1 Die Grundkonzepte von Haskell[1]

1.1 Wozu das alles!?

Reichen denn die gängigen Programmiersprachen nicht aus? Warum gleich ein völlig neues Konzept lernen? Dazu gab es im Original vom Autor (vgl. Fußnote) zwei Kommentare:

- Angeblich hat die Firma ERICSSON jede Menge Zeit und Geld gespart, seit deren Programmierer auf funktionales Programmieren umgestiegen sind. . .
- Funktionales Programmieren ist elegant. Quicksort ist in Haskell ein Dreizeiler!

1.2 Das Grundkonzept

Die Grundidee **imperativer** Programme ist das Abarbeiten - Zeile für Zeile - von einem gegebenen Programm, eventuell gesteuert durch Schleifen und Verzweigungen.
Bei **funktionalen** Programmiersprachen ist die Grundidee das **schrittweise Auswerten eines Ausdruckes**. Ein Ausdruck kann so etwas sein wie ein arithmetischer Ausdruck $3 + 4 - 2 * 1$, der in mehreren [Schritten] von *3 + 4 -2* [1], *3 + 2* [2], zu *5* [3] ausgewertet wird.
Das Auswerten erledigt ein sogenannter Interpreter, wie der oben genannte Hugs-Interpreter. Es gibt verschiedene Auswertungsstrategien, mehr dazu später.
Ein Programm in Haskell besteht im Prinzip nur aus Funktionen[2]. Diese kann man dann später in seinen Ausdrücken benutzen.

1.3 Ein erstes Beispiel

Hier ein „Programm“ in Haskell:

```
square :: Int -> Int
square x = x * x
```

Die erste Zeile ist nicht zwingend notwendig, aber sie erklärt dem System, dass *square* eine **Funktion** ist, die Integer auf Integer abbildet. Die zweite Zeile sagt dann, was *square* mit dem *x* machen soll. Wir nennen dabei *x* auch **Parameter** oder **Argument** der Funktion.
Der Interpreter würde den Ausdruck *square 3* durch *3* 3* **auswerten** (ersetzen) und diesen Ausdruck weiter zu *9* **auswerten**.

2.1 Auswertungsstrategien

Was passiert nun, wenn man einen etwas komplizierten Ausdruck hat, wie z.B. square (3 + 2)? Nun gäbe es mehrere Möglichkeiten der Auswertung:

1) Der Ausdruck wird zu *square* 5 ausgewertet und dann weiter wie im Beispiel oben zu 5 * 5 und weiter zu 25
2) Der Ausdruck wird erst zu *(3+2)*(3+2)* ausgewertet (Anwendung der Definition), dann zu *5*(3+ 2)* und schließlich zu *5 * 5* und *25*

Die erste Variante nennt man strikte Auswertung, die zweite entsprechend **nicht-strikte** Auswertung. Klar: Das Ergebnis darf nicht von der Auswertungsstrategie abhängen. Ob strikt oder nicht, man kommt zum selben Ergebnis. Um es vorwegzunehmen: Haskell arbeitet nach der nicht-strikten Auswertungs(-Strategie). Das hat unter anderem einen immensen Vorteil, wie wir gleich sehen werden.

---

[1] mit Material von Clemens Adolphs und Thorsten Görg

[2] und eigenen Datentypen, die wir allerdings im Unterricht nicht behandeln. Im Studium jedoch eines der zentralen Elemente sein werden.

Interessant wird es, wenn es um das **Terminieren** geht. Dazu basteln wir uns eine Endlosschleife:

```
endlos :: Int -> Int
endlos x = endlos (endlos x)
```

Man sieht leicht, dass ein Ausdruck wie *endlos 3* zu einer Art Endlosschleife führt[3] und der Interpreter irgendwann einen Stack-Overflow meldet. Schauen wir uns jetzt diese Funktion an:

```
drei :: Int -> Int
drei x = 3
```

Man sieht: Die Funktion *drei* gibt immer eine 3 aus, *drei 4* wird zu 3 ausgewertet, *drei (3 * 4 + 2 - 1)* wird zu 3 und so weiter.
Jetzt wird es interessant: Wir schauen uns *drei (endlos 1)* an. Bei der strikten Auswertung würde das System versuchen, zunächst *endlos 1* auszuwerten. Das führt aber zu einem Stack-Overflow, also terminiert die Anfrage an das Haskell-System nicht, das System hängt sich in einer Endlosschleife auf!
Bei der nicht-strikten Auswertung wird allerdings zunächst die Funktion *drei* ausgewertet (Anwendung der Definition), und da wird für jedes beliebige x die Zahl 3 als Auswertungsergebnis zurückgegeben.
Somit:
Wenn irgendeine Art der Auswertung für einen Ausdruck terminiert, terminiert auch die nicht-strikte Auswertung!

Bei Haskell wird die nicht-strikte Auswertung benutzt. Eine kleine Besonderheit ist hierbei noch ein Konzept, dass sich **Lazy-Evaluation** nennt: Wird derselbe Ausdruck mehrmals benutzt, so merkt der Interpreter dies und wertet dazu gehörende (wiederholende) Ausdrücke nur einmal aus:

```
square (3 + 2)
(3 + 2) * (3 + 2)
5*5
25
```

Das System rechnet dann **nur einmal** 3 + 2 aus.

Kleiner Vorgriff auf Listen

Erzeugen Sie am Hugs (oder ghcI)-Prompt eine Liste der natürlichen Zahlen von 1 bis 100

```
>[1..100] -- es müssen 2 Punkte sein!
```

Und jetzt
```
>[1..]
```

Abbruch mit CTRL+C

Aber was passiert bei
```
> [1..]!!23
```

oder
```
> [1..]!!76576329
```

➔ Nicht-strikte Auswertung, denn es wird nur das ausgewertet, was für die Funktion *!!* gebraucht wird.

---

[3] Das liegt insbesondere daran, dass bei dieser rekursiven Definition kein sog. **Rekursionsanker** angegeben ist. Dieser beendet die rekursiven Aufrufe. Beispiel:

```
nichtendlos x
  | x == 0        = 0                          -- Rekursionsanker, es wird einfach eine 0 zurückgegeben
  | otherwise     = nichtendlos (nichtendlos x-1)  -- Wichtig! Der Parameter muss beim rek. Aufruf verändert werden!
```