Anhang B

Benutzung von Haskell-Interpretern

Autor: Jost Berthold

B.1 Benutzung von Hugs

B.1.1 Aussehen und Verfügbarkeit

HUGS (Haskell-User's Gofer System) ist ein für seine Leistungsfähigkeit und einfache Bedienung bekannter Interpreter für Haskell. Er wurde an der Universität von Yale entwickelt und ist frei verfügbar unter *www.haskell.org/hugs*.
Es stehen Versionen für verschiedene Betriebssysteme zum Download zur Verfügung. Besonders komfortabel ist die Windows-Version, wo (wie unter Windows üblich) die Maus zur Bedienung benutzt werden kann. Die Linux-Version (und auch der einfache `hugs.exe` unter Windows) arbeiten dagegen mit Befehlen am Prompt (s.u. Section B.1.2).

An unserem Fachbereich ist HUGS in aktueller Version in folgendem Pfad installiert:

- **Windows:** `S:\APPLICATIONS\Languages\WinHugs`
- **Linux:** `/app/lang/functional/bin/hugs`

Für eine bessere Bedienung empfiehlt es sich, diesen Pfad der PATH-Variable hinzuzufügen, damit man den Interpreter durch den einfachen Aufruf `hugs` starten kann, siehe Section B.3. Ansonsten muss man stets den gesamten Pfad angeben. Dies funktioniert natürlich auch...

B.1.2 Verfügbare Kommandos in Hugs

Nach dem Start zeigt Hugs den Prompt, wo beliebige Ausdrücke eingegeben werden können. Der Interpreter wertet sie aus und zeigt das Ergebnis direkt an. Es können aber nur Haskell-Standardfunktionen, etwa die üblichen Rechenoperationen, benutzt werden. Natürlich wollen wir etwas mehr, nämlich eigenen Code laden und interpretieren lassen. Mit Hilfe des `load`-Kommandos laden wir eine Datei:

```
Prelude> :load MeineDatei.hs
```

kurz	lang	Beschreibung
`:?`	–	Hilfe
`:l` *datei*	`:load` *datei*	Lädt die angegebene Datei in Hugs, entlädt die vorherige.
`:l`	`:load`	`load` ohne Argument: lädt das Standardmodul *Prelude.hs*. Alles, was vorher geladen war, wird entladen.
`:r`	`:reload`	Die geladene Datei wird erneut geladen. Wichtig beim Programmieren, wo die Datei laufend geändert wird.
`:a` *datei2*	`:also` *datei2*	Die Datei *datei2* wird zu bisher geladenen Dateien hinzu genommen. Es wird nichts entladen.
`:t` *expr*	`:type`*expr*	Der Typ des Ausdrucks *expr* wird ermittelt und dargestellt. Wichtig bei der Fehlersuche.
`:i` *name*	`:info` *name*	Informationen über das Objekt mit Namen *name* werden angezeigt.

Tabelle B.1: Die wichtigsten Hugs-Kommandos

Kommandos in Hugs werden immer mit einem Doppelpunkt eingeleitet.
Neben dem Ladebefehl gibt es weitere Kommandos, die z.B. die Umgebung beeinflussen oder Informationen über die geladene Datei und ihren Inhalt liefern. Mit `:?` am Prompt erhält man eine Liste von Kommandos und erfährt auch, dass man sie alle mit ihrem ersten Buchstaben abkürzen kann. *Tabelle B.1.2 listet die wichtigsten Kommandos auf.*

Hugs

```
Prelude> :?
LIST OF COMMANDS:  Any command may be abbreviated to :c where
c is the first character in the full name.

:load <filenames>   load modules from specified files
:load               clear all files except prelude
:also <filenames>   read additional modules
:reload             repeat last load command
[...]
:quit               exit Hugs interpreter
Prelude>
```

Nach Laden einer Haskell-Datei (z.B. `Simple.hs`) stehen die Definitionen darin zur Verfügung.[1] Eingegebene Ausdrücke können alle Funktionen benutzen, die in dieser Datei definiert wurden. Auf diese Weise können alle Funktionen getestet werden, selbst interne Hilfsfunktionen.

B.1.3 Einfache Beispiele mit Hugs

Beispieldatei

Die folgende Datei enthält einfache Funktionsdefinitionen, wie sie auch in der Vorlesung gezeigt wurden. Sie dient dazu, die grundlegende Syntax von Haskell klar zu machen

[1] Das sog. *Prelude* enthält die Grunddefinitionen von Haskell 98 und bleibt immer geladen, falls man es nicht explizit verhindert.

und die elementaren Datentypen einzuführen.
Code sollte stets kommentiert werden, die Syntax dafür wird daher gleich zu Beginn gezeigt.

Haskell Code

```
{- Kommentare mit "--" bis Zeilenende oder
  mit "{ und -" in mehreren Zeilen -}

-- einzeiliger Kommentar

{- mehrzeiliger Kommentar...
  Diese Kommentare können geschachtelt werden:
  {- Dies ist ein Kommentar auf Ebene 2,
    der sich über zwei Zeilen erstreckt...  -}
  Dies ist ein Kommentar auf  Ebene 1, der
  Ebene, auf der sich auch "mehrzeiliger Kommentar..."
  befindet! -}
```

Die Datei soll ein Modul "Simple" definieren, das hinterher in andere Dateien importiert werden kann. Im Kopf kann festgelegt werden, was aus der Datei exportiert wird (falls nichts angegeben: alles).

Haskell Code

```
module Simple
  (add, square, simple, eps)
  where
-- hier gehen die Definitionen los:
```

Wir definieren ein paar Funktionen, wie sie auch im Script stehen:

Haskell Code

```
-- Funktionen
add x1 x2 = x1 + x2
square x  = x * x
simple a b c = a*( b+c )

-- x hoch 4, mit der square-Funktion
powerfour x = square x * square x

-- Konstanten, Typen
newline = '\n'
eps  = 0.000001

lessEqualEps x = x <= eps
```

Basistypen und Typfehler

In Haskell haben alle Definitionen einen bestimmten Typ, den man auch explizit angeben kann. Basistypen sind z.B. `Bool, Int, Integer, Float, Double, Char, String,...` Falls kein Typ angegeben wird, *inferiert* Hugs den Typ. Mit `:t` kann der Typ eines Ausdrucks abgefragt werden:

Hugs

```
Simple> :t eps
eps :: Double
Simple> :t eps+1
eps + 1 :: Double
Simple> :t simple eps eps
simple eps eps :: Double -> Double
Simple>
```

Die letzte definierte Funktion `lessEqualEps` soll ihr Argument mit "Epsilon" (`eps`) vergleichen, dies führt zu einem Typfehler, falls x nicht den gleichen Typ wie `eps` hat.

Hugs

```
Simple> lessEqualEps newline
ERROR - Type error in application
*** Expression     : lessEqualEps newline
*** Term           : newline
*** Type           : Char
*** Does not match : Double

Simple> lessEqualEps (42::Integer)
ERROR - Type error in application
*** Expression     : lessEqualEps 42
*** Term           : 42
*** Type           : Integer
*** Does not match : Double

Simple>
```

Wie man sieht, kann man in der Kommandozeile explizit Typen angeben (hier provoziert erst das den Fehler). Auch bei Funktionsdefinitionen kann ein Typ angegeben werden:

Haskell Code

```
simple2 :: Int -> Int -> Int -> Int
simple2 a b c = a * ( b + c )
```

Hugs

```
Simple> :t simple2
simple2 :: Int -> Int -> Int -> Int
Simple> :t simple
simple :: Num a => a -> a -> a -> a
Simple> simple2 (2 ^ 31) 2 (-1)
-2147483648
Simple> simple (2 ^ 31) 2 (-1)
2147483648
Simple>
```

Wie man hier sieht, hat die Funktion `simple` einen allgemeineren Typ, der auf einer *Typklasse* namens `Num` basiert (Genaueres später in der Vorlesung). Der obige Aufruf

rechnet mit dem Typ `a=Integer` statt `Int`, daher das unterschiedliche Ergebnis (Überlauf ins Negative).
Typfehler passieren in Haskell sehr häufig, auch sofort beim Laden einer Datei. Dadurch werden Programmierfehler wie dieser sofort entdeckt.

Haskell Code

```
-- Die "main"-Funktion beschreibt, was ein *übersetztes* Programm
-- tun würde, genauso wie Java's "main" Methode von Objekten
main = print (simple2 newline 42 "abc")
```

Hugs

```
Simple> :r
Reading file "Simple.hs":
Type checking
ERROR "Simple.hs":55 - Type error in application
*** Expression     : simple2 newline 42 "abc"
*** Term           : "abc"
*** Type           : String
*** Does not match : Int

Prelude>
```

B.1.4 Ein typisches Problem: Layoutfehler

Haskell arbeitet mit dem sog. *Layout*, um exzessive Klammerung zu vermeiden. Z.B. gelten lokale Definitionen tatsächlich "lokal", wenn sie korrekt eingerückt sind. Falsche Einrückung oder fehlende Klammern können eigenartige Fehlermeldungen beim Laden verursachen, weil der Interpreter den Code selbstständig um Trennzeichen "ergänzt". Das gleiche gilt für fehlende Klammern.

Haskell Code

```
-- lokale Definitionen und Layout, "Guards" für Fallunterscheidung
zinseszins :: Double -> Double -> Int -> Double
zinseszins   zinssatz   summe   jahre
     | jahre == 0 = summe
     | jahre >  0 = let
                      erstesJahr = summe * faktor
                      faktor     = 1 + (0.01 * zinssatz -- fehlt: ')'
                    in zinseszins zinssatz erstesJahr (jahre - 1)
     | otherwise = error "Jahre negativ"

zinseszins2 :: Double -> Double -> Int -> Double
zinseszins2 zinssatz summe jahre | jahre < 0 = error "Jahre negativ"
                                 | otherwise = summe * faktor ** dbljahre
        where faktor   = 1 + (0.01 * zinssatz)
              dbljahre = fromInt jahre -- Test: Einrückung verkleinern
```

Ohne die schließende Klammer sieht die Fehlermeldung *so* aus, immerhin mit dem Hinweis auf Layout:

Hugs

```
Simple> :r
Reading file "Simple.hs":
Parsing
ERROR "Simple.hs":73 - Syntax error in expression (unexpected `}', possibly
due to bad layout)
Prelude>
```

Hinweis: Unter *www.cs.kent.ac.uk/people/staff/sjt/craft2e/errors/allErrors.html* werden weitere einfache Fehler an ausführlichen Beispielen besprochen.

B.2 Der Glasgow Haskell Compiler (GHC)

Der *Glasgow Haskell Compiler* (GHC) ist die bekannteste und effizienteste Implementierung von Haskell, allerdings als Compiler nicht so handlich für die kleineren Übungsaufgaben, die wir bearbeiten. Trotzdem sollte man sich den GHC auf jeden Fall mal anschauen. GHC ist schon relativ lange für viele Unix-Derivate und seit einer Weile auch für Windows verfügbar (*www.haskell.org/ghc*).

Seit Version 5.x verfügt der GHC ebenfalls über einen Interpreter (namens *GHCi*), aber nicht als echtes Windows-Programm, sondern textbasiert. Er bietet nicht den gleichen Komfort wie Hugs, kann aber dafür auch mit bereits übersetzten Dateien umgehen, was für größere Programme in mehreren Modulen sehr interessant ist. Die Bedienung entspricht etwa der von Hugs (Kommando-orientiert).

Ghc(i) findet sich am Fachbereich unter:

- **Windows:** `S:\Lehre\2006WS\Loogen\ghc\ghc-6.4.2`
  Die ausführbaren Dateien (Programme wie ghc und ghci liegen in `\bin`.
- **Linux:** `/app/lang/functional/bin/ghc` bzw. `ghci`

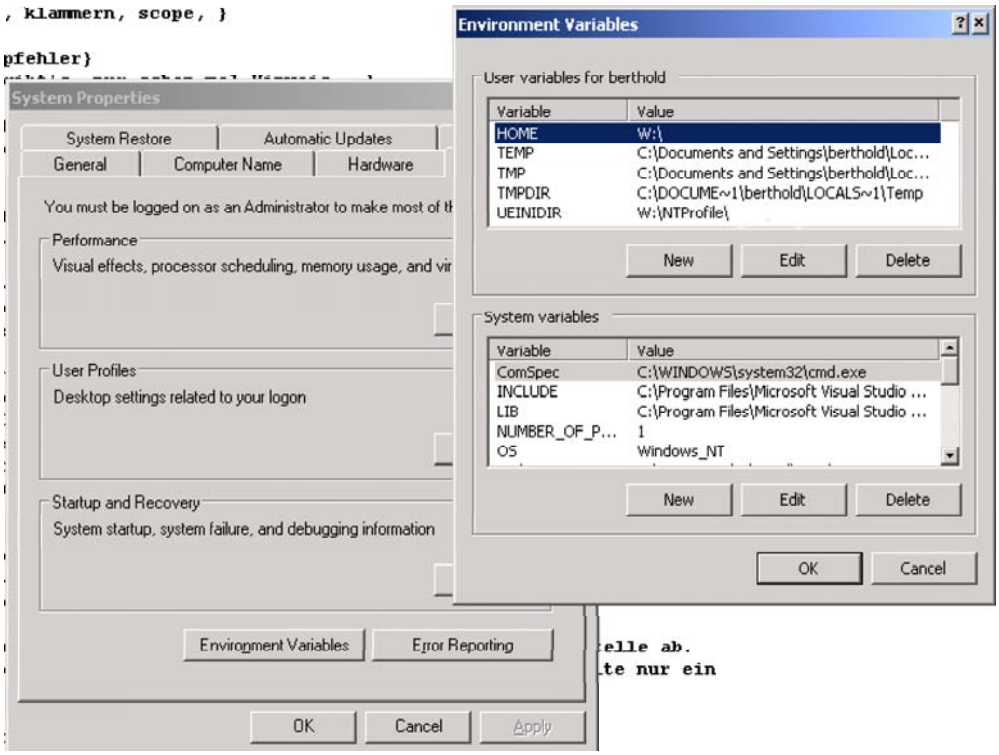
Auch hier: am besten der PATH-Variable hinzuzufügen, siehe Section B.3.

B.3 IDE-Ersatz: Konfiguration eines Editors (Beispiel)

B.3.1 Ultraedit

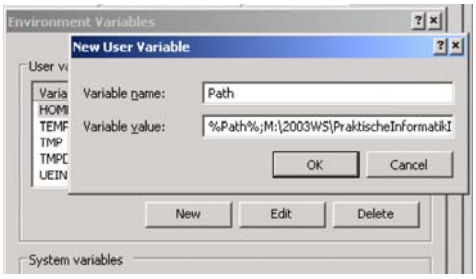
Der Editor *UltraEdit*, der am Fachbereich auf Windows-Rechnern verfügbar ist, lässt sich je Benutzer individuell konfigurieren, wenn man vorher die Variable UEINIDIR auf ein global sichtbares Verzeichnis mit Schreibrechten setzt.

Zum Setzen dieser Variable wählt man im "Control Panel" (Startmenü) die Option *Performance and Maintenance* → *System*. Im Register "Advanced" findet sich "Environment Variables", die man z.B. wie folgt modifiziert:



UltraEdit legt nun seine Konfiguration *uedit32.ini* an dieser Stelle ab. Dadurch werden Änderungen an der Konfiguration möglich und wirken sich auf allen Rechnern des Fachbereichs aus. Wohin die Variable zeigt, ist Geschmacksache, es sollte nur ein *global sichtbares* Benutzerverzeichnis sein (und natürlich mit Schreibrechten :-).

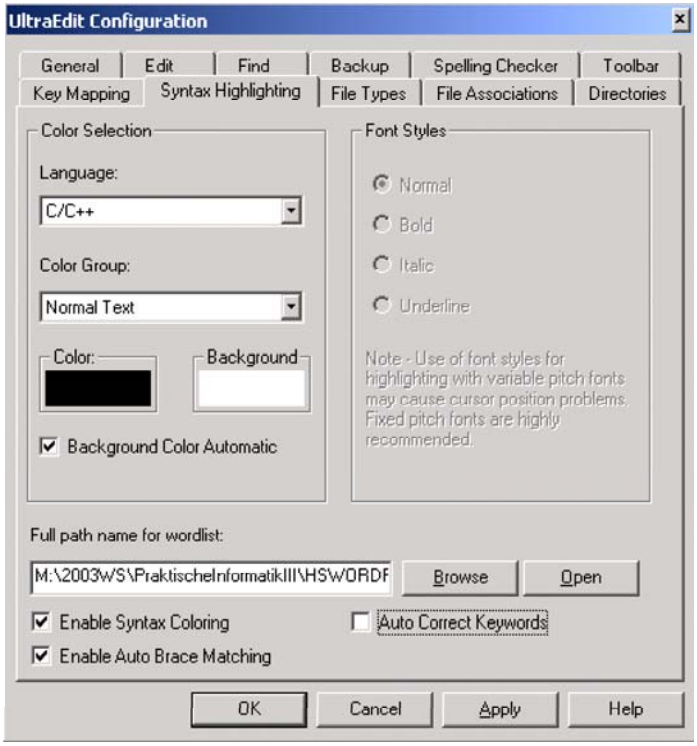
Wo wir gerade hier sind, setzen wir am besten noch einen Pfad auf das Hugs-Verzeichnis (und evtl. das des GHC, Unterverzeichnis\bin). Eine neue User-Variable PATH erhält mit *%Path%* den bisherigen Pfad, dahinter (abgetrennt mit Semikolon) zusätzlich für Hugs den Eintrag *S:\APPLICATIONS\Languages\WinHugs*:



Achtung! Wer die bisherige Konfiguration (Java-Knöpfe etc.) behalten will, muss *jetzt* die lokale Datei *C:\Program Files\UltraEdit\Uedit32.ini* an die neue Stelle kopieren, bevor Ultraedit gestartet wird.

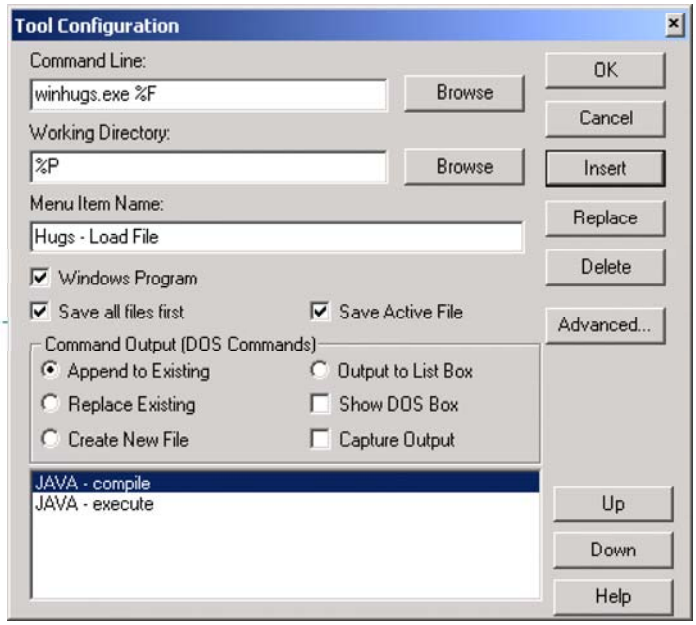
Syntaxhervorhebung

UltraEdit kann automatisch die Haskell-Syntax hervorheben, wie man es von IDEs für andere Sprachen (BlueJ, JBuilder, Visual Studio) gewohnt ist.
Dafür muss die Sprache Haskell in seinem sog. *wordfile* definiert sein. Diese Arbeit muss man sich allerdings nicht selbst machen, ein passender Abschnitt kann heruntergeladen und manuell in die Datei eingefügt werden. Auf der Webseite der Vorlesung liegt eine vorbereitete Datei *HSWORDFILE.TXT*, die man ebenfalls benutzen kann. Die gewünschte Datei wird unter *Advanced* → *Configuration* auf der Seite "Syntax Highlighting" angegeben (siehe nebenstehende Graphik).



HUGS-"Bedienknopf"

Wie die bereits vordefinierten Werkzeuge `Java - compile` und `Java - execute` kann in UltraEdit über *Advanced* → *Tool Configuration* ein Bedienelement definiert werden, mit dem die aktuelle Datei in Hugs geladen wird. Das folgende Bild zeigt die dafür vorzunehmenden Einstellungen:



Mit einem Klick auf "Insert" wird das Bedienelement definiert und kann (nach einem

Klick mit der rechten Maustaste auf die Bedienleiste und Auswahl "Customize") der Bedienleiste hinzugefügt werden.

B.3.2 Emacs

Für Linux-Benutzer steht ein Emacs-Mode zur Verfügung, der eine Syntaxhervorhebung für Haskell-Programme definiert. Daneben kann er auch z.B. automatisch Funktionstypen anzeigen und sinnvolle Einrückungen vorschlagen. Außerdem können Hugs und GHC direkt aus Emacs aufgerufen werden.
Der Haskell-Mode kann unter *http://www.haskell.org/haskell-mode/haskell-mode-1.44.tar.gz* heruntergeladen werden. Seine Installation ist im Archiv ausführlich beschrieben; sie erfolgt durch Anpassung der Datei *$HOME/.emacs* mit den folgenden Zeilen:

```
(setq load-path (cons "/hier/archiv/ausgepackt" load-path))
(setq auto-mode-alist
      (append auto-mode-alist
              '(("\\.[hg]s$"  . haskell-mode)
                ("\\.hi$"     . haskell-mode)
                ("\\.l[hg]s$" . literate-haskell-mode))))
(autoload 'haskell-mode "haskell-mode"
   "Major mode for editing Haskell scripts." t)
(autoload 'literate-haskell-mode "haskell-mode"
   "Major mode for editing literate Haskell scripts." t)
(add-hook 'haskell-mode-hook 'turn-on-haskell-font-lock)
(add-hook 'haskell-mode-hook 'turn-on-haskell-decl-scan)
(add-hook 'haskell-mode-hook 'turn-on-haskell-doc-mode)
(add-hook 'haskell-mode-hook 'turn-on-haskell-indent)
;(add-hook 'haskell-mode-hook 'turn-on-haskell-simple-indent)
(add-hook 'haskell-mode-hook 'turn-on-haskell-hugs)
```

B.3.3 Andere Werkzeuge

Für andere Editoren stehen ebenfalls Erweiterungen für Haskell zur Verfügung, es gibt (angeblich) auch richtige IDEs. Unter *http://www.haskell.org/libraries/#programDevelop* findet sich eine ganze Sammlung, wo sich jede/r sein Lieblingswerkzeug aussuchen kann.