird nicht nur wissenschaftlicher Textsatz sehr viel einfacher sondern man lernt auch sehr viel über Textauszeichnung und Formatierung im Rechner. Jedoch sprengt eine Erklärung von \LaTeX den Rahmen dieses Dokumentes¹.

Benutzt man jetzt noch das Paket *haskell*, dann kann man seinen Haskell-Code noch ein bisschen schöner von \LaTeX formatieren lassen. Unser Beispiel von oben sieht dann etwa so aus:

```

— Das Quadrat – Schon in der Antike war es der Grund fuer
— Mord und Totschlag.
sqr n = n * n

```

So werden etwa die beiden Minuszeichen zu einem langen Strich zusammengefasst und Kommentare werden kursiv gesetzt. Nur die Umlaute funktionieren nicht mehr so ganz im „code“ Block. Wir werden im folgenden die schönere Formatierung verwenden und kurz darauf hinweisen, wenn ein neues Zeichen von dem Stil verändert wurde.

5 Ein paar gute Ratschläge

Das Programmieren in Haskell kann frustrierend sein: Der Interpreter gibt kryptische Fehlermeldungen, das Typsystem ist weder einfach noch angelehnt an die Systeme bekannter Sprachen und dann hat man noch keine Variablen und Objekte!

Daher ein paar gute Ratschläge, die Frust ersparen können:

- Wer nur C, Java oder sonst imperative Sprachen kann sollte sich möglichst nie denken, wie man das jetzt in Java etc. machen könnte. Haskell ist funktional, funktional funktioniert anders.

¹Die Not so short Introduction to \LaTeX <http://www.ctan.org/tex-archive/info/lshort/english/lshort.pdf> gibt eine gute Einführung.

- Lest die *Gentle Introduction To Haskell*. Sie ist wirklich gut und klärt viel: <http://www.haskell.org/tutorial/>
- Haskell ist recht aufgebläht aber nicht ganz unintuitiv. Wir werden in den folgenden Beispielen recht viele Sprachmittel verwenden. Probiere, die Beispielaufgaben mehrere male mit verschiedenen Sprachmitteln zu lösen. Wenn du mehr gelernt has, dann komme später nochmal auf die Aufgaben am Anfang zurück. So manches lässt sich dann mit Listen und Funktionen höherer Ordnung recht elegant lösen.
- Haltet immer eine Referenz des „Prelude“ bereit, der „Standardbibliothek“ von Haskell: <http://haskell.org/ghc/docs/latest/html/libraries/base/Prelude.html>.
- Wenn mal etwas auf Anhieb nicht zu funktionieren scheint: **Don't panic!** Das wird schon irgendwie.

6 Aufgaben

6.1 Funktionen erster Ordnung, Rekursion

Folgende Aufgaben kann man sehr einfach mit Rekursion lösen. Ich empfehle die *Gentle Intruction To Haskell*² und vielleicht noch die *Tour Of The Haskell Syntax*³ beim Lösen diesen Aufgaben „nebenher“ zu lesen um sich die nötige Fähigkeit in der Sprache anzueignen.

Zu einigen Aufgabe stehen in Klammern Stichwörter als Lösungstipps.

- Schreibe jeweils Funktionen, die Summe, Produkt, Durchschnitt, geometrisches Mittel, harmonisches Mittel zweier und dreier Zahlen berechnen (functions, guards/pattern matching).
- Schreibe eine Funktion, die für gegebenes n die Summe der Zahlen von 1 bis n zurück gibt (recursion).
- Schreibe eine Funktion, die für gegebene n und k die Summe der Zahlen von 1 bis n zurück gibt.
- Die Fibonacci-Zahlen sind definiert wie folgt: $F_0 = 0$, $F_1 = 1$, $F_n = F_{n-1} + F_{n-2}$. Schreibe eine Funktion, die für gegebenes n die Zahl F_n berechnet.
- Die Ackermann-Funktion ist definiert wie folgt: $ack(0, m) = m + 1$, $ack(n + 1, 0) = ack(n, 1)$, $ack(n + 1, m + 1) = ack(n, ack(n + 1, m))$. Gebe ack in Haskell an. (Hinweis: Die Ackermannfunktion wird *sehr* schnell größer. Zum

²<http://www.haskell.org/tutorial/>

³<http://www.cs.uu.nl/~afie/haskell/tourofsyntax.html>

Testen sollte etwas größeres als $ack(2, 2)$ nur auf einiges Risiko berechnet werden⁴).

- Schreibe eine Funktion, die für gegebene b und k die k -fache Potenz von b berechnet.
- Schreibe eine Funktion, die für gegebene m und n die Zahl m genau n mal aufsummiert.
- Benutze die beiden vorherigen Funktionen die für gegebene r , s und t die Summe $\sum_{i=0}^r s^t$ berechnet.

Nachdem wir die Funktionen implementiert haben geben wir im Interpreter einmal `:type <funktionsname>` ein und betrachten einmal die Signaturausgaben. Gibt man diesen Befehl ein, so wird der Interpreter sich den allgermeinsten (s.u.), möglichen Typ herausuchen, der möglich ist. Wir vergleichen sie dann mit dem Kapitel *Values, Types and Other Goodies* sowie *Type Classes and Overloading*.

Probiert, die Typausgaben zu verstehen (zu diesem Zeitpunkt ist keine Kenntnis über den Aufbau von Haskell's Typsystem notwendig, aber Übung macht den Meister). Haskell's Typsystem zu erklären würde den Rahmen dieses Dokuments sprengen. Aber einige Hinweise:

Generell sieht die Signatur einer Funktion so aus: $f :: t_1 \rightarrow t_2 \rightarrow \dots t_k$ wobei die t_i entweder genau spezifiziert sind - etwa `Integer` - oder aber nur mit Platzhaltern wie `a` belegt sind.

Mögliche Werte für die t_i sind etwa `Integer`, `Float`, `Char` oder `Int`. Außerdem können noch Tupel `((t1, t2, t3, ..))` oder Listen `[t]` angegeben sein.

Grundsätzlich gibt der Typname nach dem letzten Pfeil den Rückgabewert der Funktion an, alles andere sind die Parameter der Funktion. Steht ein Pfeil zwischen Klammern, dann handelt es sich beim Übergebenen Wert um eine Funktion.

Zum Beispiel hat `map` die Signatur `map :: (a -> b) -> [a] -> [b]`. Sie erwartet als ersten Eingabewert also eine Funktion, als zweiten eine Liste. Der Eingabewert der übergebenen Funktion muss mit dem Typ, der in der Liste eingegeben wurde übereinstimmen. Der Rückgabewert der Funktion stimmt mit dem Rückgabewert der Listenelemente überein, die `map` zurückgibt.

Es kann auch sein, dass eine Funktion nicht unbedingt an einen Typ wie `Integer` gebunden werden muss, jeder Wert des Types jedoch einen Nachfolger haben muss. Ist dies der Fall, so gibt der Interpreter zusätzlich an, von welcher Klasse die sonst allgemeinen Typen sein müssen. Zum Beispiel verlangt er, dass die Funktion `succ` als Eingabe einen Wert mit der Klasse `Enum` erhält: `succ :: (Enum a) => a -> a`.

⁴Der Wert von $ack(4, 2)$ kann hier nachgelesen werden: <http://www.kosara.net/thoughts/ackermann42.html>

6.2 Tupel, Listen, Funktionen höherer Ordnung

Wir lesen uns Kapitel 2.1 *Polymorphic Types* aus der *Gentle Introduction* durch und lösen folgende Aufgaben. Wir benutzen gleichzeitig die Dokumentation⁵ des Haskell Preludes um nachzuschlagen, welche Funktionen uns wohl helfen könnten. Wir können auch die Datei *Prelude.hs* angucken um zu sehen, wie die GHCi-Entwickler selber den Prelude implementiert haben.

- Schau dir Tupel an und schreibe `fst`, `snd` für Paare neu als `oursnd` und `oursnd`. Dann schreiben wir `triplefst`, `triplesnd` und `tripletrd` für Tripel (pattern matching).
- Schau dir Listen an und schreiben unsere eigenen Versionen von `head` und `tail`.
- Schreibe eine Funktion, die Tupel der Länge 3 in Listen konvertiert.
- Schreibe eine Funktion, die Tupel der Länge 2 umkehrt.
- Schreibe eine Funktion, die das n -te Element einer Liste zurück gibt.
- Schreibe eine Funktion, die das n -te bis m -te Element einer Liste in einer Liste zurück gibt.
- Schreibe eine Funktion, die eine unendlich lange Liste der natürlichen Zahlen zurück gibt.
- Schreibe eine Funktion, die für eine gegebene Liste die Summe der ersten n Zahlen zurück gibt.
- Schreibe eine Funktion, die für ein n die beiden vorherigen Funktionen benutzt, um die Summe der Zahlen von 1 bis n zurück gibt.
- Schau dir *list comprehensions* an und schreibe eine Funktion, die auch die natürlichen Zahlen enthält und eine Funktion, die alte Funktion, die die Summe der natürlichen Zahlen berechnet neu.
- Schau dir `map` an und schreibe es neu.
- Benutze `map` oder deine eigene Version und berechne für gegebenes k eine Liste mit den Zahlen $\{k \cdot n \mid n \in \mathbb{N}\}$.
- Schau dir `foldl` an und schreibe es neu.
- Benutze `foldl` und schreibe eine Funktion, die es zusammen mit `(+)` und *list comprehension* benutzt, um die Zahlen von 1 bis n zu berechnen.
- Benutze `foldl`, `(+)` und `map` sowie *list comprehension* Summe aller natürlichen Zahlen bis n hoch k zu berechnen.

⁵<http://haskell.org/ghc/docs/latest/html/libraries/base/Prelude.html>