Freie Universität Berlin

Crashkurs: Haskell

Mentoring WiSe 2017/18

Justus Pfannschmidt
Freie Universität Berlin

27.10.2017

Inhalt

Kommandozeile

	Linux/Mac[1]	*Windows[2]*
1.	Kommandozeile starten nach *Terminal* suchen	 nach *cmd* suchen
2.	Ordnerinhalt anzeigen lassen	
	ls	**dir**
3.	In ein Verzeichnis wechseln	
	cd mein/pfad	**cd** mein\pfad
4.	In das übergeordnete Verzeichnis wechseln	
	cd ..	**cd**..

Haskell installieren & starten

Die Seite https://www.haskell.org/platform/windows.html aufrufen und das Betriebssystem auswählen:

Linux/Mac

1. Distribution bzw. Paket-Manager auswählen
2. Full-Haskell-Paket herunterladen und ausführen oder Befehl in Kommandozeile eingeben
3. Befehl in Kommandozeile eingeben: `ghci`

Windows

1. Full-Haskell-Paket (32 oder 64 Bit) herunterladen
2. den Installer als Admin ausführen (rechte Maustaste, als Administrator ausführen und bestätigen)
3. Die Cabal Konfigurationsdatei modifizieren (Anweisungen auf der Webseite)
4. unter *Programme* oder *Alle Apps* WinGHCi ausführen

Und so sieht es dann aus:

```
GHCi, version 7.10.3: http://www.haskell.org/ghc/  :? for help
Prelude>
```

Ein 1. Haskell-Programm

1. Einen Texteditor öffnen
   (z. B. Notepad++, Kate, Gedit)
2. Folgende Zeilen schreiben

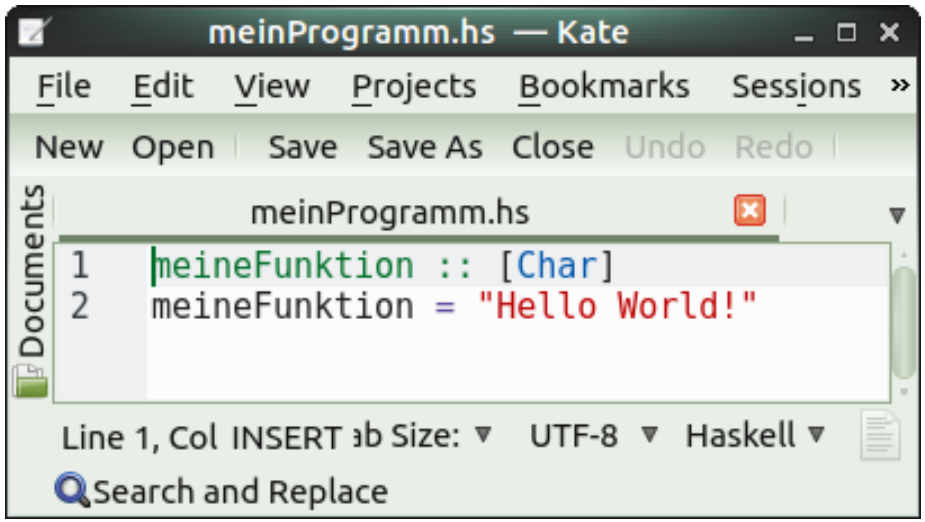
```
1 meineFunktion :: [Char]
2 meineFunktion = "Hello World!"
```

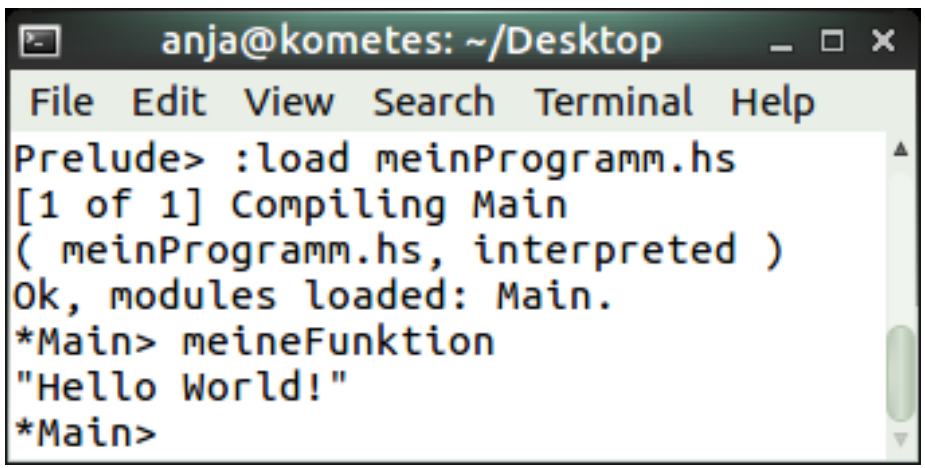
3. Datei als «`meinProgramm.hs`» speichern
4. Zur Kommandozeile wechseln und das Programm laden

```
:load meinProgramm.hs
```

5. Funktion aufrufen

```
meineFunktion
```





Ein 1. Haskell-Programm

1. Änderung im Programm

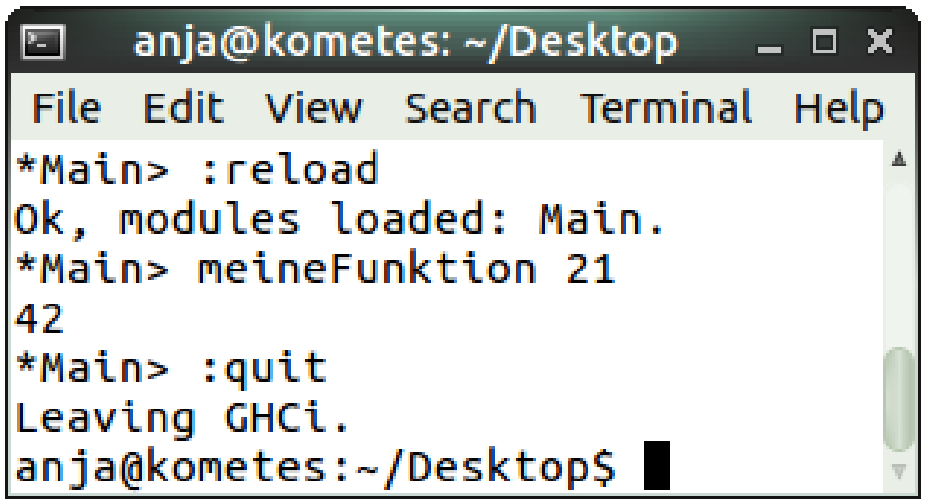
```
meineFunktion :: Int -> Int
meineFunktion n = 2*n
```

2. Programm neu laden

```
:reload
```

3. Interpreter beenden

```
:quit
```



Funktionsdefinition

- Die Funktionsdefinition umfasst sowohl die Signatur (Funktionskopf), als auch den Funktionsrumpf.
- Die Signatur enthält den Funktionsnamen, den Definitions- und Wertebereich.
- Die Funktion $f(x) = x^2$ hat den Definitionsbereich $\mathbb{R}$ und bildet auf alle Zahlen ≥ 0 ab. In der Mathematik schreibt man:

$$f : \mathbb{R} \rightarrow \mathbb{R}$$

- In Haskell ist das ähnlich:

```haskell
f :: Double -> Double  -- Signatur / Funktions-Deklaration
f x = x*x              -- Funktionsrumpf
```

- Der Funktionsrumpf gibt die konkrete Abbildung vom Definitions- zum Wertebereich an.

Primitive Datentypen

Name	**Wertebereich**
`Bool`	{True, False}
`Int`	$\{-2^{63}, \ldots, 2^{63}-1\}$
`Integer`	$\mathbb{Z}$
`Float`	$[-2^{38}, 2^{38}]$, 8 Nachkommastellen
`Double`	$[-2^{308}, 2^{308}]$, 16 Nachkommastellen
`Char`	Zeichen

Vergleiche & Operationen

Vergleiche `==, /=, <, >, >=, <=, compare`

Datentyp	**Operationen**
`Bool`	`not, &&, ||`
`Int, Integer`	`+, -, *, ^, div, mod, odd, ...`
`Float, Double`	`+, -, *, /, **, floor, ceil, sin, sqrt, ...`
`Char`	`fromEnum`
allgemein	`max, min, pred, succ`

Parameterübergabe

Als *currifizierte* Funktion beschreibt man diejenigen, welche ihre Parameter einzeln bekommen:

```
1 volumen :: Double −> Double −> Double −> Double
2 volumen a b c = a∗b∗c
```

Eine *uncurrifizierte* Funktion erhält ein Parameter-Tupel:

```
3 volumen' :: (Double, Double, Double) −> Double
4 volumen' (a,b,c) = a∗b∗c
```

Aufgabe: Schreibt eine Funktion, die entscheidet, ob eine Zahl ganzzahlig durch 2 oder durch 3, nicht aber durch 6 teilbar ist.
Beispiel:

$$\text{test } 15 \Longrightarrow \textit{True}$$

$$\text{test } 12 \Longrightarrow \textit{False}$$

Pause

Typsynonyme

Um sich Schreibarbeit zu ersparen, kann man eigene Datentypen entwerfen, die sich aus primitiven Datentypen zusammensetzen.

```
1  type Punkt = (Double, Double)
2  type Strecke = (Punkt, Punkt)
3  type Dreieck = (Punkt, Punkt, Punkt)

5  umfang :: Dreieck -> Double
6  umfang (p1,p2,p3) = distance p1 p2 + distance p2 p3 + distance p3 p1
7     where
8        distance :: Punkt -> Punkt -> Double
9        distance (x1,y1) (x2,y2) = ...
```

Fallunterscheidung

Wir wollen eine Funktion bauen, die zwei Binärziffern addiert. Dabei müssen wir für die Eingabeparameter **a,b** mehrere Fälle unterscheiden.

if-then-else

```
1  binaryAdd1 :: Int -> Int -> Int
2  binaryAdd1 a b = if a == 0
3                   then b
4                   else if b == 0
5                        then a
6                        else 0
```

Guards

```
7  binaryAdd2 :: Int -> Int -> Int
8  binaryAdd2 a b
9      | a == 0 = b
10     | b == 0 = a
11     | otherwise = 0
```

Pattern-Matching

```
12 binaryAdd3 :: Int -> Int -> Int
13 binaryAdd3 0 0 = 0
14 binaryAdd3 1 1 = 0
15 binaryAdd3 _ _ = 1
```

case-of

```
16 binaryAdd4 :: Int -> Int -> Int
17 binaryAdd4 a b = case (a,b) of
18     (0,0) -> 0
19     (1,1) -> 0
20     _     -> 1
```

Rekursion

Rekursion (Bedeutung: *auf sich rückbezüglich*) besteht aus zwei Teilen:

1. Rekursionsanker
2. Rekursionsschritt (Funktion ruft sich selbst auf)

Folgendes Programm addiert alle geraden Zahlen von $2, \ldots, n$

```
1 addEven :: Int -> Int
2 addEven 0 = 0
3 addEven n
4     | n < 0           = error "negative Eingabe"
5     | mod n 2 == 0    = n + (addEven (n-2))
6     | otherwise       = addEven (n-1)
```

Reduktion:

addEven 5 ⇒ addEven 4
⇒ 4 + (addEven 2) ⇒ 4 + (2+ (addEven 0)) ⇒ 4+(2+0) ⇒ 6

Die Funktion addEven ist *linear-rekursiv*, da sie sich in jedem Rekursionsschritt einmal aufruft.

Ein Beispiel für eine *nicht-lineare* Rekursion ist die **Fibonacci-Folge**:

```haskell
fib :: Int -> Int
fib 0 = 1
fib 1 = 1
fib n = fib (n-1) + fib (n-2) -- hier gibt es zwei Rekursionsaufrufe
```

Aufgabe: Schreibt eine linear-rekursive Funktion, welche die Summe der Quadrate aller geraden Zahlen von $2, 4, \ldots, n$ berechnet.
Beispiel:

$$\text{quadSum } 5 \Longrightarrow (2^2 + 4^2) = 20$$

Listen

Listen sind eine Folge von Elementen des gleichen Datentyps und können beliebig viele Elemente enthalten.

Form	**Beschreibung**	**Beispiele**
`[]`	leere Liste	`[]`
`[x]`	genau ein Element	`[1], ['a'], [True]`
`[x,y]`	genau zwei Elemente	`[0,1]`
`xs`	beliebige Liste	`[], [1,2], [1..]`
`(x:xs)`	mindestens ein Element	`[False]`
`(x1:x2:xs)`	mindestens zwei Elemente	`[1,2,3]`

Wobei die Darstellungen `[1,2,3]`, `(1:2:3:[])` äquivalent sind.

Listen

Folgende Funktion addiert alle Zahlen einer Liste:

```
1 addAll :: [Int] -> Int
2 addAll [] = 0
3 addAll (x:xs) = x + (addAll xs)
```

Reduktion:

```
addAll [1,2] ⇒ 1 + (addAll [2]) ⇒ 1 + (2 + addAll [])
⇒ 1 + (2 + 0) ⇒ 3
```

Man kann die Funktion auch **endrekursiv** schreiben:

```
4 addAll :: [Int] -> Int
5 addAll xs = addAll' xs 0 -- es wird ein Akkumulator übergeben
6   where
7     addAll' :: [Int] -> Int -> Int
8     addAll'  []  akk = akk
9     addAll' (x:xs) akk = addAll' xs (akk+x)
```

Reduktion:

```
addAll [1,2] ⇒ addAll' [1,2] 0
⇒ addAll' [2] (0+1) ⇒ addAll' [] (1+2) ⇒ 3
```

Teamaufgabe

Schreibt eine Funktion, welche die Auszahlung von Wechselgeld mit möglichst wenig Münzen berechnet.

- Dabei rechnen wir in Cent, um Gleitkommazahlen zu vermeiden.
- Mögliche Münzen sind: $1, 2, 5, 10, 20, 50, 100, 200$
- Beispiel: `wechselgeld 249 => [200, 20, 20, 5, 2, 2]`

Achtung: Lösung

```
1 wechselgeld :: Int -> [Int]
2 wechselgeld betrag = helper betrag [200,100,50,20,10,5,2,1] []
3   where
4     helper :: Int -> [Int] -> [Int] -> [Int]
5     helper betrag [] handtasche = handtasche
6     helper betrag (x:xs) handtasche
7       | betrag >= x = helper (betrag-x) (x:xs) (x:handtasche)
8       | otherwise   = helper  betrag     xs    handtasche
```

Quellen

[1] Jann Heider. *Shellbefehle*. Website. 2016. URL: http://www.shellbefehle.de/befehle (besucht am 05. 09. 2016).

[2] Thomas Krenn. *Cmd-Befehle unter Windows*. Website. 2015. URL: https://www.thomas-krenn.com/de/wiki/Cmd-Befehle_unter_Windows (besucht am 05. 09. 2016).

Buchtipp

learnyouahaskell.com

Noch Fragen?