**Haskell Live**

# #1 Einführung in Haskell/Hugs

Johannes Birgmeier
johannes.birgmeier@tuwien.ac.at

Jürgen Cito
juergen.cito@tuwien.ac.at

Sebastian Rumpl
sebastian.rumpl@tuwien.ac.at

Gerald Schermann
gerald.schermann@tuwien.ac.at

Michael Schröder
michael.schroeder@tuwien.ac.at

12. Oktober 2012

## Einführung in `hugs`

`hugs`[1] ist ein Interpreter für die funktionale Programmiersprache Haskell. Abhängig vom Betriebssystem wird der Interpreter entsprechend gestartet, unter GNU/Linux beispielsweise mit dem Befehl `hugs`. Tabelle 1 zeigt eine Übersicht der wichtigsten Befehle in `hugs`.

[1] Haskell User's Gofer System

| Befehl | Kurzbefehl | Beschreibung |
|---|---|---|
| `:edit name.hs` | `:e name.hs` | öffnet die Datei `name.hs` in dem Editor, der in `$EDITOR` (Unix) bzw. in WinHugs in den Optionen definiert ist. **Tipp:** Wird `hugs` mit dem Parameter `-eEDITOR` gestartet, dann bewirkt der Befehl `:edit` das öffnen im besagten Editor. (ersetze EDITOR durch einen installierten Editor bspw. `vim`) |
| `:load name.hs` | `:l name.hs` | lädt das Skript `name.hs`. Man kann fortan die einzelnen Funktionen aus dem Script im `hugs` auszuführen. |
| `:reload` | `:r` | erneuertes Laden des zuletzt geladenen Skripts |
| `:type` ***Expr*** | `:t` ***Expr*** | Typ von ***Expr*** anzeigen |
| `:info` ***Name*** | | Informationen zu ***Name*** anzeigen. ***Name*** kann z.B. ein Datentyp, Klasse oder Typ sein |
| `:cd dir` | | Verzeichnis wechseln |
| `:quit` | `:q` | `hugs` beenden |
| ***Expr*** | | Wertet ***Expr*** aus (wobei für diese Auswertung das momentan geladene Skript herangezogen wird; siehe `:load`) |

Table 1: Befehle in `hugs`

# Einfacher Haskell Code

Einfache Addition zweier Zahlen: Erster Parameter ist somit vom Type Integer, der zweite ebenfalls, der Rueckgabewert der Funktion ist auch vom Typ Integer

$myadd :: Integer \rightarrow Integer \rightarrow Integer$ -- Signatur
$myadd\ a\ b = a + b$

'myadd' kann auch Infix verwendet werden: 5 ‘myadd‘ 5

Funktion, welche eine andere Funktion verwendet
$addSeven :: Integer \rightarrow Integer$
$addSeven\ a = myadd\ a\ 7$

Beispiel fuer Pattern-Matching, achten auf die Reihenfolge
$eqSeven :: Integer \rightarrow Bool$
$eqSeven\ 7 = True$
$eqSeven\ \_ = False$

Alternative Implementierung unter Verwendung von if

```
eqSeven2 a =
  if a ≡ 7 then
    True
  else
    False
```

Noch eine Alternative, 'otherwise' ist ein Alias fuer True

```
eqSeven3 a
  | a ≡ 7 = True
  | otherwise = False
```

## Listen notieren

Listen können einfach notiert werden: Zum Beispiel erzeugt der Ausdruck `[1,2,3,4]` eine Liste von gleicher Darstellung. In Tabelle 2 sind einfache Beispiele angeführt.

| Ausdruck | Ergebnis | Beschreibung |
|---|---|---|
| `[1,2,3,4,5]` | `[1,2,3,4,5]` | Erzeugt eine Liste mit den Elementen 1 bis 5 |
| `[1..5]` | `[1,2,3,4,5]` | Erzeugt eine Liste mit den Elementen 1 bis 5 |
| `[1,4..14]` | `[1,4,7,10,13]` | Siehe nächstes Beispiel |
| `[`$a$`,`$b$`..`$x$`]` | `[` $a$, $b = a + d$, $a + 2d$, $a + 3d$, …, $x - d < a + nd \leqslant x$ `]` | Es wird ein Offset $d$ (Differenz von $a$ und $b$) ermittelt. Zu der Basis $a$, wird bis zum Wert $x$, jede Summe der Basis plus einem Vielfachen des Offsets, der Liste hinzugefügt |
| `[]` | `[]` | Leere Liste aka. “nil” |
| `1:(2:(3:(4:[])))` | `[1,2,3,4]` | `(:)` aka. “cons” |
| `1:2:3:4:[]` | `[1,2,3,4]` | “cons” ist rechts-assoziativ |
| `"asdf"` | `"asdf"` | Liste von `Char`. |
| `'a':'s':'d':'f':[]` | `"asdf"` | Beachte, dass der Typ `String` dem Typen `[Char]` entspricht. |

Table 2: Einfache Beispiele für Listen

## Listen verarbeiten

Summiert die Elemente einer Integer-Liste

```
mysum :: [Integer] → Integer
mysum [] = 0   -- Abbruchbedingung der Rekursion
mysum (x : rest) = x + (mysum rest)
```

Alternative Implementierung mit der vordefinierten Funktion 'sum'

```
mysum2 list = sum list
```

Addiert die Zahl 'a' zu jedem Listenelement

```
addX :: [Integer] → Integer → [Integer]
addX [] _ = []
addX (x : rest) a = (x + a) : (addX rest a)
```

Alternativimplementierung unter Verwendung von List-Comprehension

```
addX_lc list a = [x + a | x ← list]
```

Alternative mit der Funktion 'map' Fuer jedes Listenelement 'x' aus 'list' wird die Operation 'x + a' ausgefuehrt, das Ergebnis bildet wieder eine Liste

```
addX_map list a = map (+a) list
```

# Integer VS. Int

Konstruiertes und sinnfreies Beispiel um den Umgang von Integer und Int zu zeigen. Die Funktion 'length' liefert die Laenge der Liste als Typ Int, 'sum' allerdings berechnet die Summe als Typ Integer. Zur Konvertierung wird die Funktion 'fromIntegral' eingesetzt.

```
sumPlusLength :: [Integer] → Integer
sumPlusLength list = summe + len
  where
    len = fromIntegral (length list)
    summe = sum list
```

Grosse Zahl bei Integer

```
big :: Integer
big = 1000 * 100000000000 * 10000000
```

Grosse Zahl bei Int

```
big2 :: Int
big2 = 1000 * 10000000000 * 10000000
```

## Rudimentäres Debugging

Use `trace` from `Debug.Trace`:

```
import Debug.Trace

lafDebug :: [Integer] → [Integer]
lafDebug [] = trace "Liste zu Ende" []
lafDebug (x : xs) = trace debugMessage (neuesX : (lafDebug xs))
  where
    neuesX = addiereFuenf x
    debugMessage = "Berechne: addiereFuenf " ++ (show x) ++ " = " ++ (show neuesX)
```

## Hinweise

Diese Datei kann als sogenanntes "Literate Haskell Skript" von `hugs` geladen werden, als auch per `lhs2TeX`[2] und LaTeX in ein Dokument umgewandelt werden.

## Referenzen

Ehre wem Ehre gebührt: Dieses Dokument ist eine adaptierte Version aus dem WS11/12 von Bong Min Kim, Christoph Spörk, Florian Hassanen und Bernhard Urban.

[2] `http://people.cs.uu.nl/andres/lhs2tex`