1 Benutzung von Hugs

1.1 Einleitung

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 der einfache `hugs.exe` unter Windows arbeiten dagegen mit Befehlen am Prompt (s.u. Section 1.2).

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

- **Windows:** `M:\2003WS\PraktischeInformatikIII\Hugs98\winhugs.exe`
- **Linux:** `/app/lang/functional/bin/hugs`

Es empfiehlt sich, diesen Pfad der PATH-Variable hinzuzufügen, damit man den Interpreter durch den einfachen Aufruf `hugs` starten kann. Ansonsten muss man stets den gesamten Pfad angeben...

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
```

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 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. `FirstHaskell.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.

1.3 Einfache Beispiele mit Hugs

1.3.1 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 und die elementaren Datentypen einzuführen.

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

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 1: Die wichtigsten Hugs-Kommandos

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

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

1.4 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

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

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

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

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

FirstHaskell>
```

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

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

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

```
FirstHaskell> :r
Reading file "FirstHaskell.hs":
Type checking
ERROR "FirstHaskell.hs":55 - Type error in application
```

```
*** Expression     : simple2 newline 42 "abc"
*** Term           : "abc"
*** Type           : String
*** Does not match : Int

Prelude>
```

1.5 Anderes 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

```
FirstHaskell> :r
Reading file "FirstHaskell.hs":
Parsing
ERROR "FirstHaskell.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.

2 Mal was anderes (?) – 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:** `M:\2003WS\PraktischeInformatikIII\ghc\ghc-6.0.1`
  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.