

Vorsemesterkurs Informatik

Goethe-Universität Frankfurt am Main

Sommersemester 2015

Beteiligte Personen

Organisation:

- Ronja Düffel (Ingo Wegener-Lernzentrum)

Vorlesung:

- Teil 1: PD Dr. David Sabel
- Teil 2: Ronja Düffel

Übungen:

- Tutorinnen und Tutoren

Sommersemester laut Studienplan

Anfänger im Sommersemester (1. Semester)

- PRG2: Grundlagen der Programmierung 2 (3V+2Ü, 8CP)
- DS: Datenstrukturen (2V+1Ü, 5CP)
- HW2: Hardware 2 (2V+1Ü, 4CP)
- HWR: Hardwarearchitekturen und Rechensysteme (3V+2Ü, 8CP)
- STO: Einführung in das Studium (2CP)

Anfänger im Wintersemester (2. Semester)

- PRG2: Grundlagen der Programmierung 2 (3V+2Ü, 8CP)
- DS: Datenstrukturen (2V+1Ü, 5CP)
- HWR: Hardwarearchitekturen und Rechensysteme (3V+2Ü, 8CP)
- M2: Mathematik II (4V+2Ü, 9CP)

Ziele des Vorkurses: Vorbereitung auf ...

Grundlagen der Programmierung 2 (PRG-2)

- Funktionale Programmiersprachen (Haskell)
- Compilerbau
- Datenbanken und SQL



Prof. Schmidt-Schauß



Dr. Tolle

Datenstrukturen (DS)

- Entwurf und Analyse von Datenstrukturen
- Asymptotische Notation, Laufzeit- & Platzanalyse
- Effiziente Datenstrukturen



Prof. Schnitger

Inhalte des Vorkurses

Teil 1: (Vorbereitung auf PRG-2)

- Benutzung der Linux-Rechner
- Haskell: Einführung und Ausprobieren

Teil 2: (Vorbereitung auf DS)

- Beweistechniken
- Rekursion und Induktion
- Asymptotik und Laufzeitanalyse

Inhalte des Vorkurses

Teil 1: (Vorbereitung auf PRG-2)

- Benutzung der Linux-Rechner
- Haskell: Einführung und Ausprobieren

Teil 2: (Vorbereitung auf DS)

- Beweistechniken
- Rekursion und Induktion
- Asymptotik und Laufzeitanalyse

Man muss nicht alles (sofort) verstehen!

Aber: Gerne nachfragen!

Auch im Theorieteil durchhalten!

Zeitplan

	Mi 18.3.	Do 19.3.	Fr 20.3.	Mo 23.3.	Di 24.3.	Mi 25.3.
10 – 12	Linux & Haskell	Haskell	Haskell	Beweistechniken	Rekursion & Induktion	Asymptotik & Laufzeitanalyse
12 – 13	Pause	Pause	Pause	Pause	Pause	Pause
13 – 16	Übung	Übung	Übung	Übung	Übung	Übung

Vorlesung: Magnus Hörsaal

Übungen: in Gruppen und verschiedenen Räumen

Fr, 10. April um 11h OE der Fachschaft für Erstsemester

Informationsquellen

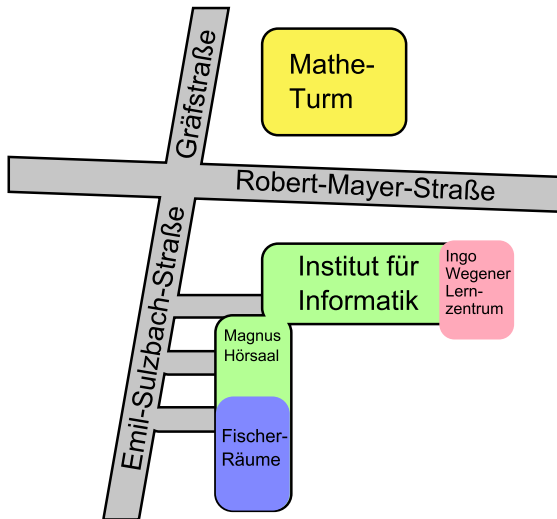
- Material:
 - Skript
 - Folien
 - Übungsaufgaben
- Webseite zum Vorkurs:
vorkurs.informatik.uni-frankfurt.de
- Email für Fragen:
vorkurs@cs.uni-frankfurt.de
- Tutoren/innen
- Dozenten/innen

Übungen

Übungsgruppen

- Aushänge: Gruppen und Räume für die Übungen ab 13:00
- Wer nicht zugeordnet ist, oder sonstiges Problem:
Ronja Düffel fragen (sitzt im Lernzentrum)
- Verschiedene Rechnerräume in der näheren Umgebung:
 - Fischerräume
 - Rechnerraum im Keller
 - Im Lernzentrum
 - Im Seminarraum 11
- Ihr Tutor holt Sie heute im Anschluss an die Vorlesung ab!

Wo ist was?



Rechner-Accounts

RBI-Accounts (wer noch keinen hat)

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

Aufgaben und Lernerfolg

Topics of the day

- morgens vor der Vorlesung ankreuzen und nachmittags nach der Übung nochmal ankreuzen
- bitte **beim Tutor abgeben**
- Rückmeldung ist wichtig, damit **wir** besser werden!

Aufgaben (nachmittags)

- Zwei Aufgabenblätter A und B
- A: Einfache Aufgaben, die jeder bearbeiten können sollte
- B: Eher schwierigere Aufgaben
- Jeder kann **wählen**, ob er A oder B bearbeitet

Shells & Verzeichnisse insbesondere in Linux

Betriebssysteme

Bekannte Betriebssysteme

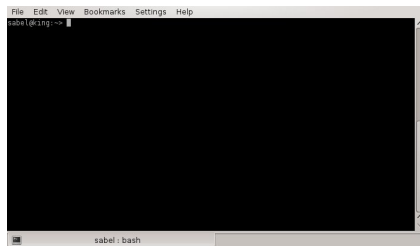
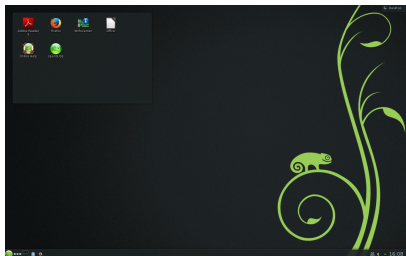
- Microsoft Windows
- Apple OS X
- Linux
- Android, iOS, ...

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

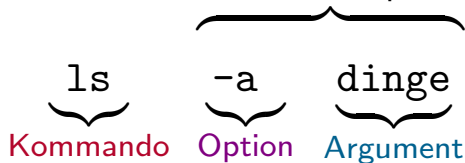
- Login = Anmelden des Benutzers am System
- Benutzername + Passwort
- danach hat man die Kontrolle in einer **Shell**
- oder kann eine solche starten
- Am sog. **prompt** kann man Kommandos eingeben
- Kommando eingeben, danach  („return“) betätigen

```
sabel@king:~> █
```


Shell-Kommandos

Kommandozeilenparameter

Beispiel:


`ls -a dinge`
Kommando Option Argument

- **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
```

Kniffe

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.

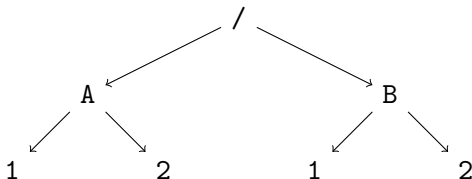
Dateien und Verzeichnisse

- 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:

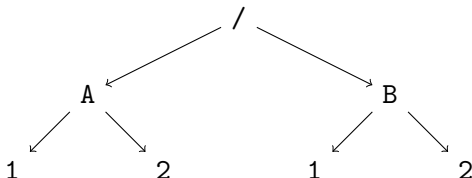
- Wurzelverzeichnis / enthält zwei Verzeichnisse A und B.
- A und B enthalten je ein Verzeichnis mit dem Namen 1 und 2.



Ein Verzeichnisbaum

Beispiel:

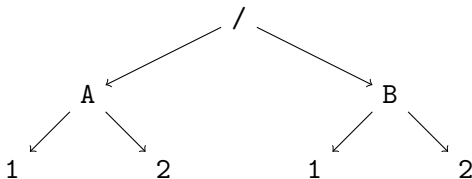
- Wurzelverzeichnis / enthält zwei Verzeichnisse A und B.
- A und B enthalten je ein Verzeichnis mit dem Namen 1 und 2.



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

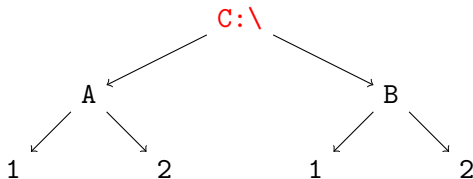
Relative und absolute Pfade (1)

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



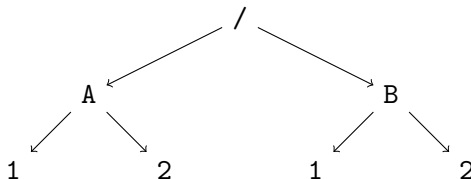
Relative und absolute Pfade (1)

- **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.



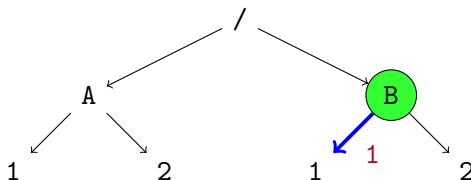
Relative und absolute Pfade (2)

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



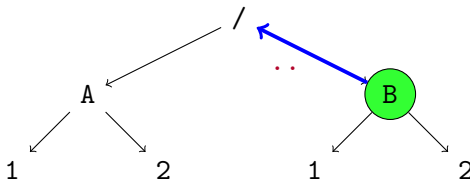
Relative und absolute Pfade (2)

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



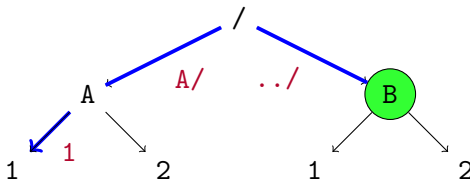
Relative und absolute Pfade (2)

- **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



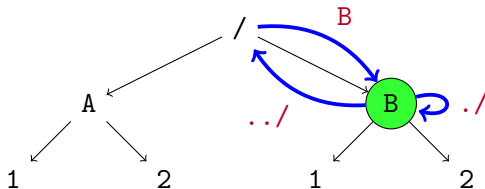
Relative und absolute Pfade (2)

- **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)

Graphische Editoren, z.B.

- **kate** (kate-editor.org/) KDE
- **gedit** (projects.gnome.org/gedit/) Gnome
- **Notepad++** (notepad-plus-plus.org/) für Windows
- **TextWrangler** (barebones.com/products/textwrangler/) für Mac OS X
- **Emacs** (www.gnu.org/software/emacs/)
- **XEmacs** (<http://www.xemacs.org/>)

Textmodus z.B. **vi**

Einige Texteditoren

gedit (Linux, Gnome)

kate (Linux, KDE)

xemacs (Linux)

Notepad++ (MS Windows)

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!

Programmieren, Erste Schritte mit Haskell

Programmiersprachen

Maschinenprogramme

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

Programmierersprachen

Maschinenprogramme

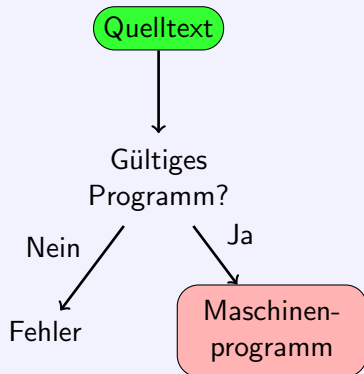
- bestehen aus Bit-Folgen (0en und 1en),
- Für den Mensch nahezu **unverständlich**
- Verständlicher, aber immer noch zu kleinschrittig: Assemblercode

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

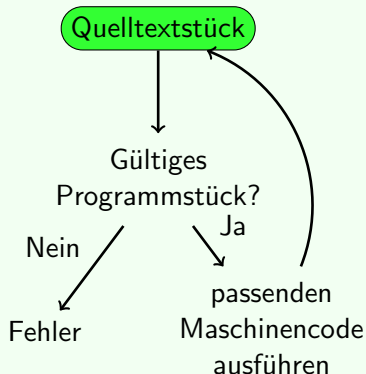
Compiler / Interpreter

Compiler



- Langsame Übersetzung auf einmal
- Ausführung schnell

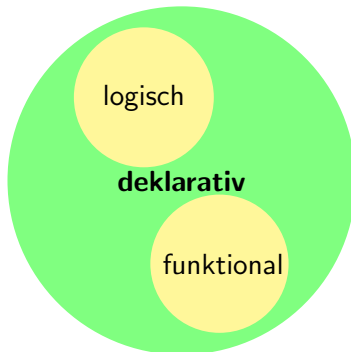
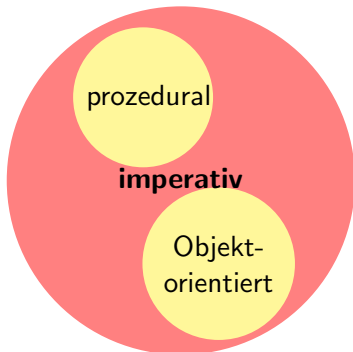
Interpreter



- Schnelle Übersetzung e. kleinen Stücks
- Ausführung eher langsam

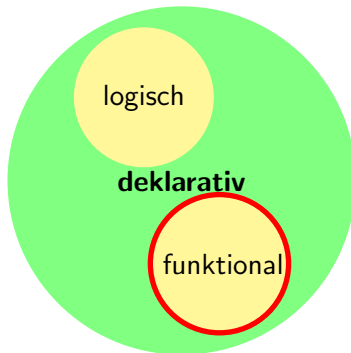
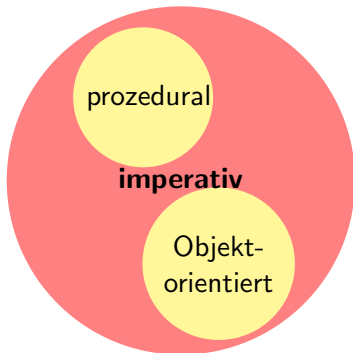
Programmierparadigmen

Es gibt viele verschiedene höhere Programmiersprachen!



Programmierparadigmen

Es gibt viele verschiedene höhere Programmiersprachen!



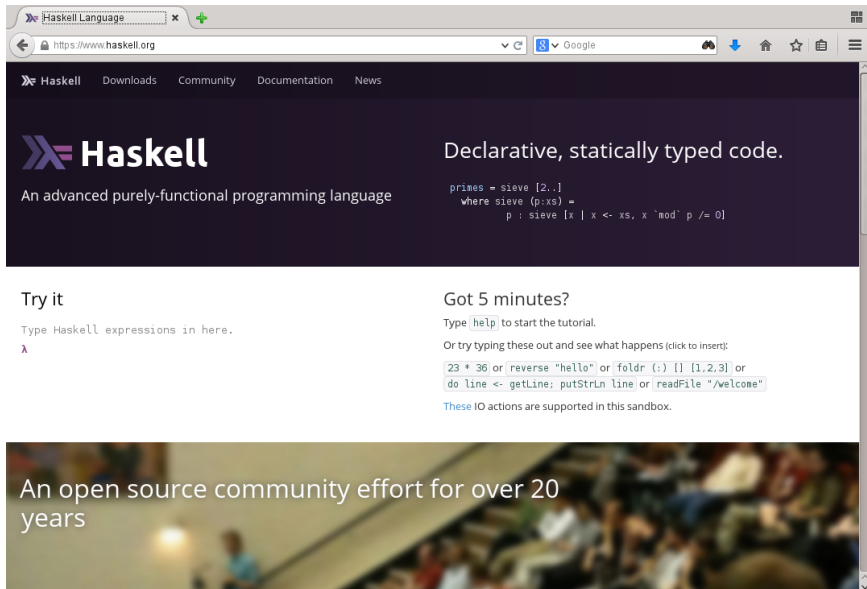
Funktionale Programmiersprachen

- Programm = Menge von Funktionsdefinitionen
- Ausführung = Auswerten eines Ausdrucks
- Resultat = ein einziger Wert
- **keine Seiteneffekte** (sichtbare Speicheränderungen)!
(insbes.: keine Programmvariablen, Zuweisung, ...)



- 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>



The screenshot shows the Haskell Language website in a web browser. The browser's address bar displays "https://www.haskell.org". The website has a dark purple header with navigation links: "Haskell", "Downloads", "Community", "Documentation", and "News". The main content area features the Haskell logo (two crossed arrows) and the text "Haskell" in a large, bold font. Below this, it says "An advanced purely-functional programming language". To the right, the phrase "Declarative, statically typed code." is displayed. A code snippet for a sieve of Eratosthenes is shown:

```
primes = sieve [2..]
  where sieve (p:xs) =
    p : sieve [x | x <- xs, x `mod` p /= 0]
```

 Under the heading "Try it", there is a text input field with the placeholder "Type Haskell expressions in here." and a small red lambda symbol. To the right, under "Got 5 minutes?", there is a text input field with the placeholder "Type `help` to start the tutorial." and a list of example expressions: `23 * 36`, `reverse "hello"`, `foldr (:) [] [1,2,3]`, `do line <- getLine; putStrLn line`, and `readFile "welcome"`. Below these, it says "These IO actions are supported in this sandbox." At the bottom of the page, there is a banner image of a crowd of people with the text "An open source community effort for over 20 years".

Haskell Language

https://www.haskell.org

Haskell Downloads Community Documentation News

Haskell

An advanced purely-functional programming language

Declarative, statically typed code.

```
primes = sieve [2..]
  where sieve (p:xs) =
    p : sieve [x | x <- xs, x `mod` p /= 0]
```

Try it

Type Haskell expressions in here.

λ

Got 5 minutes?

Type `help` to start the tutorial.

Or try typing these out and see what happens (click to insert):

```
23 * 36 or reverse "hello" or foldr (:) [] [1,2,3] or
do line <- getLine; putStrLn line or readFile "welcome"
```

These IO actions are supported in this sandbox.

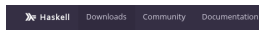
An open source community effort for over 20 years

GHC und GHCi

- Wir verwenden den **G**lasgow **H**askell **C**ompiler, bzw. den **I**nterpreter dazu: **GHCi**
- Auf den RBI-Rechnern: Kommando `ghci` bzw. `/opt/rbi/bin/ghci`

GHC und GHCi

- Wir verwenden den **G**lasgow **H**askell **C**ompiler, bzw. den **I**nterpreter dazu: **GHCi**
- Auf den RBI-Rechnern: Kommando `ghci` bzw. `/opt/rbi/bin/ghci`
- Selbst-Installieren: Am besten die **Haskell Platform**:
<http://hackage.haskell.org/platform/>



Downloads

Compiler and base libraries

Downloads are available on a per operating system basis:

- Windows
- OS X
- Linux

Haskell Platform

Many now recommend just using a bare compiler combined with new users. However, others prefer to start with the curated release which is available for Windows, OS X, and Linux.

[Get the Haskell Platform →](#)



Bedienung des GHCi

```
> ghci 
```

```
GHCi, version 7.6.3: http://www.haskell.org/ghc/  :? for help  
Prelude>
```

Bedienung des GHCi

```
> ghci   
GHCi, version 7.6.3: http://www.haskell.org/ghc/  :? for help  
Prelude> 1+1   
2  
Prelude>
```

Bedienung des GHCi

```
> ghci ↩
GHCi, version 7.6.3: 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

<code>:quit</code>	Verlassen des Interpreters
<code>:help</code>	Der Interpreter zeigt einen Hilfetext an, Übersicht über die verfügbaren Kommandos
<code>:load <i>Dateiname</i></code>	Lädt Haskell-Quellcode der entsprechenden Datei, die Dateierendung von <i>Dateiname</i> muss <code>.hs</code> lauten.
<code>:reload</code>	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 

Haskell-Quellcode

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 7.6.3: 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 7.6.3: 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!"
```

Kniff

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

```
> ghci programme/hallowelt.hs   
GHCi, version 7.6.3: 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 Compilieren** entdeckt
- Z.B.: Keine Typfehler zur Laufzeit
- Der GHCi liefert **Fehlermeldungen** $\Rightarrow$ **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 mit Haskell

Ausdrücke und Typen

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**

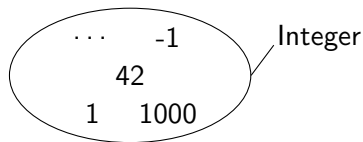
`1 + "Hallo"`

Die Typen passen nicht zusammen (Zahl und Zeichenkette)

Typen (2)

Andere Sichtweise:

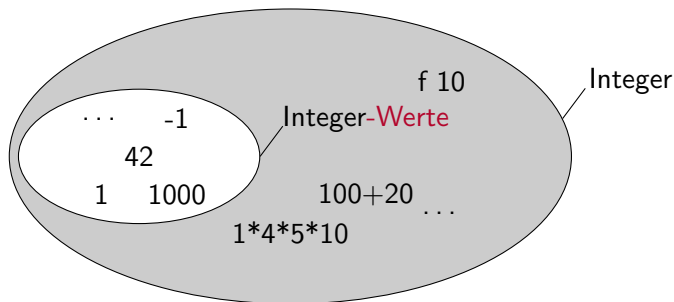
Typ = Menge von Werten



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

Anna sagt: „Bettina lügt.“ Bettina sagt: „Anna oder Claudia lügen.“ Claudia sagt: „Anna und Bettina lügen.“ Wer lügt denn nun?

luegelei.hs

Beispiel zur Booleschen Logik

Anna sagt: „Bettina lügt.“ Bettina sagt: „Anna oder Claudia lügen.“ Claudia sagt: „Anna und Bettina lügen.“ Wer lügt denn nun?

```
-- Boolesche Variablen:  X_luegt fuer X=anna,bettina,claudia  
anna_luegt      = undefined -- True oder False  
bettina_luegt   = undefined -- True oder False  
claudia_luegt   = undefined -- True oder False
```

luegelei.hs

Beispiel zur Booleschen Logik

Anna sagt: „Bettina lügt.“ Bettina sagt: „Anna oder Claudia lügen.“ Claudia sagt: „Anna und Bettina lügen.“ Wer lügt denn nun?

```
-- Boolesche Variablen:  X_luegt fuer X=anna,bettina,claudia  
  
anna_luegt    = undefined -- True oder False  
bettina_luegt = undefined -- True oder False  
claudia_luegt = undefined -- True oder False  
  
aussage1 = (not anna_luegt && bettina_luegt)  
          || (anna_luegt && not bettina_luegt)
```

luegelei.hs

Beispiel zur Booleschen Logik

Anna sagt: „Bettina lügt.“ Bettina sagt: „Anna oder Claudia lügen.“ Claudia sagt: „Anna und Bettina lügen.“ Wer lügt denn nun?

```
-- Boolesche Variablen:  X_luegt fuer X=anna,bettina,claudia
anna_luegt      = undefined -- True oder False
bettina_luegt   = undefined -- True oder False
claudia_luegt   = undefined -- True oder False
aussage1 = (not anna_luegt && bettina_luegt)
           || (anna_luegt && not bettina_luegt)
aussage2 = (not bettina_luegt && (anna_luegt || claudia_luegt))
           || (bettina_luegt && not (anna_luegt || claudia_luegt))
```

luegelei.hs

Beispiel zur Booleschen Logik

Anna sagt: „Bettina lügt.“ Bettina sagt: „Anna oder Claudia lügen.“ Claudia sagt: „Anna und Bettina lügen.“ Wer lügt denn nun?

```
-- Boolesche Variablen:  X_luegt fuer X=anna,bettina,claudia

anna_luegt    = undefined -- True oder False
bettina_luegt = undefined -- True oder False
claudia_luegt = undefined -- True oder False

aussage1 = (not anna_luegt && bettina_luegt)
          || (anna_luegt && not bettina_luegt)

aussage2 = (not bettina_luegt && (anna_luegt || claudia_luegt))
          || (bettina_luegt && not (anna_luegt || claudia_luegt))

aussage3 = (not claudia_luegt && anna_luegt && bettina_luegt)
          || (claudia_luegt && not (anna_luegt && bettina_luegt))
```

luegelei.hs

Beispiel zur Booleschen Logik

Anna sagt: „Bettina lügt.“ Bettina sagt: „Anna oder Claudia lügen.“ Claudia sagt: „Anna und Bettina lügen.“ **Wer lügt denn nun?**

```
-- Boolesche Variablen:  X_luegt fuer X=anna,bettina,claudia  
  
anna_luegt    = undefined -- True oder False  
bettina_luegt = undefined -- True oder False  
claudia_luegt = undefined -- True oder False  
  
aussage1 = (not anna_luegt && bettina_luegt)  
          || (anna_luegt && not bettina_luegt)  
  
aussage2 = (not bettina_luegt && (anna_luegt || claudia_luegt))  
          || (bettina_luegt && not (anna_luegt || claudia_luegt))  
  
aussage3 = (not claudia_luegt && anna_luegt && bettina_luegt)  
          || (claudia_luegt && not (anna_luegt && bettina_luegt))  
  
alleAussagen = aussage1 && aussage2 && aussage3
```

luegelei.hs

Beispiel zur Booleschen Logik

Anna sagt: „Bettina lügt.“ Bettina sagt: „Anna oder Claudia lügen.“ Claudia sagt: „Anna und Bettina lügen.“ Wer lügt denn nun?

```
-- Boolesche Variablen: X_luegt fuer X=anna,bettina,claudia
anna_luegt      = undefined -- True oder False
bettina_luegt   = undefined -- True oder False
claudia_luegt   = undefined -- True oder False

aussage1 = (not anna_luegt && bettina_luegt)
          || (anna_luegt && not bettina_luegt)

aussage2 = (not bettina_luegt && (anna_luegt || claudia_luegt))
          || (bettina_luegt && not (anna_luegt || claudia_luegt))

aussage3 = (not claudia_luegt && anna_luegt && bettina_luegt)
          || (claudia_luegt && not (anna_luegt && bettina_luegt))

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

luegelei.hs

Testen im GHCi ergibt: Nur bei einer Belegung liefert alleAussagen den Wert True: anna_luegt = True, bettina_luegt = False, claudia_luegt = True

Ganze Zahlen: Int und Integer

- Der Typ **Int** umfasst ganze Zahlen **beschränkter** Größe
- Der Typ **Integer** umfasst ganze Zahlen **beliebiger** Größe

Ganze Zahlen: Int und Integer

- Der Typ **Int** umfasst ganze Zahlen **beschränkter** Größe
- Der Typ **Integer** umfasst ganze Zahlen **beliebiger** Größe
- Darstellung der Zahlen ist identisch z.B. 1000
- Defaulting: Integer, wenn es nötig ist, sonst offenlassen:

Ganze Zahlen: Int und Integer

- Der Typ **Int** umfasst ganze Zahlen **beschränkter** Größe
- Der Typ **Integer** umfasst ganze Zahlen **beliebiger** Größe
- Darstellung der Zahlen ist identisch z.B. 1000
- Defaulting: Integer, wenn es nötig ist, sonst offenlassen:

```
Prelude> :type 1000
1000 :: (Num t) => t
Prelude> :set +t
Prelude> 1000
1000
it :: Integer
```

- In etwa 1000 ist vom Typ **t**, wenn **t** ein **numerischer** Typ ist.
- Genauer: $(\text{Num } t) \Rightarrow$ ist eine sog. **Typklassenbeschränkung**
- Int und Integer sind numerische Typen (haben Instanzen für Num)

Gleitkommazahlen

- Typen `Float` und `Double` (mit doppelter Genauigkeit)
- Kommastelle wird mit `.` (Punkt) dargestellt
- Typklasse dazu: `Fractional`

```
Prelude> :type 10.5   
10.5 :: (Fractional t) => t  
Prelude> 10.5   
10.5  
it :: Double
```

- Beachte: Das Rechnen mit solchen Kommazahlen ist **ungenau**!

Zeichen und Zeichenketten

- Der Typ **Char** repräsentiert Zeichen
Darstellung: Zeichen in einfache Anführungszeichen, z.B. 'A'
- Spezielle Zeichen (Auswahl):

Darstellung in Haskell	Zeichen, das sich dahinter verbirgt
'\\'	Backslash \
'\''	einfaches Anführungszeichen '
'\"'	doppeltes Anführungszeichen "
'\n'	Zeilenumbruch
'\t'	Tabulator

- Zeichenketten: Typ **String**
Darstellung in doppelten Anführungszeichen, z.B. "Hallo".
- Genauer ist der Typ **String** gleich zu **[Char]**
d.h. eine Liste von Zeichen (Listen behandeln wir später)

Beispiel Zeichenketten

```
> :set +t
> "Ein \n\'mehrzeiliger\'\'nText mit \"Anfuehrungszeichen\""
"Ein \n\'mehrzeiliger\'\'nText mit \"Anfuehrungszeichen\""
it :: [Char]
> putStrLn "Ein \n\'mehrzeiliger\'\'nText mit \"Anfuehrungszeichen\""
Ein
\'mehrzeiliger\'
Text mit "Anfuehrungszeichen"
it :: ()
```

Operatoren auf Zahlen

- Operatoren auf Zahlen in Haskell:
Addition $+$, Subtraktion $-$, Multiplikation $*$ und Division $/$
- Beispiele: $3 * 6$, $10.0 / 2.5$, $4 + 5 * 4$
- Beim $-$ muss man aufpassen, da es auch für negative Zahlen benutzt wird

```
Prelude> 2 * -2 
```

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

```
  Precedence parsing error
```

```
    cannot mix '*' [infixl 7] and prefix '-' [infixl 6]  
    in the same infix expression
```

```
Prelude> 2 * (-2) 
```

```
-4
```

Vergleichsoperationen

Gleichheitstest `==` und
Ungleichheitstest `/=`

```
Prelude> 1 == 3   
False  
Prelude> 3*10 == 6*5   
True  
Prelude> True == False   
False  
Prelude> False == False   
True  
Prelude> 2*8 /= 64   
True  
Prelude> 2+8 /= 10   
False  
Prelude> True /= False   
True
```

größer: `>`, größer oder gleich: `>=`
kleiner: `<`, kleiner oder gleich: `<=`

```
Prelude> 5 >= 5   
True  
Prelude> 5 > 5   
False  
Prelude> 6 > 5   
True  
Prelude> 4 < 5   
True  
Prelude> 4 < 4   
False  
Prelude> 4 <= 4   
True
```


Programmieren mit Haskell

Funktionen

Was ist eine Funktion?

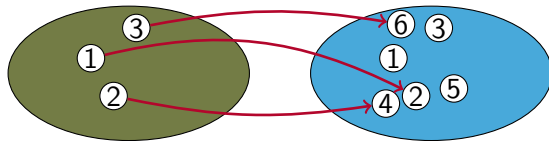
Definition (mathematisch):

Seien D und Z Mengen. Eine **Funktion**

$$f : D \rightarrow Z$$

ordnet jedem Element $x \in D$ ein Element $y \in Z$ zu.

D nennt man den **Definitionsbereich** und Z den **Zielbereich**.



Was ist eine Funktion?

Definition (mathematisch):

Seien D und Z Mengen. Eine **Funktion**

$$f : D \rightarrow Z$$

ordnet jedem Element $x \in D$ ein Element $y \in Z$ zu.

D nennt man den **Definitionsbereich** und Z den **Zielbereich**.

Beispiel:

$$\begin{array}{ccc} \text{verdopple} : & \underbrace{\mathbb{Z}} & \rightarrow \underbrace{\mathbb{Z}} \\ & \text{Definitionsbereich} & \text{Zielbereich} \\ \text{verdopple}(x) & = & x + x \end{array}$$

Was ist eine Funktion?

Definition (mathematisch):

Seien D und Z Mengen. Eine **Funktion**

$$f : D \rightarrow Z$$

ordnet jedem Element $x \in D$ ein Element $y \in Z$ zu.

D nennt man den **Definitionsbereich** und Z den **Zielbereich**.

Beispiel:

$$\begin{array}{ccc} \text{verdopple} : & \underbrace{\mathbb{Z}}_{\text{Definitionsbereich}} & \rightarrow \underbrace{\mathbb{Z}}_{\text{Zielbereich}} \\ & \text{verdopple}(x) = x + x \end{array}$$

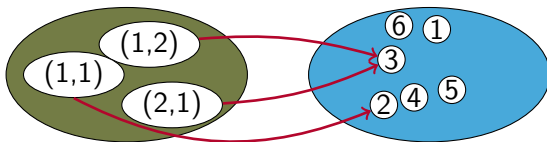
In Haskell:

```
verdopple :: Integer -> Integer
verdopple x = x + x
```

Mehrstellige Funktionen

Möglichkeit 1: Definitionsbereich ist eine Menge von **Tupeln**

$$\begin{array}{ccc}
 \text{summiere} : & \underbrace{(\mathbb{Z} \times \mathbb{Z})}_{\text{Definitionsbereich}} & \rightarrow \underbrace{\mathbb{Z}}_{\text{Zielbereich}} \\
 \text{summiere}(x, y) & = & x + y
 \end{array}$$

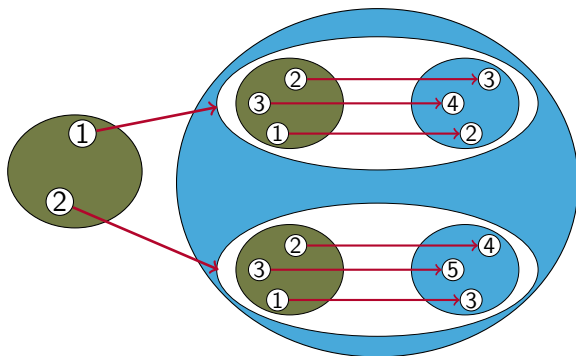


Verwendung: *summiere*(1,2)

Mehrstellige Funktionen(2)

Möglichkeit 2: (Currying) Statt Funktion mit n -Argumenten, Funktion mit einem Argument, Rückgabe: ist Funktion!

$$\begin{array}{ccc}
 \text{summiere} : & \underbrace{\mathbb{Z}}_{\text{Definitionsbereich}} & \rightarrow \underbrace{(\mathbb{Z} \rightarrow \mathbb{Z})}_{\text{Zielbereich}} \\
 \text{summiere } x \ y = & x + y &
 \end{array}$$



Mehrstellige Funktionen (3)

Möglichkeit 2: (Currying) Statt Funktion mit n -Argumenten, Funktion mit einem Argument, Rückgabe: ist Funktion!

$$\begin{array}{ccc}
 \text{summiere} : & \underbrace{\mathbb{Z}}_{\text{Definitionsbereich}} & \rightarrow \underbrace{(\mathbb{Z} \rightarrow \mathbb{Z})}_{\text{Zielbereich}} \\
 & & \text{summiere } x \ y = x + y
 \end{array}$$

- Vorteil ggü. Variante 1: Man kann **partiell anwenden**, z.B. *summiere 2*
- Variante 2 wird in Haskell fast immer verwendet!

```
summiere :: Integer -> (Integer -> Integer)
summiere x y = x + y
```

Mehrstellige Funktionen (3)

Möglichkeit 2: (Currying) Statt Funktion mit n -Argumenten, Funktion mit einem Argument, Rückgabe: ist Funktion!

$$\begin{array}{ccc} \text{summiere} : & \underbrace{\mathbb{Z}}_{\text{Definitionsbereich}} & \rightarrow \underbrace{(\mathbb{Z} \rightarrow \mathbb{Z})}_{\text{Zielbereich}} \\ \text{summiere } x \ y = & x + y & \end{array}$$

- Vorteil ggü. Variante 1: Man kann **partiell anwenden**, z.B. *summiere 2*
- Variante 2 wird in Haskell fast immer verwendet!

```
summiere :: Integer -> (Integer -> Integer)
summiere x y = x + y
```

$\rightarrow$ in Haskell ist rechts-assoziativ: $a \rightarrow b \rightarrow c = a \rightarrow (b \rightarrow c)$
Daher kann man Klammern weglassen.

Mehrstellige Funktionen (3)

Möglichkeit 2: (Currying) Statt Funktion mit n -Argumenten, Funktion mit einem Argument, Rückgabe: ist Funktion!

$$\begin{array}{ccc}
 \text{summiere} : & \underbrace{\mathbb{Z}}_{\text{Definitionsbereich}} & \rightarrow \underbrace{(\mathbb{Z} \rightarrow \mathbb{Z})}_{\text{Zielbereich}} \\
 & & \\
 & \text{summiere } x \ y = x + y
 \end{array}$$

- Vorteil ggü. Variante 1: Man kann **partiell anwenden**, z.B. *summiere 2*
- Variante 2 wird in Haskell fast immer verwendet!

```
summiere :: Integer -> Integer -> Integer
summiere x y = x + y
```

$\rightarrow$ in Haskell ist rechts-assoziativ: $a \rightarrow b \rightarrow c = a \rightarrow (b \rightarrow c)$
Daher kann man Klammern weglassen.

Funktionstypen

Einfacher: f erwartet n Eingaben, dann ist der Typ von f :

$$f :: \underbrace{\text{Typ}_1}_{\substack{\text{Typ des} \\ \text{1. Arguments}}} \rightarrow \underbrace{\text{Typ}_2}_{\substack{\text{Typ des} \\ \text{2. Arguments}}} \rightarrow \dots \rightarrow \underbrace{\text{Typ}_n}_{\substack{\text{Typ des} \\ \text{n. Arguments}}} \rightarrow \underbrace{\text{Typ}_{n+1}}_{\substack{\text{Typ des} \\ \text{Ergebnisses}}}$$

- $\rightarrow$ in Funktionstypen ist **rechts-geklammert**
- $(\&\&) :: \text{Bool} \rightarrow \text{Bool} \rightarrow \text{Bool}$
entspricht $(\&\&) :: \text{Bool} \rightarrow (\text{Bool} \rightarrow \text{Bool})$
und **nicht** $(\&\&) :: (\text{Bool} \rightarrow \text{Bool}) \rightarrow \text{Bool}$

Funktionen: Beispiele

Beispiel: `mod` und `div`

- `mod`: Rest einer Division mit Rest
- `div`: Ganzzahliger Anteil der Division mit Rest

Funktionen: Beispiele

Beispiel: mod und div

- **mod**: Rest einer Division mit Rest
- **div**: Ganzzahliger Anteil der Division mit Rest

```
Prelude> mod 10 3
```



```
1
```

```
Prelude> div 10 3
```



```
3
```

```
Prelude> mod 15 5
```



```
0
```

```
Prelude> div 15 5
```



```
3
```

```
Prelude> (div 15 5) + (mod 8 6)
```



```
5
```

Präfix und Infix

- `+`, `*`, ... werden **infix** verwendet:
 zwischen den Argumenten, z.B. `5+6`
- `mod`, `div` werden **präfix** verwendet:
 vor den Argumenten, z.B. `mod 10 3`

Präfix und Infix

- $+$, $*$, ... werden **infix** verwendet:
zwischen den Argumenten, z.B. $5+6$
- `mod`, `div` werden **präfix** verwendet:
vor den Argumenten, z.B. `mod 10 3`
- Präfix-Operatoren infix verwenden:
In Hochkommata setzen ( + )
z.B. $10 \text{ 'mod' } 3$
- Infix-Operatoren präfix verwenden:
In runde Klammern setzen
z.B. $(+) 5 6$

Boolesche Funktionen: Beispiele

```
Prelude> :type not
```



```
not :: Bool -> Bool
```

```
Prelude> :type (&&)
```



```
(&&) :: Bool -> Bool -> Bool
```

```
Prelude> :type (||)
```



```
(||) :: Bool -> Bool -> Bool
```

- not: Eine Eingabe vom Typ Bool und Ausgabe vom Typ Bool
- (&&): Zwei Eingaben vom Typ Bool und Ausgabe vom Typ Bool
- (||): Zwei Eingaben vom Typ Bool und Ausgabe vom Typ Bool

Funktionstypen mit Typklassen

Typen von mod und div:

```
mod :: Integer -> Integer -> Integer
```

```
div :: Integer -> Integer -> Integer
```

In Wirklichkeit:

```
Prelude> :type mod   
mod :: (Integral a) => a -> a -> a  
Prelude> :type div   
div :: (Integral a) => a -> a -> a
```

In etwa: Für alle Typen `a` die `Integral`-Typen sind,
hat `mod` den Typ `a -> a -> a`

Weitere Funktionen

```
Prelude> :type (==)   
(==) :: (Eq a) => a -> a -> Bool  
Prelude> :type (<)   
(<) :: (Ord a) => a -> a -> Bool  
Prelude> :type (<=)   
(<=) :: (Ord a) => a -> a -> Bool  
Prelude> :type (+)   
(+) :: (Num a) => a -> a -> a  
Prelude> :type (-)   
(-) :: (Num a) => a -> a -> a
```

Typgerecht programmieren

Die Typen müssen stets passen, sonst gibt's einen **Typfehler**:

```
Prelude> :type not
not :: Bool -> Bool
Prelude> not 'C'
<interactive>:1:4:
  Couldn't match expected type 'Bool' against inferred type 'Char'
    In the first argument of 'not', namely 'C'
    In the expression: not 'C'
    In the definition of 'it': it = not 'C'
```

Manchmal “merkwürdige” Fehlermeldung:

```
Prelude> not 5
<interactive>:1:4:
  No instance for (Num Bool)
    arising from the literal '5' at <interactive>:1:4
  Possible fix: add an instance declaration for (Num Bool)
    In the first argument of 'not', namely '5'
    In the expression: not 5
    In the definition of 'it': it = not 5
```

Funktionen selbst definieren

```
verdopple x = x + x
```

Funktionen selbst definieren

```
verdopple x = x + x
```

Allgemein:

$$\textit{funktion_Name} \quad \textit{par}_1 \quad \dots \quad \textit{par}_n \quad = \quad \textit{Haskell_Ausdruck}$$

wobei

- $\textit{par}_i$: Formale Parameter, z.B. Variablen x , y , ...
- Die $\textit{par}_i$ dürfen rechts im *Haskell_Ausdruck* verwendet werden
- *funktion_Name* muss mit Kleinbuchstaben oder einem Unterstrich beginnen

Funktionen selbst definieren

```
verdopple :: Integer -> Integer  
verdopple x = x + x
```

Allgemein:

$$\text{funktion_Name} \quad \text{par}_1 \quad \dots \quad \text{par}_n = \text{Haskell_Ausdruck}$$

wobei

- par_i : Formale Parameter, z.B. Variablen x , y , ...
- Die par_i dürfen rechts im *Haskell_Ausdruck* verwendet werden
- *funktion_Name* muss mit Kleinbuchstaben oder einem Unterstrich beginnen

Man darf auch den Typ angeben!

Formale Parameter

Gesucht:

Funktion erhält zwei Eingaben und liefert
"Die Eingaben sind gleich",
wenn die beiden Eingaben gleich sind.

Formale Parameter

Gesucht:

Funktion erhält zwei Eingaben und liefert
"Die Eingaben sind gleich",
wenn die beiden Eingaben gleich sind.

Falscher Versuch:

```
sonicht x x = "Die Eingaben sind gleich!"
```

```
Conflicting definitions for 'x'
In the definition of 'sonicht'
Failed, modules loaded: none.
```

Die formalen Parameter müssen **unterschiedliche** Namen haben.

```
vergleiche x y =
  if x == y then "Die Eingaben sind gleich!" else "Ungleich!"
```

Funktion testen

```
Prelude> :load programme/einfacheFunktionen.hs   
[1 of 1] Compiling Main ( programme/einfacheFunktionen.hs)  
Ok, modules loaded: Main.  
*Main> verdopple 5   
10  
*Main> verdopple 100   
200  
*Main> verdopple (verdopple (2*3) + verdopple (6+9))   
84
```


Fallunterscheidung: if-then-else

Syntax: `if  $b$  then  $e_1$  else  $e_2$`

Bedeutung:

$$\text{if } b \text{ then } e_1 \text{ else } e_2 = \begin{cases} e_1 & \text{wenn } b \text{ zu True ausgewertet} \\ e_2 & \text{wenn } b \text{ zu False ausgewertet} \end{cases}$$

Beispiel

```
verdoppleGerade :: Integer -> Integer  
verdoppleGerade x = if even x then verdopple x else x
```

even testet, ob eine Zahl gerade ist:

`even x = x 'mod' 2 == 0`

```
*Main> :reload   
[1 of 1] Compiling Main ( programme/einfacheFunktionen.hs )  
Ok, modules loaded: Main.  
*Main> verdoppleGerade 50   
100  
*Main> verdoppleGerade 17   
17
```

Weitere Beispiele zu if-then-else

Verschachteln von if-then-else:

```
jenachdem :: Integer -> Integer
jenachdem x = if x < 100 then 2*x else
               if x <= 1000 then 3*x else x
```

Falsche Einrückung:

```
jenachdem x =
if x < 100 then 2*x else
  if x <= 1000 then 3*x else x
```

```
Prelude> :reload 
[1 of 1] Compiling Main ( programme/einfacheFunktionen.hs )
programme/einfacheFunktionen.hs:9:0:
    parse error (possibly incorrect indentation)
Failed, modules loaded: none.
Prelude>
```

Noch ein Beispiel

```
verdoppeln_oder_verdreifachen :: Bool -> Integer -> Integer
verdoppeln_oder_verdreifachen b x =
    if b then 2*x else 3*x
```

```
*Main> verdoppeln_oder_verdreifachen True 10
20
*Main> verdoppeln_oder_verdreifachen False 10
30
```

verdoppeln mithilfe von verdoppeln_oder_verdreifachen:

```
verdoppeln2 :: Integer -> Integer
verdoppeln2 x = verdoppeln_oder_verdreifachen True x
-- oder auch:
verdoppeln3 :: Integer -> Integer
verdoppeln3 = verdoppeln_oder_verdreifachen True
```

verdoppeln3: **keine** Eingabe, **Ausgabe ist eine Funktion**

Aufgabe

Ordnung BSc Informatik §26 Abs.7

„Die Gesamtnote einer bestandenen Bachelorprüfung lautet:

Bei einem Durchschnitt bis einschließlich 1,5:

sehr gut

bei einem Durchschnitt von 1,6 bis einschließlich 2,5:

gut

bei einem Durchschnitt von 2,6 bis einschließlich 3,5:

befriedigend

bei einem Durchschnitt von 3,6 bis einschließlich 4,0:

ausreichend“

Implementiere eine Haskellfunktion

```
gesamtnote :: Double -> String
```

die bei Eingabe eines Durchschnitts die Gesamtnote als String ausgibt.

Higher-Order Funktionen

- D.h.: **Rückgabewerte** dürfen in Haskell auch **Funktionen** sein
- Auch **Argumente** (Eingaben) dürfen **Funktionen** sein:

```
wende_an_und_addiere f x y = (f x) + (f y)
```

```
*Main> wende_an_und_addiere verdopple 10 20 
```

```
60
```

```
*Main> wende_an_und_addiere jenachdem 150 3000 
```

```
3450
```

Daher spricht man auch von **Funktionen höherer Ordnung**!

Nochmal Typen

Typ von `wende_an_und_addiere`

`wende_an_und_addiere :: (Integer -> Integer) -> Integer -> Integer -> Integer`

`wende_an_und_addiere f x y = (f x) + (f y)`

Nochmal Typen

Typ von `wende_an_und_addiere`

`wende_an_und_addiere` :: $\underbrace{(\text{Integer} \rightarrow \text{Integer})}_{\text{Typ von } f} \rightarrow \underbrace{\text{Integer}}_{\text{Typ von } x} \rightarrow \underbrace{\text{Integer}}_{\text{Typ von } y} \rightarrow \underbrace{\text{Integer}}_{\text{Typ des Ergebnisses}}$

`wende_an_und_addiere` $f\ x\ y = (f\ x) + (f\ y)$

Nochmal Typen

Typ von `wende_an_und_addiere`

`wende_an_und_addiere :: (Integer -> Integer) -> Integer -> Integer -> Integer`

`wende_an_und_addiere f x y = (f x) + (f y)`

Achtung: Im Typ

`(Integer -> Integer) -> Integer -> Integer -> Integer`

darf man die Klammern **nicht** weglassen:

`Integer -> Integer -> Integer -> Integer -> Integer`

denn das entspricht

`Integer -> (Integer -> (Integer -> (Integer -> Integer)))`.

Polymorphe Typen

```
zweimal_anwenden :: (a -> a) -> a -> a  
zweimal_anwenden f x = f (f x)
```

Da **Typvariablen** in Haskell erlaubt sind, spricht man von
polymorphen Typen

Für Typvariablen kann man Typen einsetzen!

Polymorphe Typen

```
zweimal_anwenden :: (a -> a) -> a -> a  
zweimal_anwenden f x = f (f x)
```

Da **Typvariablen** in Haskell erlaubt sind, spricht man von **polymorphen Typen**

Für Typvariablen kann man Typen einsetzen!

z.B. `a = Int`

```
zweimal_anwenden :: (a -> a) -> a -> a
```

Polymorphe Typen

```
zweimal_anwenden :: (a -> a) -> a -> a  
zweimal_anwenden f x = f (f x)
```

Da **Typvariablen** in Haskell erlaubt sind, spricht man von
polymorphen Typen

Für Typvariablen kann man Typen einsetzen!

z.B. `a = Int`

```
zweimal_anwenden :: (Int -> Int) -> Int -> Int
```

Polymorphe Typen

```
zweimal_anwenden :: (a -> a) -> a -> a  
zweimal_anwenden f x = f (f x)
```

Da **Typvariablen** in Haskell erlaubt sind, spricht man von **polymorphen Typen**

Für Typvariablen kann man Typen einsetzen!

z.B. `a = Int`

```
zweimal_anwenden :: (Int -> Int) -> Int -> Int
```

z.B. `a = Bool`

```
zweimal_anwenden :: (a -> a) -> a -> a
```

Polymorphe Typen

```
zweimal_anwenden :: (a -> a) -> a -> a  
zweimal_anwenden f x = f (f x)
```

Da **Typvariablen** in Haskell erlaubt sind, spricht man von
polymorphen Typen

Für Typvariablen kann man Typen einsetzen!

z.B. `a = Int`

```
zweimal_anwenden :: (Int -> Int) -> Int -> Int
```

z.B. `a = Bool`

```
zweimal_anwenden :: (Bool -> Bool) -> Bool -> Bool
```

Polymorphe Typen

```
zweimal_anwenden :: (a -> a) -> a -> a
zweimal_anwenden f x = f (f x)
```

Da **Typvariablen** in Haskell erlaubt sind, spricht man von **polymorphen Typen**

Für Typvariablen kann man Typen einsetzen!

z.B. $a = \text{Int}$

```
zweimal_anwenden :: (Int -> Int) -> Int -> Int
```

z.B. $a = \text{Bool}$

```
zweimal_anwenden :: (Bool -> Bool) -> Bool -> Bool
```

z.B. $a = \text{Char} \rightarrow \text{Char}$

```
zweimal_anwenden :: (a -> a) -> a -> a
```

Polymorphe Typen

```
zweimal_anwenden :: (a -> a) -> a -> a
zweimal_anwenden f x = f (f x)
```

Da **Typvariablen** in Haskell erlaubt sind, spricht man von **polymorphen Typen**

Für Typvariablen kann man Typen einsetzen!

z.B. $a = \text{Int}$

```
zweimal_anwenden :: (Int -> Int) -> Int -> Int
```

z.B. $a = \text{Bool}$

```
zweimal_anwenden :: (Bool -> Bool) -> Bool -> Bool
```

z.B. $a = \text{Char} \rightarrow \text{Char}$

```
zweimal_anwenden :: ((Char -> Char) -> (Char -> Char)) -> (Char -> Char) -> (Char -> Char)
```


Programmieren mit Haskell

Rekursion

Rekursion

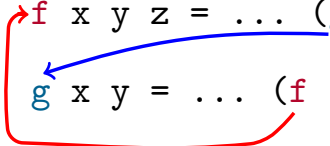
Eine Funktion ist **rekursiv**, wenn sie sich **selbst aufrufen** kann.

f x y z = ... (**f** a b c) ...



oder z.B. auch

f x y z = ... (**g** a b) ...
g x y = ... (**f** c d e) ...



Rekursion (2)

Bei Rekursion muss man aufpassen:

```
endlos_eins_addieren x = endlos_eins_addieren (x+1)
```

```
endlos_eins_addieren 0  
--> endlos_eins_addieren 1  
--> endlos_eins_addieren 2  
--> endlos_eins_addieren 3  
--> endlos_eins_addieren 4  
--> ...
```

endlos_eins_addieren 0 **terminiert nicht!**

Rekursion (3)

So macht man es richtig:

- **Rekursionsanfang**: Der Fall, für den sich die Funktion **nicht** mehr selbst aufruft, sondern **abbricht**
- **Rekursionsschritt**: Der rekursive Aufruf

Darauf achten, dass der Rekursionsanfang stets **erreicht** wird.

Beispiel:

```
erste_rekursive_Funktion x =  
  if x <= 0 then 0 -- Rekursionsanfang  
  else x+(erste_rekursive_Funktion (x-1)) -- Rekursionsschritt
```

Rekursion (4)

```
erste_rekursive_Funktion x =  
  if x <= 0 then 0                -- Rekursionsanfang  
  else x+(erste_rekursive_Funktion (x-1)) -- Rekursionsschritt
```

Was berechnet `erste_rekursive_Funktion`?

- `erste_rekursive_Funktion n` ergibt 0 für $n \leq 0$
- $n > 0$?

Testen:

```
*Main> erste_rekursive_Funktion 5  
15  
*Main> erste_rekursive_Funktion 10  
55  
*Main> erste_rekursive_Funktion 11  
66  
*Main> erste_rekursive_Funktion 12  
78  
*Main> erste_rekursive_Funktion 1  
1
```

Rekursion (5)

```
erste_rekursive_Funktion x =  
  if x <= 0 then 0                -- Rekursionsanfang  
  else x+(erste_rekursive_Funktion (x-1)) -- Rekursionsschritt
```

Ein Beispiel nachvollziehen:

```
erste_rekursive_Funktion 5  
= 5 + erste_rekursive_Funktion 4  
= 5 + (4 + erste_rekursive_Funktion 3)  
= 5 + (4 + (3 + erste_rekursive_Funktion 2))  
= 5 + (4 + (3 + (2 + erste_rekursive_Funktion 1)))  
= 5 + (4 + (3 + (2 + (1 + erste_rekursive_Funktion 0))))  
= 5 + (4 + (3 + (2 + (1 + 0))))  
= 15
```

Rekursion (6)

```
erste_rekursive_Funktion x =  
  if x <= 0 then 0                -- Rekursionsanfang  
  else x+(erste_rekursive_Funktion (x-1)) -- Rekursionsschritt
```

Allgemein:

```
erste_rekursive_Funktion x  
= x + erste_rekursive_Funktion (x-1)  
= x + (x-1) + erste_rekursive_Funktion (x-2))  
= x + (x-1) + (x-2) + erste_rekursive_Funktion (x-3)))  
= ...
```

Rekursion (6)

```
erste_rekursive_Funktion x =
  if x <= 0 then 0                -- Rekursionsanfang
  else x+(erste_rekursive_Funktion (x-1)) -- Rekursionsschritt
```

Allgemein:

```
erste_rekursive_Funktion x
= x + erste_rekursive_Funktion (x-1)
= x + (x-1) + erste_rekursive_Funktion (x-2))
= x + (x-1) + (x-2) + erste_rekursive_Funktion (x-3)))
= ...
```

Das ergibt $x + (x - 1) + (x - 2) + \dots + 0 = \sum_{i=0}^x i$

Rekursion (7)

Warum ist Rekursion nützlich?

- Man kann damit **schwierige Probleme** einfach **lösen**

Wie geht man vor? (z.B. implementiere $f(n) = \sum_{i=0}^n i$)

- Rekursionsanfang:

Der **einfache** Fall, für den man die Lösung direkt kennt

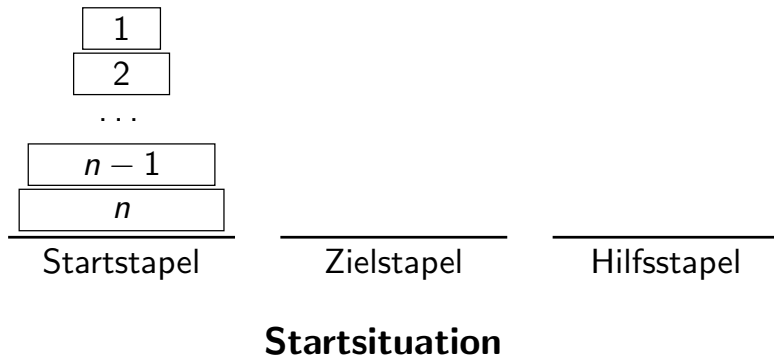
$$f(0) = \sum_{i=0}^0 i = 0$$

- Rekursionsschritt:

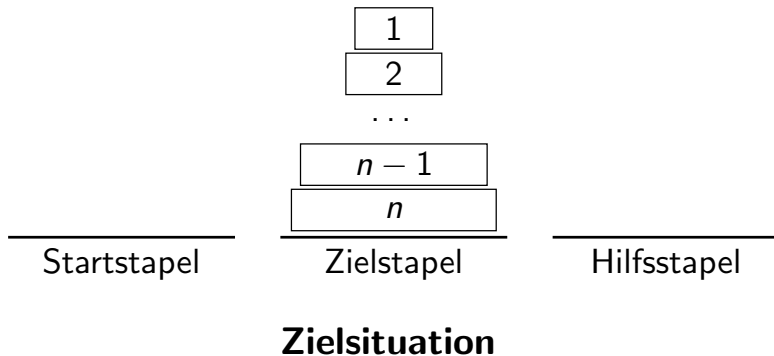
Man löst **ganz wenig selbst**, bis das Problem etwas kleiner ist.
Das (immer noch große) **Restproblem** erledigt die **Rekursion**,

$$f(n) = \sum_{i=0}^n i = n + \sum_{i=0}^{n-1} i = n + f(n-1) \quad (\text{für } n > 0)$$

Rekursion: Türme von Hanoi



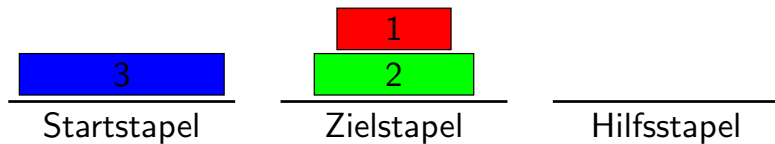
Rekursion: Türme von Hanoi

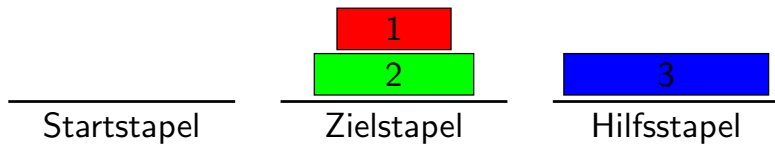


Beispiel $n = 3$ 

Beispiel $n = 3$ 

Beispiel $n = 3$ 

Beispiel $n = 3$ 

Beispiel $n = 3$ 

Beispiel $n = 3$ 

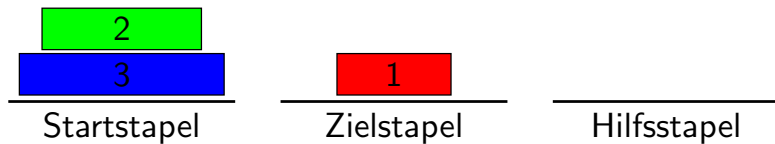
Beispiel $n = 3$ 

Beispiel $n = 3$ 

keine korrekte Lösung

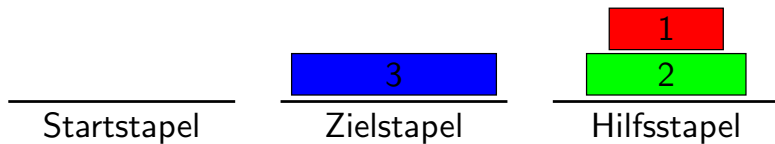
Beispiel $n = 3$ 

zurück zum Anfang

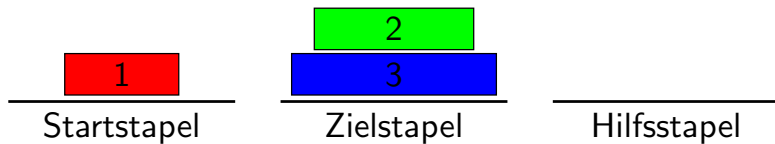
Beispiel $n = 3$ 

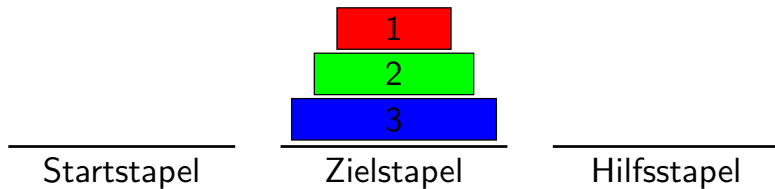
Beispiel $n = 3$ 

Beispiel $n = 3$ 

Beispiel $n = 3$ 

Beispiel $n = 3$ 

Beispiel $n = 3$ 

Beispiel $n = 3$ 

Korrekte Lösung

Lösen durch Rekursion: Rekursionanfang



$n = 1$: Verschiebe Scheibe von Startstapel auf Zielstapel

Lösen durch Rekursion: Rekursionanfang



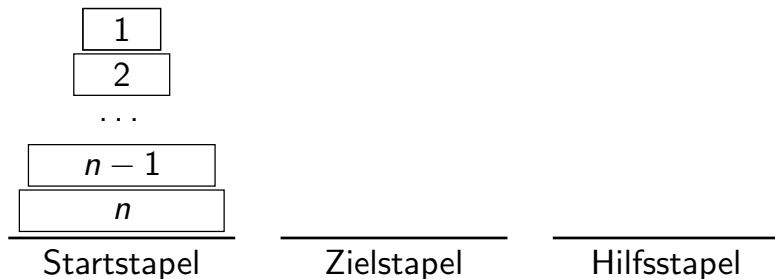
$n = 1$: Verschiebe Scheibe von Startstapel auf Zielstapel

Lösen durch Rekursion: Rekursionanfang

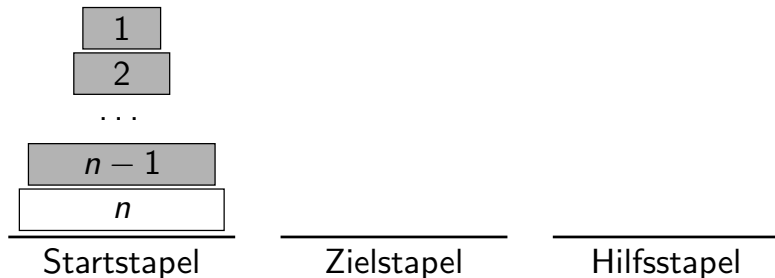


$n = 1$: Verschiebe Scheibe von Startstapel auf Zielstapel

Lösen durch Rekursion: Rekursionsschritt



Lösen durch Rekursion: Rekursionsschritt



1. Verschiebe den Turm der Höhe $n - 1$ **rekursiv** auf den Hilfsstapel

Lösen durch Rekursion: Rekursionsschritt



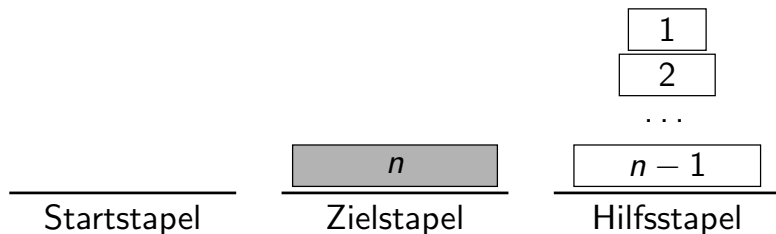
1. Verschiebe den Turm der Höhe $n-1$ **rekursiv** auf den Hilfsstapel

Lösen durch Rekursion: Rekursionsschritt



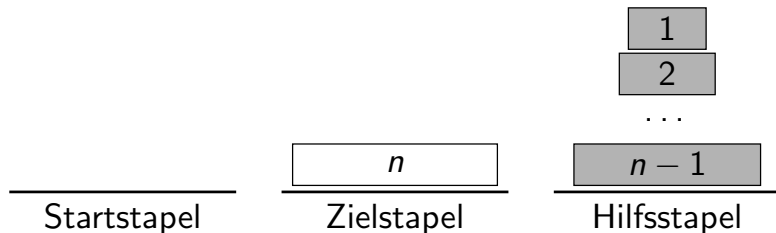
2. Verschiebe Scheibe n auf den Zielstapel

Lösen durch Rekursion: Rekursionsschritt



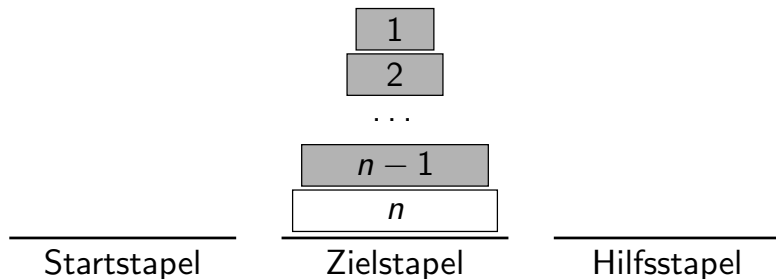
2. Verschiebe Scheibe n auf den Zielstapel

Lösen durch Rekursion: Rekursionsschritt



3. Verschiebe den Turm der Höhe $n - 1$ **rekursiv** auf den Zielstapel

Lösen durch Rekursion: Rekursionsschritt



3. Verschiebe den Turm der Höhe $n - 1$ **rekursiv** auf den Zielstapel

Pseudo-Algorithmus

verschiebe(n , start, ziel, hilf)

1. Wenn $n > 1$, dann **verschiebe**($n-1$, start, hilf, ziel)
2. Schiebe Scheibe n von start auf ziel
3. Wenn $n > 1$, dann **verschiebe**($n-1$, hilf, ziel, start)

- Rekursionanfang ist bei $n = 1$: keine rekursiven Aufrufe
- Beachte: Zwei rekursive Aufrufe pro Rekursionsschritt
- Haskell-Implementierung: Später

Rekursion, weitere Beispiele

Beispiel: n -mal Verdoppeln

- Rekursionsanfang:

```
n_mal_verdoppeln :: Integer -> Integer -> Integer  
n_mal_verdoppeln x n =
```

Rekursion, weitere Beispiele

Beispiel: n -mal Verdoppeln

- Rekursionsanfang: $n = 0$: Gar nicht verdoppeln

```
n_mal_verdoppeln :: Integer -> Integer -> Integer
n_mal_verdoppeln x n =
    if n == 0 then x
```


Rekursion, weitere Beispiele

Beispiel: n -mal Verdoppeln

- Rekursionsanfang: $n = 0$: Gar nicht verdoppeln
- Rekursionsschritt: $n > 0$: Einmal selbst verdoppeln, die restlichen $n - 1$ Verdopplungen der Rekursion überlassen

```
n_mal_verdoppeln :: Integer -> Integer -> Integer
n_mal_verdoppeln x n =
  if n == 0 then x
  else n_mal_verdoppeln (verdopple x) (n-1)
```

Pattern matching (auf Zahlen)

```
n_mal_verdoppeln :: Integer -> Integer -> Integer
n_mal_verdoppeln x n =
  if n == 0 then x
  else n_mal_verdoppeln (verdopple x) (n-1)
```

Man darf statt Variablen auch **Pattern** in der Funktionsdefinition verwenden, und **mehrere Definitionsgleichungen** angeben. Die Pattern werden von unten nach oben abgearbeitet.

```
n_mal_verdoppeln2 :: Integer -> Integer -> Integer
n_mal_verdoppeln2 x 0 = x
n_mal_verdoppeln2 x n = n_mal_verdoppeln2 (verdopple x) (n-1)
```

Falsch:

```
n_mal_verdoppeln2 :: Integer -> Integer -> Integer
n_mal_verdoppeln2 x n = n_mal_verdoppeln2 (verdopple x) (n-1)
n_mal_verdoppeln2 x 0 = x
```

Guards

Guards (Wächter): Boolesche Ausdrücke,
die die Definition der Funktion festlegen

```
f x1 ... xm  
| 1. Guard = e1  
...  
| n. Guard = en
```

- Abarbeitung von oben nach unten
- Erster Guard, der zu True auswertet, bestimmt die Definition.

```
n_mal_verdoppeln3 :: Integer -> Integer -> Integer  
n_mal_verdoppeln3 x n  
| n == 0      = x  
| otherwise = n_mal_verdoppeln3 (verdopple x) (n-1)
```

Vordefiniert: otherwise = True

Die Error-Funktion

```
n_mal_verdoppeln3 :: Integer -> Integer -> Integer
n_mal_verdoppeln3 x n
  | n == 0      = x
  | otherwise = n_mal_verdoppeln3 (verdopple x) (n-1)
```

Was passiert bei negativem n?

Die Error-Funktion

```
n_mal_verdoppeln3 :: Integer -> Integer -> Integer
n_mal_verdoppeln3 x n
  | n == 0      = x
  | otherwise = n_mal_verdoppeln3 (verdopple x) (n-1)
```

Was passiert bei negativem n?

- `error :: String -> a`

```
n_mal_verdoppeln4 :: Integer -> Integer -> Integer
n_mal_verdoppeln4 x n
  | n < 0      = error "Negatives Verdoppeln verboten!"
  | n == 0     = x
  | otherwise = n_mal_verdoppeln4 (verdopple x) (n-1)
```

```
*Main> n_mal_verdoppeln4 10 (-10)
*** Exception:
in n_mal_verdoppeln4: negatives Verdoppeln ist verboten
```

Beispiel

In einem Wald werden am 1.1. des ersten Jahres 10 Rehe gezählt. Der erfahrene Förster weiß, dass sich im Laufe eines Jahres, die Anzahl an Rehen durch Paarung verdreifacht. In jedem Jahr schießt der Förster 17 Rehe. In jedem 2. Jahr gibt der Förster die Hälfte der verbleibenden Rehe am 31.12. an einen anderen Wald ab.

Wieviel Rehe gibt es im Wald am 1.1. des Jahres n ?

Beispiel

*In einem Wald werden **am 1.1. des ersten Jahres 10 Rehe gezählt**. Der erfahrene Förster weiß, dass sich im Laufe eines Jahres, die Anzahl an Rehen durch Paarung verdreifacht. In jedem Jahr schießt der Förster 17 Rehe. In jedem 2. Jahr gibt der Förster die Hälfte der verbleibenden Rehe am 31.12. an einen anderen Wald ab.*

Wieviel Rehe gibt es im Wald am 1.1. des Jahres n ?

Rekursionsanfang: Jahr 1, 10 Rehe

```
anzahlRehe 1 = 10
```

Beispiel

In einem Wald werden am 1.1. des ersten Jahres 10 Rehe gezählt. Der erfahrene Förster weiß, dass sich im Laufe eines Jahres, die Anzahl an Rehen durch Paarung verdreifacht. In jedem Jahr schießt der Förster 17 Rehe. In jedem 2. Jahr gibt der Förster die Hälfte der verbleibenden Rehe am 31.12. an einen anderen Wald ab.

Wieviel Rehe gibt es im Wald am 1.1. des Jahres n ?

Rekursionsanfang: Jahr 1, 10 Rehe

Rekursionsschritt: Sei k = Anzahl Rehe am 1.1. des Jahres $n - 1$

```
anzahlRehe 1 = 10
```

```
anzahlRehe n =
```

```
    anzahlRehe (n-1)
```


Beispiel

In einem Wald werden am 1.1. des ersten Jahres 10 Rehe gezählt. Der erfahrene Förster weiß, dass sich im Laufe eines Jahres, die Anzahl an Rehen durch Paarung **verdreifacht**. In jedem Jahr schießt der Förster **17 Rehe**. In jedem **2. Jahr** gibt der Förster **die Hälfte** der verbleibenden Rehe am 31.12. an einen anderen Wald ab.

Wieviel Rehe gibt es im Wald am 1.1. des Jahres n ?

Rekursionsanfang: Jahr 1, 10 Rehe

Rekursionsschritt: Sei k = Anzahl Rehe am 1.1. des Jahres $n - 1$

$$\text{Jahr } n: \begin{cases} 3 * k - 17, & \text{falls } n - 1 \text{ kein zweites Jahr} \\ (3 * k - 17) / 2, & \text{falls } n - 1 \text{ ein zweites Jahr} \end{cases}$$

```
anzahlRehe 1 = 10
anzahlRehe n = if even (n-1)
                then (3*(anzahlRehe (n-1))-17) `div` 2
                else 3*(anzahlRehe (n-1))-17
```

Beispiel (2)

```
*Main> anzahlRehe 1
10
*Main> anzahlRehe 2
13
*Main> anzahlRehe 3
11
*Main> anzahlRehe 4
16
*Main> anzahlRehe 5
15
*Main> anzahlRehe 6
28
*Main> anzahlRehe 7
33
*Main> anzahlRehe 8
82
*Main> anzahlRehe 9
114
*Main> anzahlRehe 10
325
*Main> anzahlRehe 50
3626347914090925
```

Let-Ausdrücke: Lokale Definitionen

```
anzahlRehe 1 = 10
anzahlRehe n = if even (n-1)
               then (3*(anzahlRehe (n-1))-17) 'div' 2
               else 3*(anzahlRehe (n-1))-17
```

Mit let:

```
anzahlRehe2 1 = 10
anzahlRehe2 n = let k = (3*anzahlRehe2 (n-1))-17
                 in  if even (n-1) then k 'div' 2
                 else !k
```

Let-Ausdrücke: Lokale Definitionen (2)

Allgemeiner:

```
let  Variable1  =  Ausdruck1  
     Variable2  =  Ausdruck2  
     ...  
     VariableN  =  AusdruckN  
in   Ausdruck
```

Programmieren mit Haskell

Programmieren mit Listen

Listen

- Liste = Folge von Elementen
- z.B. [True,False,False,True,True] und [1,2,3,4,5,6]
- In Haskell sind nur **homogene** Listen erlaubt:
Alle Elemente haben den **gleichen Typ**
- z.B. **verboten**: [True,'a',False,2]

```
Prelude> [True,'a',False,2]
```

```
<interactive>:1:6:
```

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

```
In the expression: 'a'
```

```
In the expression: [True, 'a', False, 2]
```

```
In the definition of 'it': it = [True, 'a', False, ....]
```

Listen: Typ

- Allgemeiner Typ: `[a]`
- Das `a` bezeichnet den Typ der Elemente
- z.B. Liste von Zeichen: Typ `[Char]` usw.

```
Prelude> :type [True,False]
```

```
[True,False] :: [Bool]
```

```
Prelude> :type ['A','B']
```

```
['A','B'] :: [Char]
```

```
Prelude> :type [1,2,3]
```

```
[1,2,3] :: (Num t) => [t]
```

Listen: Typ

- Allgemeiner Typ: `[a]`
- Das `a` bezeichnet den Typ der Elemente
- z.B. Liste von Zeichen: Typ `[Char]` usw.

```
Prelude> :type [True,False]
```

```
[True,False] :: [Bool]
```

```
Prelude> :type ['A','B']
```

```
['A','B'] :: [Char]
```

```
Prelude> :type [1,2,3]
```

```
[1,2,3] :: (Num t) => [t]
```

Auch möglich:

```
*Main> :type [verdopple, verdoppleGerade, jenachdem]
```

```
[verdopple, verdoppleGerade, jenachdem] :: [Integer -> Integer]
```

```
*Main> :type [[True,False], [False,True,True], [True,True]]
```

```
[[True,False], [False,True,True], [True,True]] :: [[Bool]]
```


Listen erstellen

- Eckige Klammern und Kommas z.B. `[1,2,3]`
- Das ist jedoch nur **Syntaktischer Zucker**

Listen sind **rekursiv** definiert:

- „Rekursionsanfang“ ist die **leere Liste** `[]` („Nil“)
- „Rekursionsschritt“ mit `:` („Cons“)
 - `x` ein **Listenelement**
 - `xs` eine **Liste** (mit $n - 1$ Elementen)

Dann ist `x:xs` Liste mit n Elementen beginnend mit `x` und anschließend folgen die Elemente aus `xs`

`[1,2,3]` ist in Wahrheit `1:(2:(3:[]))`

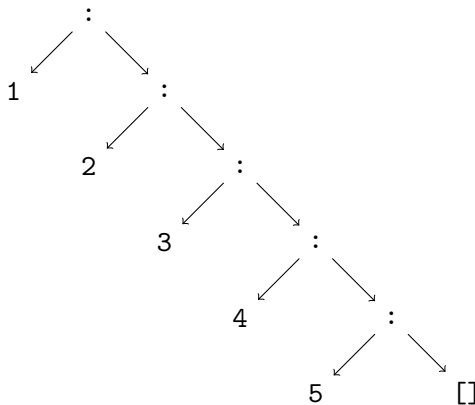
Typen:

`[] :: [a]`

`(:) :: a -> [a] -> [a]`

Interne Darstellung der Listen

$[1,2,3,4,5] = 1:(2:(3:(4:(5:[])))$



Beispiel

Konstruiere die Listen der Zahlen $n, n - 1 \dots, 1$:

```
nbis1 :: Integer -> [Integer]
nbis1 0 = []
nbis1 n = n:(nbis1 (n-1))
```

```
*Main> nbis1 0
[]
*Main> nbis1 1
[1]
*Main> nbis1 10
[10,9,8,7,6,5,4,3,2,1]
*Main> nbis1 100
[100,99,98,97,96,95,94,93,92,91,90,89,88,87,86,85,84,83,82,81,80,79,
 78,77,76,75,74,73,72,71,70,69,68,67,66,65,64,63,62,61,60,59,58,57,
 56,55,54,53,52,51,50,49,48,47,46,45,44,43,42,41,40,39,38,37,36,35,
 34,33,32, 31,30,29,28,27,26,25,24,23,22,21,20,19,18,17,16,15,14,13,
 12,11,10,9,8,7,6,5,4,3,2,1]
```

Listen zerlegen

Vordefiniert:

- `head :: [a] -> a` liefert das erste Element einer nicht-leeren Liste
- `tail :: [a] -> [a]` liefert den Schwanz einer nicht-leeren Liste.
- `null :: [a] -> Bool` testet, ob eine Liste leer ist

```
*> head [1,2]
1
*> tail [1,2,3]
[2,3]
*> head []
*** Exception: Prelude.head: empty list
*> null []
True
*> null [4]
False
```

Beispiel

Funktion, die das letzte Element einer Liste liefert.

```
letztesElement :: [a] -> a
letztesElement xs = if null xs then
                      error "Liste ist leer"
                    else
                      if null (tail xs) then head xs
                      else letztesElement (tail xs)
```

```
Main> letztesElement [True,False,True]
True
*Main> letztesElement [1,2,3]
3
*Main> letztesElement (nbis1 1000)
1
*Main> letztesElement [[1,2,3], [4,5,6], [7,8,9]]
[7,8,9]
```

Listen zerlegen mit Pattern

In

$$f \text{ } par_1 \dots par_n = rumpf$$

dürfen par_i auch sog. **Pattern** sein.

Z.B.

```
eigenesHead []      = error "empty list"
eigenesHead (x:xs) = x
```

Auswertung von `eigenesHead (1:(2:[]))`

- Das erste Pattern das zu `(1:(2:[]))` passt („**matcht**“) wird genommen
- Dies ist `(x:xs)`. Nun wird anhand des Patterns zerlegt:

$$x = 1$$

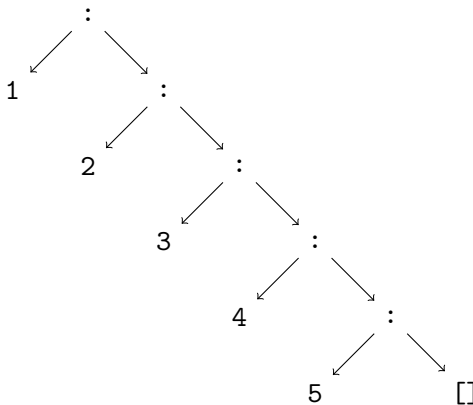
$$xs = (2:[])$$

Pattern-Matching

Pattern []

[]

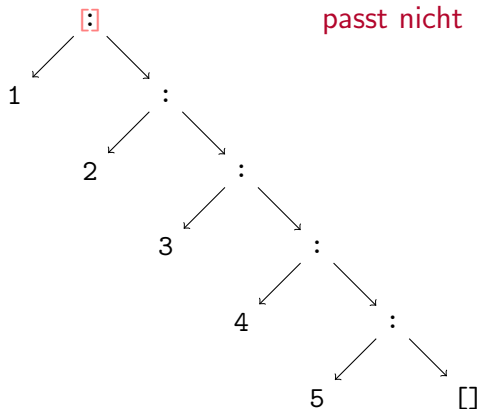
Liste



Pattern-Matching

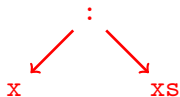
Pattern `[]`

Liste

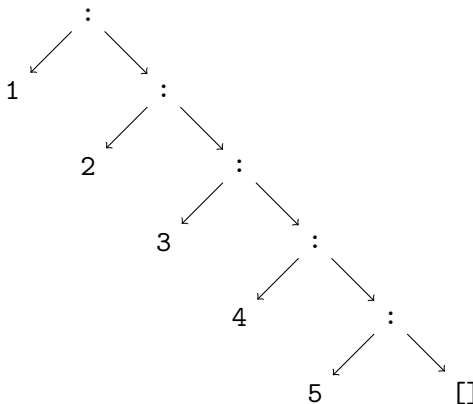


Pattern-Matching

Pattern $x:xs$



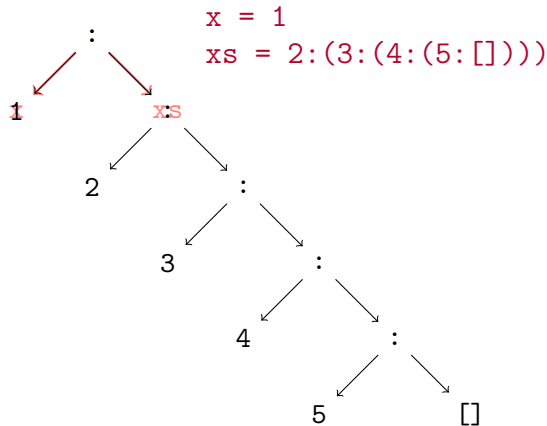
Liste



Pattern-Matching

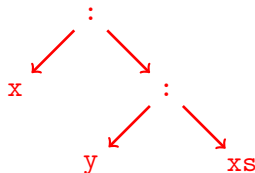
Pattern $x:xs$

Liste

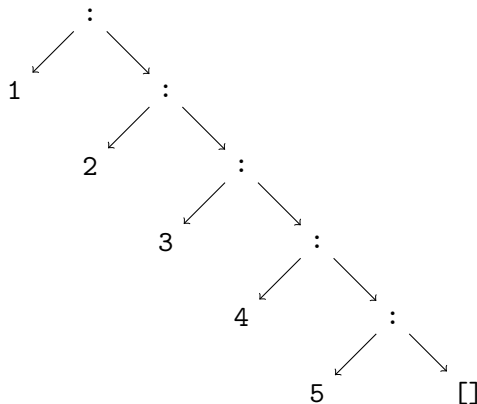


Pattern-Matching

Pattern $x:(y:xs)$



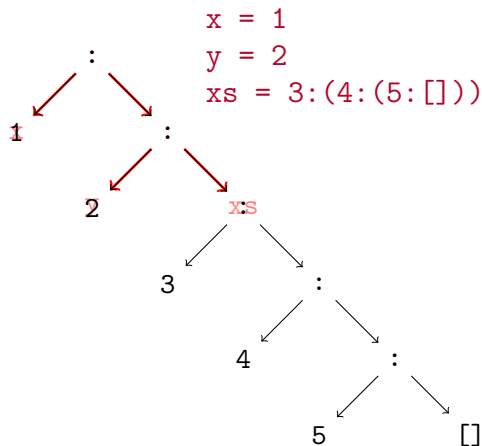
Liste



Pattern-Matching

Pattern $x:(y:xs)$

Liste



Beispiele: Pattern-Match

```
eigenesNull []      = True
eigenesNull (x:xs) = False

-- alternativ:
eigenesNull2 []     = True
eigenesNull2 xs     = False -- matcht immer!

-- falsch!:
falschesNull xs     = False
falschesNull []     = True
```

Warning: Pattern match(es) are overlapped

In the definition of 'falschesNull': falschesNull [] = ...

```
*Main> falschesNull [1]
False
*Main> falschesNull []
False
*Main> eigenesNull []
True
```

Letztes Element mit Pattern:

```
letztesElement2 []      = error "leere Liste"  
letztesElement2 (x:[]) = x  
letztesElement2 (x:xs) = letztesElement2 xs
```

`(x:[])` passt nur für einelementige Listen!

Vordefinierte Listenfunktionen (Auswahl)

- `length :: [a] -> Int` berechnet die Länge einer Liste
- `take :: Int -> [a] -> [a]` erwartet eine Zahl k und eine Liste xs und liefert die Liste der ersten k Elemente von xs
- `drop :: Int -> [a] -> [a]` erwartet eine Zahl k und eine Liste xs und liefert xs ohne die ersten k Elemente.
- `(++) :: [a] -> [a] -> [a]` „append“: hängt zwei Listen aneinander, kann infix in der Form $xs ++ ys$ verwendet werden.
- `concat :: [[a]] -> [a]` glättet eine Liste von Listen. Z.B.
`concat [xs,ys]` ist gleich zu $xs ++ ys$.
- `reverse :: [a] -> [a]` dreht die Reihenfolge der Elemente einer Liste um.

Nochmal Strings

"Hallo Welt" ist nur syntaktischer Zucker für

```
['H','a','l','l','o',' ','W','e','l','t']
```

bzw.

```
'H':('a':('l':('l':('o':(' '(:('W':('e':('l':('t':[]))))))))))
```

```
*Main> head "Hallo Welt"
'H'
*Main> tail "Hallo Welt"
"allo Welt"
*Main> null "Hallo Welt"
False
*Main> null ""
True
*Main> letztesElement "Hallo Welt"
't'
```


Funktionen auf Strings

- `words :: String -> [String]`: Zerlegt eine Zeichenkette in eine *Liste von Worten*
- `unwords :: [String] -> String`: Macht aus einer Liste von Worten einen einzelnen String.
- `lines :: String -> [String]`: Zerlegt eine Zeichenkette in eine *Liste von Zeilen*

Z.B.

```
anzahlWorte :: String -> Int
anzahlWorte text = length (words text)
```

Paare und Tupel

- Paare in Haskell: (e1,e2) z.B. (1,'A')
- Die Typen der Komponenten dürfen **verschieden** sein.

```
Main> :type ("Hallo",True)
("Hallo",True) :: ([Char], Bool)
Main> :type ([1,2,3], 'A')
([1,2,3],False) :: (Num t) => ([t], Char)
*Main> :type (letztesElement, "Hallo" ++ "Welt")
(letztesElement, "Hallo" ++ "Welt") :: ([a] -> a, [Char])
```

Paare und Tupel (2)

Zugriffsfunktionen:

- `fst :: (a,b) -> a` liefert das linke Element eines Paares.
- `snd :: (a,b) -> b` liefert das rechte Element eines Paares.

```
*Main> fst (1,'A')  
1  
*Main> snd (1,'A')  
'A'  
*Main>
```

Pattern-Matching auf Paaren

```
eigenesFst (x,y) = x  
eigenesSnd (x,y) = y  
paarSumme  (x,y) = x+y
```

Tupel

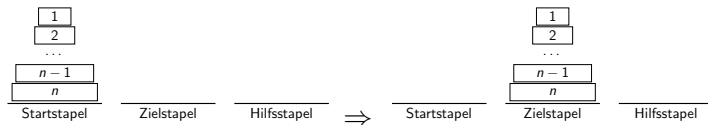
Wie Paare, aber mit mehr Komponenten.

```
*Main> :set +t
*Main> ('A',True,'B')
('A',True,'B')
it :: (Char, Bool, Char)
*Main> ([1,2,3],(True,'A',False,'B'),'B')
([1,2,3],(True,'A',False,'B'),'B')
it :: ([Integer], (Bool, Char, Bool, Char), Char)
```

Auch hier kann man Pattern verwenden:

```
erstes_aus_vier_tupel (w,x,y,z) = w
-- usw.
viertes_aus_vier_tupel (w,x,y,z) = z
```

Türme von Hanoi in Haskell



Pseudo-Algorithmus:

verschiebe(n , start, ziel, hilf)

1. Wenn $n > 1$, dann **verschiebe**($n-1$, start, hilf, ziel)
2. Schiebe Scheibe n von start auf ziel
3. Wenn $n > 1$, dann **verschiebe**($n-1$, hilf, ziel, start)

Modellierung in Haskell

- Stapel sind durchnummeriert
(am Anfang $\text{start} = 1$, $\text{ziel} = 2$, $\text{hilf} = 3$)
- Funktion `hanoi` erhält
 - Zahl n = Höhe des Stapels der verschoben werden soll
 - die drei Stapel
- Ausgabe: Liste von Zügen.
Ein Zug ist ein Paar (x, y)
= Schiebe oberste Scheibe vom Stapel x auf Stapel y

Die Funktion hanoi

```
-- Basisfall: 1 Scheibe verschieben
hanoi 1 start ziel hilf = [(start,ziel)]

-- Allgemeiner Fall:
hanoi n start ziel hilf =
  -- Schiebe den Turm der Hoehe n-1 von start zu hilf:
  (hanoi (n-1) start hilf ziel)    ++
  -- Schiebe n. Scheibe von start auf ziel:
  [(start,ziel)]                  ++
  -- Schiebe Turm der Hoehe n-1 von hilf auf ziel:
  (hanoi (n-1) hilf ziel start)
```

Starten mit

```
start_hanoi n = hanoi n 1 2 3
```


Ein Beispiel

```
*Main> start_hanoi 4  
[(1,3),(1,2),(3,2),(1,3),(2,1),(2,3),(1,3),(1,2),  
 (3,2),(3,1),(2,1),(3,2),(1,3),(1,2),(3,2)]
```