LV „ Programmierung “
Prof. Dr. H. Vogler

Dresden, den 05.05.2010

Hinweise zur praktischen Nutzung von Haskell

Angeboten und empfohlen wird die Möglichkeit, die in der Lehrveranstaltung „Programmierung“ erworbenen Haskell-Kenntnisse durch praktische Übungen/Tests am PC zu vertiefen. In den Kabinetten E065 und E067 des Fakultätsrechenzentrums finden Sie die technischen Voraussetzungen, um unter WINDOWS wahlweise mit Hilfe des Haskell-Interpreters `hugs` oder des gleichwertigen Kompilers `ghc` Haskell-Programme abarbeiten zu können.

Die im Folgenden gegebenen Hinweise sollen Ihnen den Zugang zur erforderlichen Rechentechnik erleichtern und hilfreich bei der Abfassung erster einfacher Haskell-Programme sein. Der Einfachheit halber konzentrieren wir uns auf `hugs`; hier ist insbesondere die Ausgabeformatierung des Funktionswertes oft standardmäßig vorhanden.

- Starten Sie `winhugs`,
- öffnen Sie dann Ihr Haskell-Quellcodefile (z.B. bsp1.hs),
- geben Sie am Prompt (`Hugs>`, bei einigen Installationen auch `Prelude>`) `main` ein, um die Hauptfunktion zu starten,
- die Abarbeitung beginnt– falls Eingabewerte erforderlich sind, erscheint eine Leerzeile, um den jeweiligen Wert einzugeben.

Zur Herstellung der Programmdatei kann ein Texteditor verwendet werden; die Datei erhält die Endung `.hs`. Nutzen Sie für Einrückungen im Programmtext die Leer- und NICHT die Tabulatortaste.

Wir geben nun einfache Programmrahmen für spezielle Klassen zu berechnender Funktionen an:

Eingabe: Int-Werte von Konsole

```
module Main where

... Datentypdefinitionen
... Funktionsdefinitionen
...

main = do e1 <-readln
          ..
          en <-readln
          print(f a1 .. am)  f sei Name der zu berechnenden Fkt.
```

Erlaubt die Eingabe von mehreren `Int`-Werten (d. h. `e1..en`), die bei der Angabe der Argumentausdrücke `a1..am` benutzt werden können, und gibt dann für diese Argumente den Ergebniswert von `f` aus. Beachten Sie, dass die Einrückungen in `main` unbedingt notwendig sind!

Hier ein Beispielprogramm bsp1.hs (Lösung der AGS 11.2)

```
module Main where

   f:: Int->Int->Int->[Int]
   f x y z = [x] ++ (f y z (x+y+z))

   p :: [Int]->Int->Int
   p (x:xs) 0    = x
   p (x:xs) (p+1)= p xs (p-1)

main = do e <- readln
          print(p (f 1 1 1) e)
```

Eingaben: als Argumentausdruck

Ist die einfachste und praktikabelste Möglichkeit, Instanzen von algebraischen Datentypen einzugeben. Soll ein algebraischer Datentyp ebenfalls als Funktionswert ausgegeben werden, muss durch Anfügen von `deriving Show` an der Datendefinition diese Ausgabe vorbereitet werden. Sowohl Eingabe als auch Ergebnisausgabe dieser Instanzen sind in einer algebraischen Schreibweise notiert.

```
...
...
data Tree = K Int Tree Tree | Nil deriving Show
...
f :: Tree -> Tree
...
...
main = print (f(K 2 Nil (K 3 Nil Nil))
```

Ein Beispiel für diese Eingabeform ist bsp2.hs (Lösung der AGS 11.11b)

```
module Main where

data TA = A Int TA TB | NilA deriving Show
data TB = B Int TB TA | NilB deriving Show

trans :: TA -> TA
trans t = transA t 0

transA :: TA -> Int -> TA
transA NilA _ = NilA
transA (A _ ta tb) z = A z (transA ta z) (transB tb z)

transB :: TB -> Int -> TB
transB NilB _ = NilB
transB (B _ tb ta) z = B (z+1) (transB tb (z+1)) (transA ta (z+1))

main = print(trans (A 5 (A 1 (A 2 NilA NilB)(B 4 (B 0 NilB  NilA)(NilA)))
                        (B 3 (B 6 NilB NilA)(A 8 NilA NilB))))
```

Sollen weder Daten/Funktionen exportiert noch importiert werden, so genügt es, ausschließlich die Definitionen (Daten und Funktionen) in einer Datei zusammenzufassen. Ist dann diese Datei geladen, wird am Prompt die auszuführende Funktion mit ihren aktuellen Parametern aufgeschrieben/aufgerufen.

Ein Beispiel eines solchen Rumpfprogramms ist bsp3.hs (Lösung AGS 11.11b,c)

```
data TA = A Int TA TB | NilA deriving Show
data TB = B Int TB TA | NilB deriving Show

trans :: TA -> TA
trans t = transA t 0

transA :: TA -> Int -> TA
transA NilA _ = NilA
transA (A _ ta tb) z = A z (transA ta z) (transB tb z)

transB :: TB -> Int -> TB
transB NilB _ = NilB
transB (B _ tb ta) z = B (z+1) (transB tb (z+1)) (transA ta (z+1))

list :: TA -> [Int]
list NilA = []
list (A n ta tb) = [n]++(list ta)++ (listb tb)

listb :: TB -> [Int]
listb NilB = []
listb (B _ tb ta) = (listb tb)++(list ta)
```

Sinnvolle Aufrufe wären hier:

```
Hugs> list (A 3 (A 2 NilA NilB) NilB)
```

oder

```
Hugs> trans (A 3 (A 2 NilA NilB) NilB)
```

Mit den vorgestellten Programmrahmen lassen sich alle in den Übungen besprochenen Aufgaben (und weitere aus der AGS) rechentechnisch lösen.
Umfassende Informationen zur Haskell-Programierung und Haskell-Installation finden Sie im Internet unter haskell.org.