

# Erste Schritte mit Haskell

Vorsemesterkurs  
Sommersemester 2022  
Ronja Düffel

31. März 2022

## RBI-Accounts

- **Achtung:** RBI-Account  $\neq$  HRZ-Account  
RBI: Rechnerbetriebsgruppe Informatik  
HRZ: Hochschulrechenzentrum
- Werden **zu Beginn der Übung von den Tutor:innen** verteilt!
- **Antrag schonmal ausfüllen und unterschreiben!**

# Betriebssysteme

## Bekannte Betriebssysteme

- Microsoft Windows
- Apple OS X
- Linux
- Android, iOS, ...

## Viele Betriebssysteme basieren auf Unix:

- Mehrbenutzer-Betriebssystem
- ursprünglich 1969 in den Bell Labs entwickelt
- Viele moderne Betriebssysteme basieren auf Unix
- Bekanntes Beispiel: GNU Linux

# Terminals / Shells

- Terminal = Schnittstelle zwischen Mensch und Maschine
- Textbasierte Terminals: Kommandos über die Tastatur
- Graphische Terminals (GUIs): Fenster, Tastatur, Maus, ...
- Auf den Rechnern der RBI u.a.:  
Linux Benutzeroberflächen: Gnome und KDE

# Login

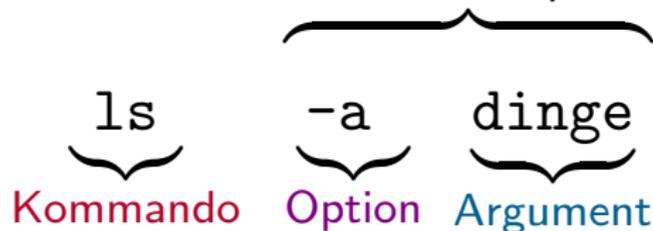
- Login = Anmelden des Benutzers am System
- Benutzername + Passwort
- danach hat man die Kontrolle in einer **Shell**
- oder kann eine solche starten
- Am sog. **prompt** kann man Kommandos eingeben
- Kommando eingeben, danach  („return“) betätigen

```
dallmeyer@kitana ~ █
```

# Shell-Kommandos

## Kommandozeilenparameter

Beispiel:



- **Kommando:** der eigentliche Befehl (ein Programm), im Bsp. `ls` für (list)
- **Optionen:** werden meist durch `-` oder `--` eingeleitet, verändern die Ausführung des Befehls
- **Argumente:** Dateien, Verzeichnisse, Texte auf die das Kommando angewendet wird.

# Einige Kommandos

Aufruf	Erläuterung
<code>echo <i>Text</i></code>	gibt <i>Text</i> aus
<code>whoami</code>	gibt Benutzername aus
<code>hostname</code>	gibt Rechnername aus
<code>pwd</code>	(print working directory) gibt das aktuelle Arbeitsverzeichnis aus
<code>mkdir <i>Verzeichnis</i></code>	(make directory) erzeugt <i>Verzeichnis</i>
<code>cd <i>Verzeichnis</i></code>	(change directory) wechselt in <i>Verzeichnis</i>
<code>cd ..</code>	wechselt ein Verzeichnis nach oben
<code>ls</code>	(list) Anzeigen des Verzeichnisinhalts
<code>man <i>Kommando</i></code>	(manual) Man page zum <i>Kommando</i> anzeigen

# Beispiele

```
> echo "Hallo Welt!" ↵  
Hallo Welt!  
> mkdir dinge ↵  
> ls ↵  
dinge  
> cd dinge ↵  
> ls ↵  
> ls -a ↵  
. ..  
> cd .. ↵  
> mkdir .versteckt ↵  
> ls ↵  
dinge  
> ls -a ↵  
. .. dinge .versteckt
```

In vielen Shells verfügbar:

- Blättern in der **History** (zuletzt eingegebene Befehle) mit  und 
- Auto-Completion mit der  Taste, z.B. 1s  listet alle Befehle auf die mit 1s beginnen
- Auto-Completion mit  und  : versucht die aktuelle Eingabe anhand der History zu vervollständigen.

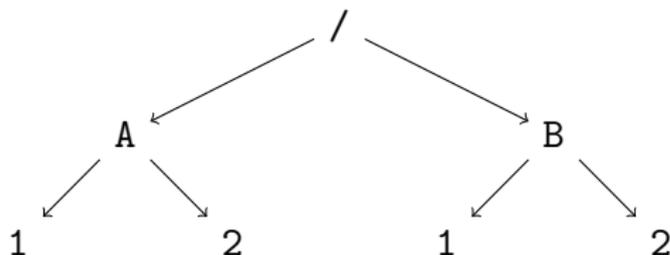
# Dateien und Verzeichnisse

- Jedes Unix-System verwaltet einen **Dateibaum**:  
virtuelles Gebilde zur Datenverwaltung
- Bausteine sind dabei **Dateien** (*files*):  
enthält Daten: Text, Bilder, Maschinenprogramme,...
- Spezielle Dateien: **Verzeichnisse** (*directories*), enthalten selbst wieder Dateien.
- Jede Datei hat einen **Namen**
- Jede Datei befindet sich in einem Verzeichnis, dem **übergeordneten** Verzeichnis
- **Wurzelverzeichnis** / (root directory) ist in sich selbst enthalten.

# Ein Verzeichnisbaum

Beispiel:

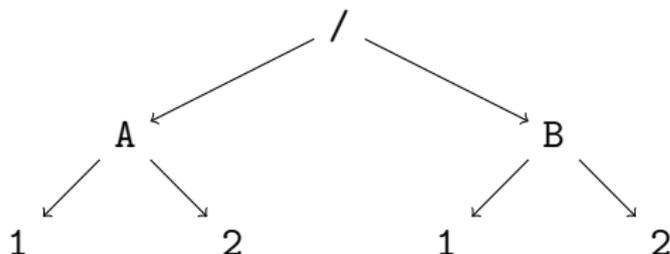
- Wurzelverzeichnis / enthält zwei Verzeichnisse A und B.
- A und B enthalten je ein Verzeichnis mit dem Namen 1 und 2.



# Ein Verzeichnisbaum

Beispiel:

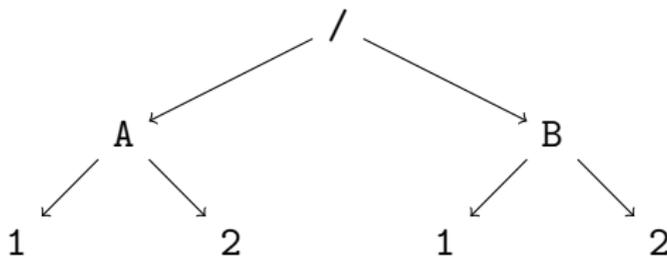
- Wurzelverzeichnis / enthält zwei Verzeichnisse A und B.
- A und B enthalten je ein Verzeichnis mit dem Namen 1 und 2.



```
> tree   
/  
+-- A  
|   +-- 1  
|   +-- 2  
+-- B  
     +-- 1  
     +-- 2
```

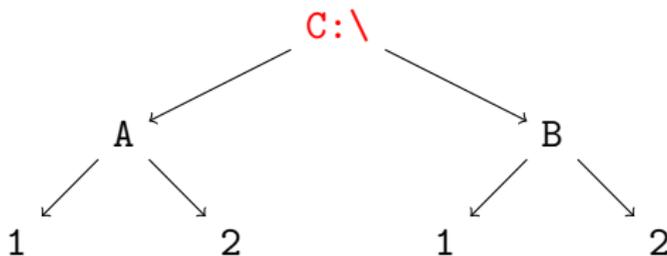
# Relative und absolute Pfade (1)

- **Absoluter Pfad** einer Datei oder eines Verzeichnisses:  
Pfad von der Wurzel beginnend, Verzeichnisse getrennt mit / (slash)
- z.B. /A/1 und /B/1.



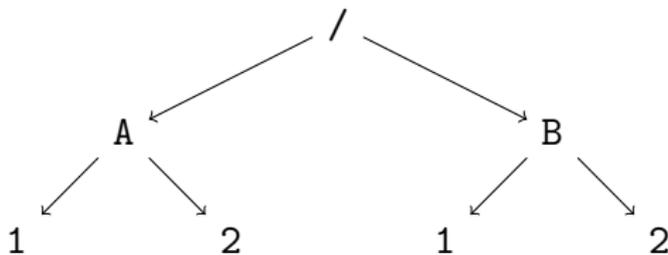
# Relative und absolute Pfade (1)

- **Absoluter Pfad** einer Datei oder eines Verzeichnisses:  
Pfad von der Wurzel beginnend, Verzeichnisse getrennt mit / (slash)
- z.B. /A/1 und /B/1.
- Unter **Windows**: Wurzelverzeichnis ist Laufwerk und Backslash \ statt / z.B. C:\A\1.



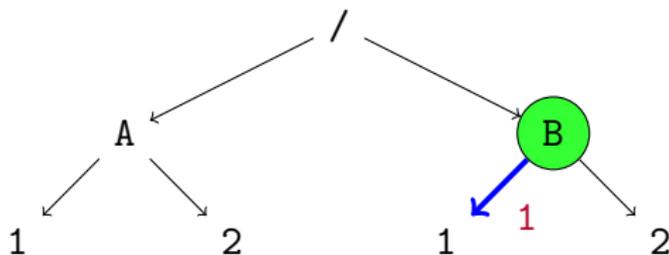
## Relative und absolute Pfade (2)

- **Relative Pfade:** Pfad vom aktuellen Verzeichnis aus, beginnen **nicht** mit /.



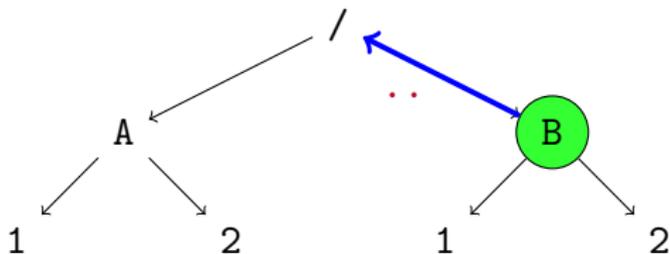
## Relative und absolute Pfade (2)

- **Relative Pfade:** Pfad vom aktuellen Verzeichnis aus, beginnen **nicht** mit /.
- z.B. man ist in /B: Dann bezeichnet 1 das Verzeichnis /B/1.



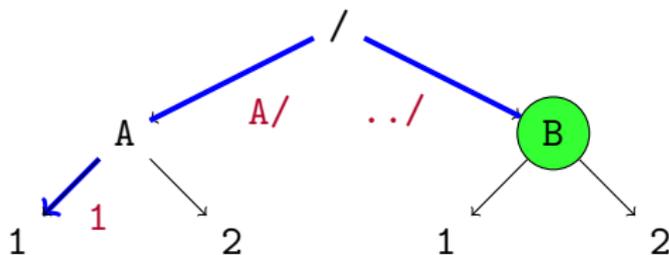
## Relative und absolute Pfade (2)

- **Relative Pfade:** Pfad vom aktuellen Verzeichnis aus, beginnen **nicht** mit /.
- z.B. man ist in /B: Dann bezeichnet 1 das Verzeichnis /B/1.
- .. ist das **übergeordnete** Verzeichnis
- z.B. man ist in /B: Dann bezeichnet .. das Wurzelverzeichnis und ../A/1 bezeichnet /A/1



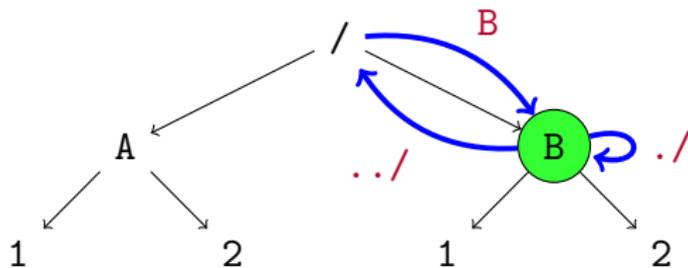
## Relative und absolute Pfade (2)

- **Relative Pfade:** Pfad vom aktuellen Verzeichnis aus, beginnen **nicht** mit /.
- z.B. man ist in /B: Dann bezeichnet 1 das Verzeichnis /B/1.
- .. ist das **übergeordnete** Verzeichnis
- z.B. man ist in /B: Dann bezeichnet .. das Wurzelverzeichnis und ../A/1 bezeichnet /A/1



## Relative und absolute Pfade (2)

- **Relative Pfade:** Pfad vom aktuellen Verzeichnis aus, beginnen **nicht** mit /.
- z.B. man ist in /B: Dann bezeichnet 1 das Verzeichnis /B/1.
- .. ist das **übergeordnete** Verzeichnis
- z.B. man ist in /B: Dann bezeichnet .. das Wurzelverzeichnis und ../A/1 bezeichnet /A/1
- . bezeichnet das **aktuelle** Verzeichnis, z.B. ./../B gleich zu ../B



# Dateien editieren

**Texteditor:** Programm zum Erstellen und Verändern von Textdateien (insbesondere Programmen)

Betriebssystem	Editor
MacOS	TextmateAtom
Windows	Notepad ++
Linux	KateAtom
Alle	VSCode

# Tabulatoren

- Drücken von  erzeugt einen Tabulator
- Zur Einrückung von Text
- Haskell „rechnet intern“ mit 8 Leerzeichen pro Tabulator

Zur Programmierung in Haskell **dringend empfohlen**:

Editor so **einstellen**, dass Tabulatoren  
automatisch durch Leerzeichen ersetzt werden!

## Maschinenprogramme

- bestehen aus Bit-Folgen (0en und 1en),
- Für den Mensch nahezu **unverständlich**
- Verständlicher, aber immer noch zu kleinschrittig: Assemblercode

# Programmiersprachen

## Maschinenprogramme

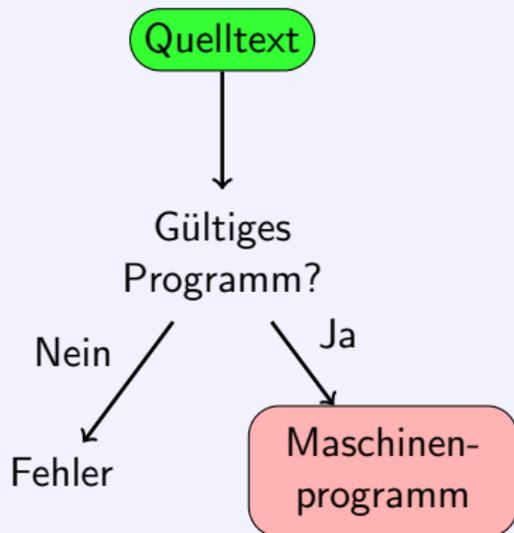
- bestehen aus Bit-Folgen (0en und 1en),
- Für den Mensch nahezu **unverständlich**
- Verständlicher, aber immer noch zu kleinschrittig: Assemblercode

## Höhere Programmiersprachen

- Für den Mensch (meist) verständliche Sprache
- Abstraktere Konzepte, nicht genau am Rechner orientiert
- Der Rechner versteht diese Sprachen **nicht**
- **Quelltext** = Programm in höherer Programmiersprache

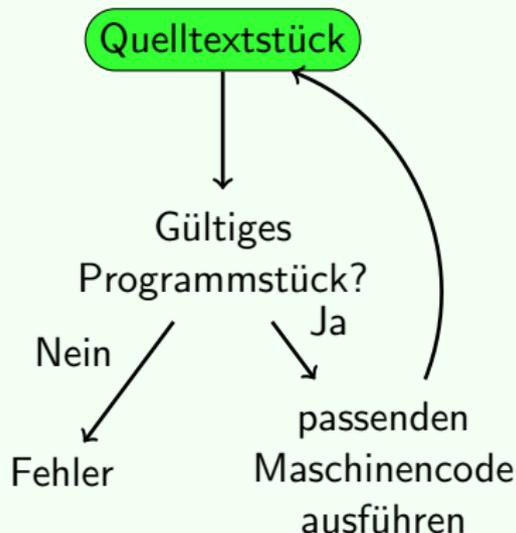
# Compiler / Interpreter

## Compiler



- Langsame Übersetzung auf einmal
- Ausführung schnell

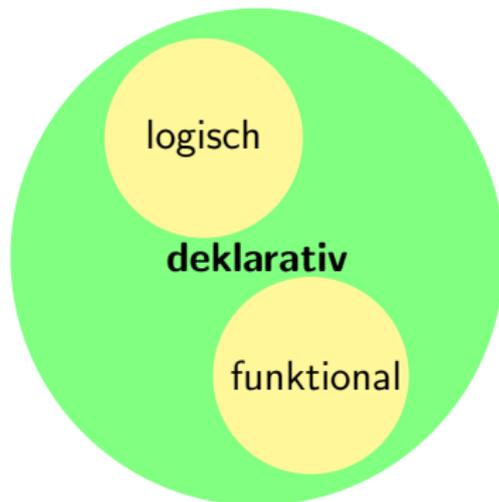
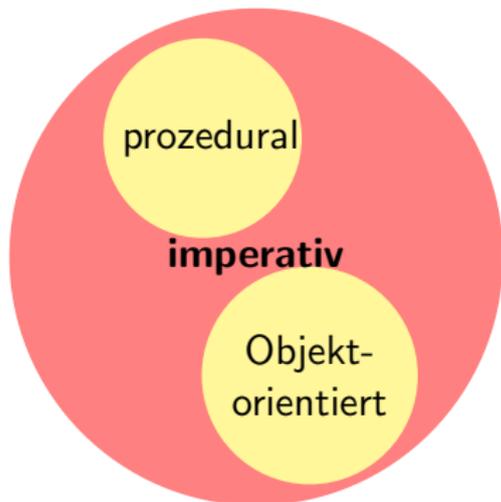
## Interpreter



- Schnelle Übersetzung e. kleinen Stücks
- Ausführung eher langsam

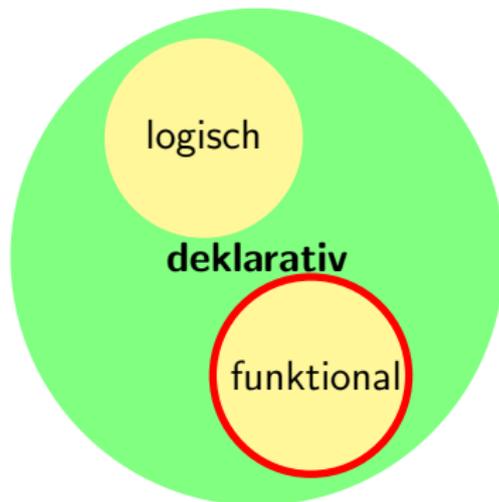
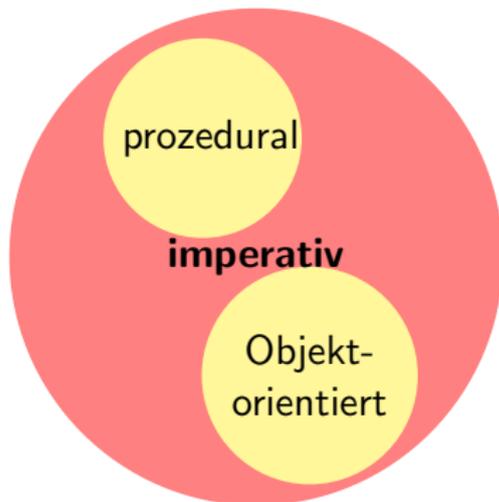
# Programmierparadigmen

Es gibt viele verschiedene höhere Programmiersprachen!



# Programmierparadigmen

Es gibt viele verschiedene höhere Programmiersprachen!



# imperativ vs funktional

	<b>imperativ</b>	<b>funktional</b>
Programm:	Sequenz von Befehlen	Menge von Funktionsdefinitionen
Ausführung:	schrittweise Änderung des Speichers	Auswertung eines Ausdrucks
Resultat:	Geänderter Speicher	ein einziger Wert
“Vorteil”:	u.U. effizient	keine Seiteneffekte (insb. keine Programmvariablen, Zuweisung, Schleifen,...)



# Haskell

- die pure funktionale Programmiersprache
- relativ neu: erster Standard 1990
- Benannt nach dem amerik. Mathematiker *Haskell B. Curry* (1900 - 1982)
- Haskell 98 veröffentlicht 1999, Revision 2003
- Haskell 2010, veröffentlicht Juli 2010

Die Informationsquelle: <http://haskell.org>

# Bedienung des GHCi

```
> ghci 
```

```
GHCi, version 8.0.1: http://www.haskell.org/ghc/ :? for help  
Prelude>
```

# Bedienung des GHCi

```
> ghci ↵  
GHCi, version 8.0.1: http://www.haskell.org/ghc/  :? for help  
Prelude> 1+1 ↵  
2  
Prelude>
```

# Bedienung des GHCi

```
> ghci ↩
GHCi, version 8.0.1: http://www.haskell.org/ghc/  :? for help
Prelude> 1+1 ↩
2
Prelude> 3*4 ↩
12
Prelude> 15-6*3 ↩
-3
Prelude> -3*4 ↩
-12
Prelude> 1+2+3+4+ ↩
<interactive>:1:8: parse error (possibly incorrect indentation)
Prelude> :quit ↩
Leaving GHCi.
>
```

# Einige Interpreterkommandos

- `:quit` Verlassen des Interpreters
- `:help` Der Interpreter zeigt einen Hilfetext an, Übersicht über die verfügbaren Kommandos
- `:load Dateiname` Lädt Haskell-Quellcode der entsprechenden Datei, die Dateiendung von *Dateiname* muss `.hs` oder `.lhs` lauten.
- `:reload` Lädt die aktuelle geladene Datei erneut (hilfreich, wenn man die aktuell geladene Datei im Editor geändert hat).

# Kniffe im GHCi

- Mit `it` (für "es") erhält man das letzte Ergebnis:

```
Prelude> 1+1   
2  
Prelude> it   
2  
Prelude> it+it   
4  
Prelude> it+it   
8
```

- History des GHCi mit Pfeiltasten  und 
- Auto-Completion mit 

## Textdatei

- im Editor erstellen
- Endung: `.hs`

## Zur Erinnerung:

- einen Editor verwenden, **kein** Textverarbeitungsprogramm!
- Editor so einstellen, dass **Tabulatoren durch Leerzeichen** ersetzt werden
- Auf die **Dateiendung** achten

# Beispiel

## Datei hallowelt.hs

```
1 wert = "Hallo Welt!"
```

hallowelt.hs

```
> /opt/rbi/bin/ghci   
GHCi, version 8.0.1: http://www.haskell.org/ghc/  :? for help  
Prelude> :load hallowelt.hs   
[1 of 1] Compiling Main  ( hallowelt.hs, interpreted )  
Ok, modules loaded: Main.  
*Main>
```

funktioniert nur, wenn hallowelt.hs im aktuellen Verzeichnis ist

# Beispiel (2)

Datei hallowelt.hs liegt in **programme/**

```
1 wert = "Hallo Welt!"
```

hallowelt.hs

```
> /opt/rbi/bin/ghci   
GHCi, version 8.0.1: http://www.haskell.org/ghc/  :? for help  
Prelude> :load hallowelt.hs   
  
<no location info>: can't find file: hallowelt.hs  
Failed, modules loaded: none.  
Prelude> :load programme/hallowelt.hs   
[1 of 1] Compiling Main ( programme/hallowelt.hs, interpreted )  
Ok, modules loaded: Main.  
*Main> wert   
"Hallo Welt!"
```

statt erst den `ghci` zu laden und `:load` zu verwenden, geht auch

```
> ghci programme/hallowelt.hs   
GHCi, version 8.0.1: http://www.haskell.org/ghc/  :? for help  
Loading package ghc-prim ... linking ... done.  
Loading package integer-gmp ... linking ... done.  
Loading package base ... linking ... done.  
[1 of 1] Compiling Main ( programme/hallowelt.hs, interpreted )  
Ok, modules loaded: Main.  
*Main>
```

# Nächstes Beispiel

```
1 zwei_mal_Zwei = 2 * 2
2
3 oft_fuenf_addieren = 5 + 5 + 5 + 5 +5 + 5 + 5 + 5 + 5 + 5
4
5 beides_zusammenzaehlen = zwei_mal_Zwei + oft_fuenf_addieren
```

einfacheAusdruecke.hs

```
Prelude> :load einfacheAusdruecke.hs 
*Main> zwei_mal_Zwei 
4
*Main> oft_fuenf_addieren 
55
*Main> beides_zusammenzaehlen 
59
*Main> 3*beides_zusammenzaehlen 
177
```

# Funktionsnamen

müssen mit einem Kleinbuchstaben oder dem Unterstrich `_` beginnen,  
sonst

```
1  
2  
3 Zwei_mal_Zwei = 2 * 2
```

grossKleinschreibungFalsch.hs

```
Prelude> :load grossKleinschreibungFalsch.hs  
[1 of 1] Compiling Main ( grossKleinschreibungFalsch.hs )
```

```
grossKleinschreibungFalsch.hs:3:1:  
    Not in scope: data constructor 'Zwei_mal_Zwei'  
Failed, modules loaded: none.
```

# Kommentare

Eine Quelltext-Datei enthält neben dem Programm:

- Erklärungen und Erläuterungen
- Was macht jede der definierten Funktionen?
- Wie funktioniert die Implementierung?
- Was ist die Idee dahinter? Man sagt auch:

→ **Der Quelltext soll dokumentiert sein!**

Wie kennzeichnet man etwas als Kommentar in Haskell?

- **Zeilenkommentare:** -- Kommentar...
- **Kommentarblöcke:** Durch {- Kommentar -}

# Kommentare: Beispiele

```
wert = "Hallo Welt" -- ab hier ist ein Kommentar bis zum Zeileende  
wert2 = "Nochmal Hallo Welt"  
  
-- Diese ganze Zeile ist auch ein Kommentar!
```

# Kommentare: Beispiele

```
wert = "Hallo Welt" -- ab hier ist ein Kommentar bis zum Zeileende  
  
wert2 = "Nochmal Hallo Welt"  
  
-- Diese ganze Zeile ist auch ein Kommentar!
```

# Kommentare: Beispiele

```
wert = "Hallo Welt" -- ab hier ist ein Kommentar bis zum Zeileende  
wert2 = "Nochmal Hallo Welt"  
  
-- Diese ganze Zeile ist auch ein Kommentar!
```

```
{- Hier steht noch gar keine Funktion,  
   da auch die naechste Zeile noch im  
   Kommentar ist  
  
wert = "Hallo Welt"  
  
   gleich endet der Kommentar -}  
  
wert2 = "Hallo Welt"
```

# Kommentare: Beispiele

```
wert = "Hallo Welt" -- ab hier ist ein Kommentar bis zum Zeileende  
  
wert2 = "Nochmal Hallo Welt"  
  
-- Diese ganze Zeile ist auch ein Kommentar!
```

```
{- Hier steht noch gar keine Funktion,  
   da auch die naechste Zeile noch im  
   Kommentar ist  
  
wert = "Hallo Welt"  
  
   gleich endet der Kommentar -}  
  
wert2 = "Hallo Welt"
```

# Fehler

Beim Programmieren passieren **Fehler**:

- **Syntaxfehler**: Der Quellcode entspricht nicht der Syntax der Programmiersprache. Z.B. falsches Zeichen, fehlende Klammern, falsche Einrückung, ...
- **Logische / Semantische Fehler**: Das Programm implementiert die **falsche** Funktionalität
- **Typfehler**: Der Code ist syntaktisch korrekt, aber die Typen passen nicht, z.B. `1 + 'A'`, etc. (später dazu mehr)

# Fehler (2)

Unterscheiden nach Zeitpunkt des Auftretens

- **Compilezeitfehler:** Fehler, die bereits vom Compiler / Interpreter entdeckt werden und daher in einer Fehlermeldung enden.
- **Laufzeitfehler:** Fehler, die erst zur Laufzeit auftreten und daher nicht vom Compiler/Interpreter schon erkannt werden. Z.B. Division durch 0, Datei lesen, die nicht existiert, etc.

# Haskell und Fehler

- In Haskell werden **viele** Fehler **schon beim Kompilieren** entdeckt
- Z.B.: Keine Typfehler zur Laufzeit
- Der GHCi liefert **Fehlermeldungen** ⇒ **genau lesen!**

```

1 -- 1 und 2 addieren
3 eineAddition = (1+2)
3
4 -- 2 und 3 multiplizieren
5 eineMultiplikation = (2*3))

```

fehler.hs

```

Prelude> :load fehler.hs
[1 of 1] Compiling Main                ( fehler.hs, interpreted )

```

```

fehler.hs:5:27: parse error on input '('
Failed, modules loaded: none.

```

# Programmieren in Haskell

## Haskell-Programmieren:

- Im Wesentlichen formt man **Ausdrücke**
- z.B. arithmetische Ausdrücke  $17*2+5*3$
- Ausführung: Berechnet den Wert eines Ausdrucks

```
*> 17*2+5*3   
49
```

- Ausdrücke **zusammensetzen** durch:

**Anwendung** von Funktionen auf Argumente,  
dabei sind **Werte** die kleinsten „Bauteile“

# Typen

In Haskell hat jeder Ausdruck (und Unterausdruck) einen

## Typ

- Typ = Art des Ausdrucks  
z.B. Buchstabe, Zahl, Liste von Zahlen, Funktion, ...
- Die Typen müssen **zueinander passen**:  
Z.B. **verboten**

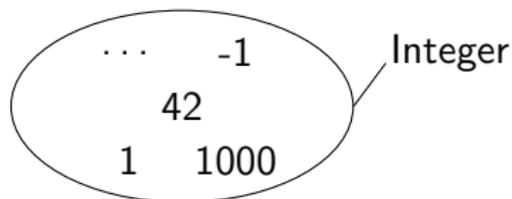
```
1 + "Hallo"
```

Die Typen passen nicht zusammen (Zahl und Zeichenkette)

# Typen (2)

Andere Sichtweise:

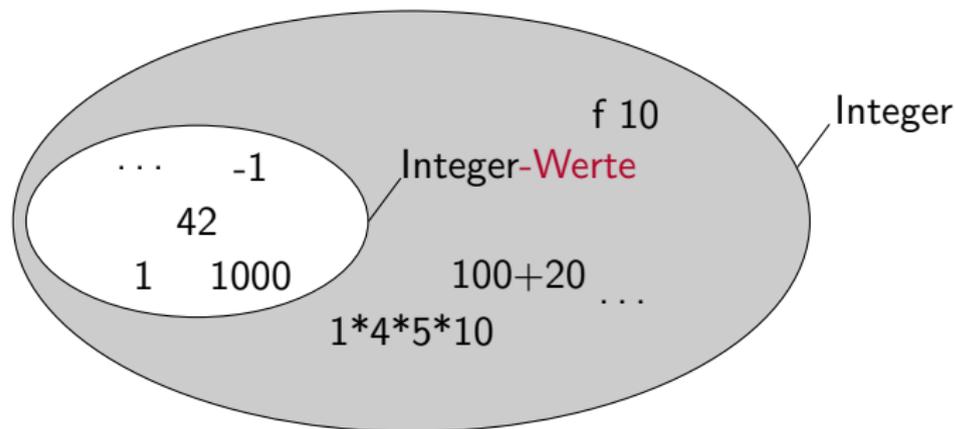
Typ = Menge von Werten



# Typen (2)

Andere Sichtweise:

Typ = Menge von **Werten** **Ausdrücken**



# Typen (3)

Im GHCi Typen anzeigen lassen:

```
Prelude> :type 'C'   
'C' :: Char
```

Sprechweisen:

- „'C' hat den Typ Char“
- „'C' ist vom Typ Char“
- „'C' gehört zum Typ Char“
- Char ist der Typ in Haskell für Zeichen (engl. Character)
- Typnamen beginnen immer mit einem Großbuchstaben

# Typen (4)

- Im GHCi: `:set +t` führt dazu, dass mit jedem Ergebnis auch dessen Typ gedruckt wird.

```
*Main> :set +t ↵
*Main> zwei_mal_Zwei ↵
4
it :: Integer
*Main> oft_fuenf_addieren ↵
55
it :: Integer
```

# Typen (5)

- Der Typ `Integer` stellt beliebig große ganze Zahlen dar
- Man kann Typen auch selbst angeben:

Schreibweise *Ausdruck* :: *Typ*

```
*Main> 'C' :: Char
```

```
'C'
```

```
it :: Char
```

```
*Main> 'C' :: Integer
```

```
<interactive>:1:0:
```

```
Couldn't match expected type 'Integer'  
against inferred type 'Char'
```

```
In the expression: 'C' :: Integer
```

```
In the definition of 'it': it = 'C' :: Integer
```

# Wahrheitswerte: Der Datentyp Bool

Werte (Datenkonstruktoren) vom Typ `Bool`:

- `True` steht für „wahr“
- `False` steht für „falsch“

Basisoperationen (Funktionen):

- Logische Negation: `not`: liefert `True` für `False` und `False` für `True`
- Logisches Und: `a && b`: nur `True`, wenn `a` und `b` zu `True` auswerten
- Logisches Oder: `a || b`: `True`, sobald `a` oder `b` zu `True` ausgewertet.

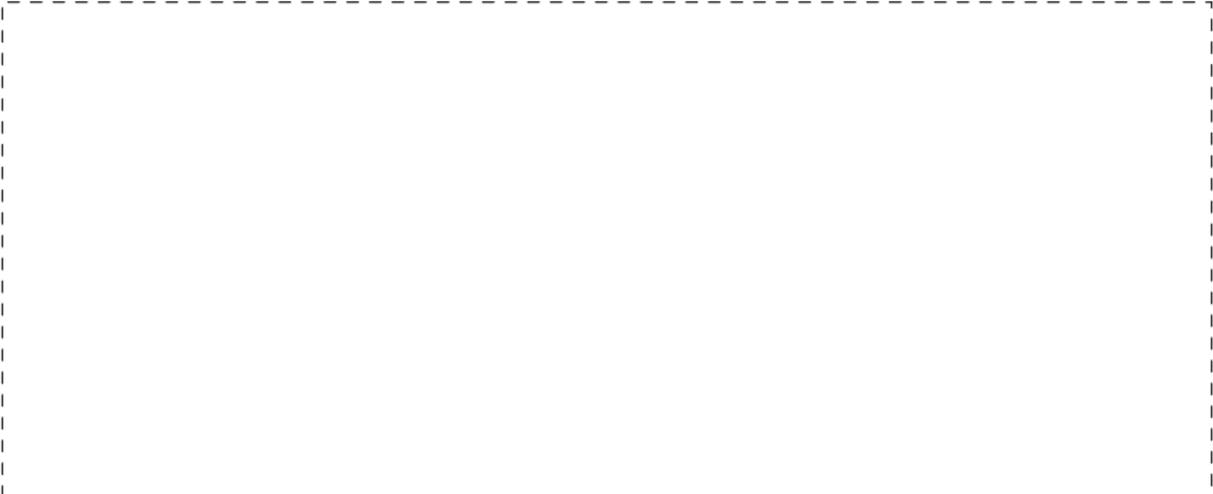
# Wahrheitswerte: Operationen im GHCi ausprobieren

```
*Main> not True   
False  
it :: Bool  
*Main> not False   
True  
it :: Bool  
*Main> True && True   
True  
it :: Bool
```

```
*Main> False && True   
False  
it :: Bool  
*Main> False || False   
False  
it :: Bool  
*Main> True || False   
True  
it :: Bool
```

# Beispiel zur Booleschen Logik

Der Hund sagt: „Die Katze lügt.“ Die Katze sagt: „Der Hund oder die Maus lügen.“  
Die Maus sagt: „Der Hund und die Katze lügen.“ Wer lügt denn nun?



luegelei.hs

# Beispiel zur Booleschen Logik

Der Hund sagt: „Die Katze lügt.“ Die Katze sagt: „Der Hund oder die Maus lügen.“  
Die Maus sagt: „Der Hund und die Katze lügen.“ Wer lügt denn nun?

```
-- Boolesche Variablen: X_luegt fuer X=Hund,Katze,Maus
hund_luegt    = undefined -- True oder False
katze_luegt  = undefined -- True oder False
maus_luegt   = undefined -- True oder False
```

luegelei.hs

# Beispiel zur Booleschen Logik

Der Hund sagt: „Die Katze lügt.“ Die Katze sagt: „Der Hund oder die Maus lügen.“  
Die Maus sagt: „Der Hund und die Katze lügen.“ Wer lügt denn nun?

```
-- Boolesche Variablen: X_luegt fuer X=Hund,Katze,Maus
hund_luegt    = undefined -- True oder False
katze_luegt  = undefined -- True oder False
maus_luegt   = undefined -- True oder False
aussage1 = (not hund_luegt && katze_luegt)
           || (hund_luegt && not katze_luegt)
```

luegelei.hs

# Beispiel zur Booleschen Logik

Der Hund sagt: „Die Katze lügt.“ Die Katze sagt: „Der Hund oder die Maus lügen.“ Die Maus sagt: „Der Hund und die Katze lügen.“ Wer lügt denn nun?

```
-- Boolesche Variablen:  X_luegt fuer X=Hund,Katze,Maus
hund_luegt      = undefined -- True oder False
katze_luegt    = undefined -- True oder False
maus_luegt     = undefined -- True oder False
aussage1 = (not hund_luegt && katze_luegt)
           || (hund_luegt && not katze_luegt)
aussage2 = (not katze_luegt && (hund_luegt || maus_luegt))
           || (katze_luegt && not (hund_luegt || maus_luegt))
```

luegelei.hs

# Beispiel zur Booleschen Logik

Der Hund sagt: „Die Katze lügt.“ Die Katze sagt: „Der Hund oder die Maus lügen.“  
Die Maus sagt: „Der Hund und die Katze lügen.“ Wer lügt denn nun?

```
-- Boolesche Variablen: X_luegt fuer X=Hund,Katze,Maus
hund_luegt    = undefined -- True oder False
katze_luegt  = undefined -- True oder False
maus_luegt   = undefined -- True oder False

aussage1 = (not hund_luegt && katze_luegt)
          || (hund_luegt && not katze_luegt)

aussage2 = (not katze_luegt && (hund_luegt || maus_luegt))
          || (katze_luegt && not (hund_luegt || maus_luegt))

aussage3 = (not maus_luegt && hund_luegt && katze_luegt)
          || (maus_luegt && not (hund_luegt && katze_luegt))
```

luegelei.hs

# Beispiel zur Booleschen Logik

Der Hund sagt: „Die Katze lügt.“ Die Katze sagt: „Der Hund oder die Maus lügen.“  
Die Maus sagt: „Der Hund und die Katze lügen.“ **Wer lügt denn nun?**

```
-- Boolesche Variablen: X_luegt fuer X=Hund,Katze,Maus
hund_luegt    = undefined -- True oder False
katze_luegt  = undefined -- True oder False
maus_luegt   = undefined -- True oder False

aussage1 = (not hund_luegt && katze_luegt)
          || (hund_luegt && not katze_luegt)

aussage2 = (not katze_luegt && (hund_luegt || maus_luegt))
          || (katze_luegt && not (hund_luegt || maus_luegt))

aussage3 = (not maus_luegt && hund_luegt && katze_luegt)
          || (maus_luegt && not (hund_luegt && katze_luegt))

alleAussagen = aussage1 && aussage2 && aussage3
```

luegelei.hs

# Beispiel zur Booleschen Logik

Der Hund sagt: „Die Katze lügt.“ Die Katze sagt: „Der Hund oder die Maus lügen.“  
Die Maus sagt: „Der Hund und die Katze lügen.“ Wer lügt denn nun?

```
-- Boolesche Variablen:  X_luegt fuer X=Hund,Katze,Maus
hund_luegt    = undefined -- True oder False
katze_luegt  = undefined -- True oder False
maus_luegt   = undefined -- True oder False

aussage1 = (not hund_luegt && katze_luegt)
          || (hund_luegt && not katze_luegt)

aussage2 = (not katze_luegt && (hund_luegt || maus_luegt))
          || (katze_luegt && not (hund_luegt || maus_luegt))

aussage3 = (not maus_luegt && hund_luegt && katze_luegt)
          || (maus_luegt && not (hund_luegt && katze_luegt))

alleAussagen = aussage1 && aussage2 && aussage3
```

luegelei.hs

Testen im GHCi ergibt: Nur bei einer Belegung liefert alleAussagen den Wert True: hund\_luegt = True, katze\_luegt = False, maus\_luegt = True

# Fragen?

?