
Heutiges Thema...

Fortführung vom letzten Mal, d.h. mehr über Haskell, insbesondere über...

- Funktionen
 - ...und darüber wie man sie definieren/notieren kann
 - ↪ *Notationelle Alternativen (siehe Vorlesungsteil 1)*
 - ↪ *Funktionssignaturen, Funktionsausdrücke, Klammereinsparungsregeln (siehe Vorlesungsteil 2)*
 - ↪ Ergänzungen zu Funktionstermen & -signaturen, curryfizierte vs. uncurryfizierte Funktionsdarst.
 - ↪ Layout-Konventionen, Abseitsregel
 - Klassifikation von Rekursionstypen
 - Anmerkungen zu Effektivität und Effizienz
 - Komplexitätsklassen

Hinweis: Die beiden kursiv hervorgehobenen Punkte sind bereits in der Vorlesung am 04.10.2007 und 16.10.2007 besprochen worden.

Ergänzungen zu Funktionstermen (1)

Betrachten wir noch einmal die Funktion `add`:

```
add  :: Int -> (Int -> Int)
add m n = m+n
```

...und die Frage nach der “Existenz(berechtigung)” von

```
add 2 :: Int -> Int
```

...welches eine Funktion auf ganzen Zahlen ist, die ihr um 2 erhöhtes Argument als Resultat liefert.

Wir können diese Funktion `doubleInc` nennen...

Ergänzungen zu Funktionstermen (2)

...und in natürlicherweise wie folgt definieren:

```
doubleInc :: Int -> Int
doubleInc n = 2+n
```

Wir können die Definition von `doubleInc` aber auch auf die Funktion `(add 2)` abstützen:

```
doubleInc :: Int -> Int
doubleInc n = (add 2) n
```

...oder noch kürzer argumentlos (als Identität von Funktionen) einführen:

```
doubleInc :: Int -> Int
doubleInc = (add 2)
```

Beobachtung: `doubleInc` ist (nur noch) ein anderer Name für die Funktion `(add 2)`, die hier und in den obigen Bsp. nur der Deutlichkeit halber geklammert ist.

Ergänzungen zu Funktionstermen (3)

Vergleiche `doubleInc`, `add 2`

```
doubleInc :: Int -> Int
doubleInc = add 2
```

mit

```
\n -> add 2 n
```

Beobachtung: `doubleInc`, `add 2` und `\n -> add 2 n` sind...

- i.w. gleichwertige Formulierungen derselben Funktion
- i.w. dadurch unterschieden, dass `doubleInc` eine herkömmlich und im gewohnten Sinn benannte Funktion ist, wohingegen `(add 2)` und `(\n -> (add 2) n)` unbenannt, zumindest nicht im gewohnten Sinn mit einem Namen benannt sind; die Funktion `(\n -> (add 2) n)` speziell ist im Haskell-Jargon eine sog. *anonyme Funktion*!

“Erfahrenheits” - Faustregel

Die Implementierung einer Funktion wie `doubleInc`

- durch

```
doubleInc :: Int -> Int
doubleInc n = 2+n
```

...deutet darauf hin, dass vermutlich noch wenig Erfahrung mit funktionaler Programmierung vorliegt

- durch

```
doubleInc :: Int -> Int      doubleInc :: Int -> Int
doubleInc = (+) 2           doubleInc = (+2)  -- sog. operator section
```

...deutet darauf hin, dass vermutlich bereits mehr Erfahrung mit funktionaler Programmierung vorliegt

- durch

```
\n -> 2+n
```

...deutet gleichfalls darauf hin, dass schon mehr Erfahrung mit funktionaler Programmierung vorliegt, und darüberhinaus, dass in der konkreten Anwendungssituation ein Name, unter dem auf die Funktion mit der Bedeutung “`doubleInc`” zugegriffen werden könnte, keine Rolle spielt.

Als Ausblick... (1)

...ein kleines Beispiel schon jetzt:

```
map :: (Int -> Int) -> [Int] -> [Int]
map f [] = []
map f (x:xs) = (f x) : (map f xs)
```

Anwendung:

```
map (\n -> 2+n) [1,2,3] => [3,4,5]
```

...oder genausogut

```
map (add 2) [1,2,3] => [3,4,5]
map (2+) [1,2,3] => [3,4,5]
```

Machen Sie sich klar, dass die Typisierung von `add'` folgendes nicht zulässt:

```
map (add' 2) [1,2,3]
```

↪ später mehr dazu unter dem Stichwort “Funktionale”, speziell Funktionale auf Listen...

Als Ausblick... (2)

Als Beispiel aussagekräftiger und überzeugender:

```
map (\n -> 3*n+42) [1,2,3]    =>    [45,48,51]
```

Wird eine Funktion mit der Abbildungsvorschrift von ($\backslash n \rightarrow 3*n+42$) ansonsten nicht gebraucht, spart man sich durch Verwendung der anonymen Funktion wie oben die Deklaration einer ansonsten nur genau einmal benutzten Funktion wie `dreifachPlus42`:

```
dreifachPlus42 :: Int -> Int
dreifachPlus42 n = 3*n+42
```

```
map dreifachPlus42 [1,2,3]    =>    [45,48,51]
```

Ein anderer Nachtrag: Operatoren in Haskell

Operatoren in Haskell sind...

- ...grundsätzlich *Präfixoperatoren*, insbesondere alle selbst-deklarierten Operatoren (*vulgo*: selbstdeklarierte Funktionen)

Beispiele: `fac 5`, `imax 2 3`, `tripleMax 2 5 3`,...

- ...in einigen wenigen Fällen *Infixoperatoren*, dies gilt insbesondere für arithmetische Operatoren

Beispiele: `2+3`, `3*5`, `7-4`, `5^3`,...

Binäre Operatoren in Haskell: Infix- vs. Präfix

Für binäre Operatoren in Haskell gilt...

- Binäre Operatoren `bop`, die standardmäßig als...
 - Präfixoperatoren verwendet werden, können in der Form `'bop'` als Infixoperator verwendet werden
Beispiel: `2 'imax' 3` (statt standardmäßig `imax 2 3`)
 - Infixoperatoren verwendet werden, können in der Form `(bop)` als Präfixoperator verwendet werden
Beispiel: `(+) 2 3` (statt standardmäßig `2+3`)

Abschließend zu Funktionstermen (1)

Betrachten wir noch einmal die Funktionen `add` und `add'`:

`add` :: `Int -> (Int -> Int)`

`add'` :: `(Int,Int) -> Int`

Abschließend zu Funktionstermen (2)

...hier noch einmal zusammen mit ihren Implementierungen:

```
add  :: Int -> (Int -> Int)
```

```
add m n = m+n
```

```
add' :: (Int,Int) -> Int
```

```
add' (m,n) = m+n
```

Sprechweise: Die Funktion...

- add ist *curryfiziert*
- add' ist *uncurryfiziert*

Curryfiziert vs. uncurryfiziert (1)

Idee: ...ziehe die Art der Konsumation mehrerer Argumente zur Klassifizierung von Funktionen heran

Erfolgt die Konsumation mehrerer Argumente durch Funktionen...

- einzeln Argument für Argument: *curryfiziert*
- gebündelt als Tupel: *uncurryfiziert*

Beispiele:

Funktion `add` curryfiziert: `add 2 3` bzw. `(add 2) 3`

Funktion `add'` uncurryfiziert: `add' (2,3)`

Curryfiziert vs. uncurryfiziert (2)

Zentral sind die beiden *Funktionale* (synonym: *Funktionen höherer Ordnung*) `curry` und `uncurry`...

$$\text{curry} :: ((a,b) \rightarrow c) \rightarrow (a \rightarrow b \rightarrow c)$$
$$\text{curry } f \ x \ y = f \ (x,y)$$
$$\text{uncurry} :: (a \rightarrow b \rightarrow c) \rightarrow ((a,b) \rightarrow c)$$
$$\text{uncurry } g \ (x,y) = g \ x \ y$$

Intuitiv:

- *Curryfizieren* ersetzt Produkt-/Tupelbildung “ \times ” durch Funktionspfeil “ \rightarrow ”.
- *Decurryfizieren* ersetzt Funktionspfeil “ \rightarrow ” durch Produkt-/Tupelbildung “ \times ”.

Bemerkung: Die Bezeichnung geht auf Haskell B. Curry zurück, die (weit ältere) Idee auf M. Schönfinkel aus der Mitte der 20er-Jahre.

Curryfiziert vs. uncurryfiziert (3)

Die Funktionale `curry` und `uncurry` bilden...

- uncurryfiziert vorliegende Funktionen auf ihr curryfiziertes Gegenstück ab, d.h.

...für uncurryfiziertes $f :: (a,b) \rightarrow c$ ist

$\text{curry } f :: a \rightarrow b \rightarrow c$ curryfiziert.

- curryfiziert vorliegende Funktionen auf ihr uncurryfiziertes Gegenstück ab, d.h.

...für curryfiziertes $g :: a \rightarrow b \rightarrow c$ ist

$\text{uncurry } g :: (a,b) \rightarrow c$ uncurryfiziert.

$\text{curry} :: ((a,b) \rightarrow c) \rightarrow (a \rightarrow b \rightarrow c)$

$\text{curry } f \ x \ y = f \ (x,y)$

$\text{curry } f :: a \rightarrow b \rightarrow c$

$\text{uncurry} :: (a \rightarrow b \rightarrow c) \rightarrow ((a,b) \rightarrow c)$

$\text{uncurry } g \ (x,y) = g \ x \ y$

$\text{uncurry } g :: (a,b) \rightarrow c$

Im Beispiel...

```
add  :: Int -> (Int -> Int)
add m n = m+n
```

```
add' :: (Int,Int) -> Int
add' (m,n) = m+n
```

Damit gilt:

```
curry add' :: Int -> Int -> Int
```

```
uncurry add :: (Int,Int) -> Int
```

...und somit sind die folgenden Aufrufe gültige Aufrufe:

```
curry add' 17 4
  ⇒ add' (17,4) ⇒ 17+4 ⇒ 21
```

```
uncurry add (17,4)
  ⇒ add 17 4 ⇒ 17+4 ⇒ 21
```

Curryfiziert oder uncurryfiziert?

...das ist hier die Frage.

Zum einen...

- Geschmackssache (sozusagen eine notationelle Spielerei)
...sicher, auch das, aber: die Verwendung curryfzierter Formen ist in der Praxis vorherrschend
 $\rightsquigarrow f\ x, f\ x\ y, f\ x\ y\ z, \dots$ möglicherweise eleganter als
 $f\ x, f\ (x,y), f\ (x,y,z), \dots?$

Zum anderen (und weit wichtiger!) folgendes...

- Sachargument
 ...(nur) Funktionen in curryfzierter Darstellung unterstützen *partielle Auswertung*
 \rightsquigarrow Funktionen liefern Funktionen als Ergebnis!

Beispiel: `add 4711 :: Int -> Int`

...ist eine einstellige Funktion auf den ganzen Zahlen, die ihr Argument um 4711 erhöht als Resultat zurückliefert.

Layout-Konventionen für Haskell-Programme

Für die meisten gängigen Programmiersprachen gilt:

- Das Layout eines Programms hat einen Einfluss
 - auf seine Leserlichkeit, Verständlichkeit, Wartbarkeit
 - aber nicht auf seine Bedeutung

Für Haskell gilt das nicht!

- Das Layout eines Programms trägt in Haskell Bedeutung!
- Reminiszenz an Cobol, Fortran. Layoutabhängigkeit aber auch zu finden in modernen Sprachen wie z.B. occam.
- Für Haskell ist für diesen Aspekt des Sprachentwurfs eine grundsätzlich andere Entwurfsentscheidung getroffen worden als z.B. für Java, Pascal, C, etc.

Abseitsregel (engl. offside rule) (1)

...layout-abhängige Syntax als notationelle Besonderheit in Haskell

“Abseits”-Regel...

- Erstes Zeichen einer Deklaration (bzw. nach `let`, `where`):
...Startspalte neuer “Box” wird festgelegt
- Neue Zeile...
 - gegenüber der aktuellen Box nach rechts eingerückt:
...aktuelle Zeile wird fortgesetzt
 - genau am linken Rand der aktuellen Box: *...neue Deklaration wird eingeleitet*
 - weiter links als die aktuelle Box: *...aktuelle Box wird beendet (“Abseitssituation”)*

Ein Beispiel zur Abseitsregel (1)

Unsere Funktion `kVA` zur Berechnung von Volumen und Oberfläche einer Kugel mit Radius `r`:

```
kVA r =  
  ((4/3) * myPi * rcube r, 4 * myPi * square r)  
where  
myPi      = 3.14  
rcube x   = x *  
           square x  
  
square x  = x * x
```

...nicht schön, aber korrekt. Das Layout genügt der Abseitsregel von Haskell und damit den Layout-Konventionen.

Abseitsregel (2)

Graphische Veranschaulichung der Abseitsregel...

```
-----  
|  
| kVA r =  
| ((4/3) * myPi * rcube r, 4 * myPi * square r)  
| -----  
| |  
| | where  
| | myPi      = 3.14  
| | rcube x   = x *  
| | | square x  
| | ----->  
| ----->  
| -----  
| square x = x * x  
|  
| \/  
|
```

Layout-Konventionen

...bewährt hat es sich, eine Layout-Konvention nach folgendem Muster einzuhalten:

```
funName f1 f2... fn
| g1    = e1
| g2    = e2
...
| gk    = ek
```

```
funName f1 f2... fn
| diesIstEinGanz
  BesondersLanger
  Waechter
    = diesIstEinEbenso
      BesondersLangerAusdruck
| g2          = e2
...
| otherwise = ek
```

Verantwortung des Programmierers (1)

...die Auswahl einer angemessenen Notation. Vergleiche...

```
triMax :: Integer -> Integer -> Integer -> Integer
```

```
a) triMax = \p q r ->
    if p>=q then (if p>=r then p
                  else r)
                else (if q>=r then q
                      else r)
```

```
b) triMax p q r =
    if (p>=q) && (p>=r) then p
    else
    if (q>=p) && (q>=r) then q
    else r
```

```
c) triMax p q r
    | (p>=q) && (p>=r)    = p
    | (q>=p) && (q>=r)    = q
    | (r>=p) && (r>=q)    = r
```

Auswahlkriterium: Welche Variante lässt sich am einfachsten verstehen?

Verantwortung des Programmierers (2)

Hilfreich ist auch eine Richtschnur von C.A.R. Hoare:

Programme können grundsätzlich auf zwei Arten geschrieben werden:

- So einfach, dass sie offensichtlich keinen Fehler enthalten
- So kompliziert, dass sie keinen offensichtlichen Fehler enthalten

Es liegt am Programmierer, welchen Weg er einschlägt.

Rekursion

..speziell in funktionalen Sprachen

- Das zentrale Ausdrucksmittel/Sprachmittel, Wiederholungen auszudrücken. *Beachte:* Wir haben keine Schleifen in funktionalen Sprachen.
- Erlaubt oft sehr elegante Lösungen, oft wesentlich einfacher als schleifenbasierte Lösungen. Typisches Beispiel: *Türme von Hanoi*.
- Insgesamt so wichtig, dass eine *Klassifizierung* von Rekursionstypen angezeigt ist.

Eine solche Klassifizierung wird uns in der Folge beschäftigen.

Zuvor aber zwei Beispiele: Quicksort und Türme von Hanoi

Quicksort

...ein Beispiel, für das Rekursion auf eine elegante Lösung führt:

```
quickSort :: [Int] -> [Int]
```

```
quickSort [] = []
```

```
quickSort (x:xs) =
```

```
    quickSort [ y | y<-xs, y<=x ] ++
```

```
        [x] ++ quickSort [ y | y<-xs, y>x ]
```

Türme von Hanoi (1)

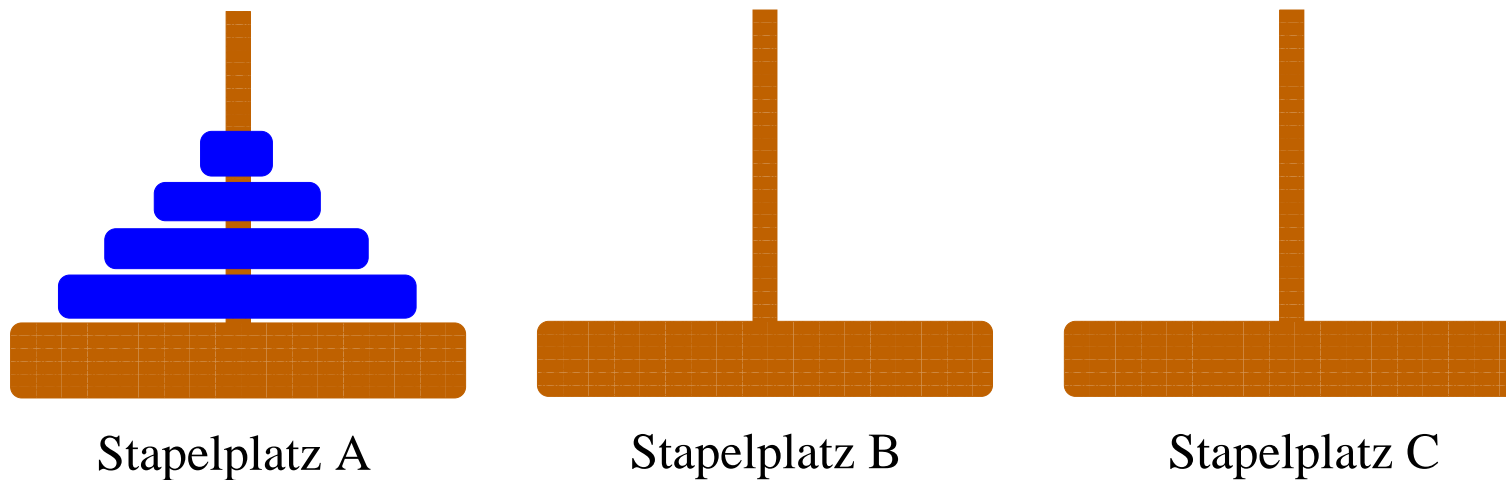
...ein anderes Beispiel, für das Rekursion auf eine elegante Lösung führt:

- *Ausgangssituation:*
Gegeben sind drei Stapelplätze A, B und C. Auf Platz A liegt ein Stapel unterschiedlich großer Scheiben, die ihrer Größe nach sortiert aufgeschichtet sind, d.h. die Größe der Scheiben nimmt von unten nach oben sukzessive ab.
- *Aufgabe:* Verlege unter Zuhilfenahme von Platz B den Stapel von Scheiben von Platz A auf Platz C, wobei Scheiben stets nur einzeln verlegt werden dürfen und zu keiner Zeit eine größere Scheibe oberhalb einer kleineren Scheibe auf einem der drei Plätze liegen darf.

Lösung: Übungsaufgabe

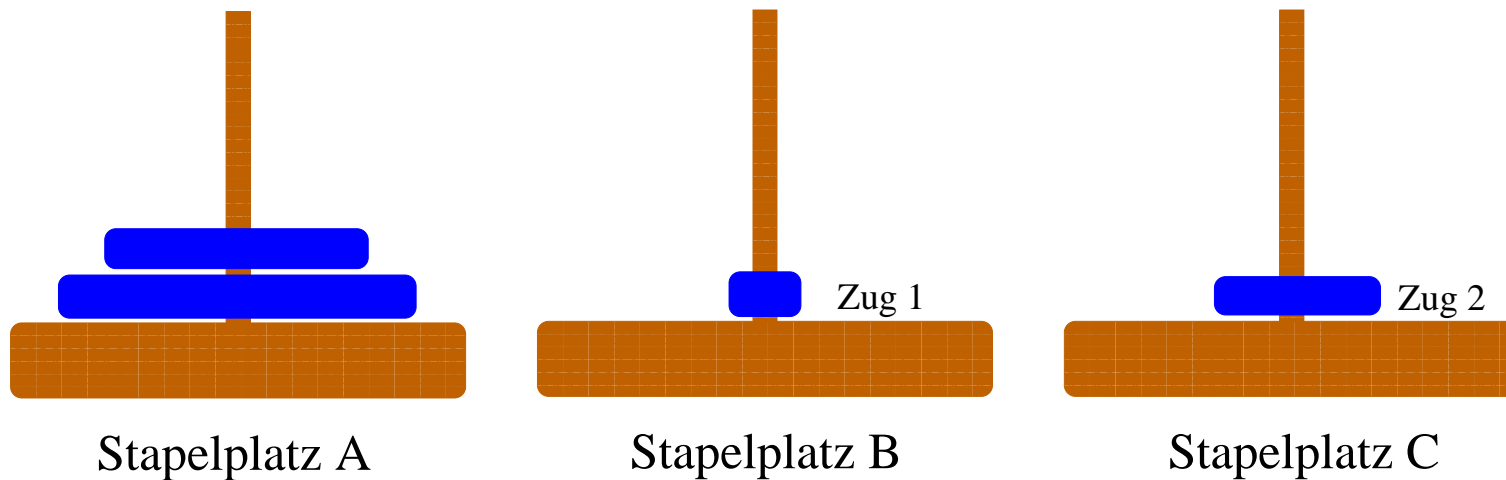
Türme von Hanoi (2)

Veranschaulichung:



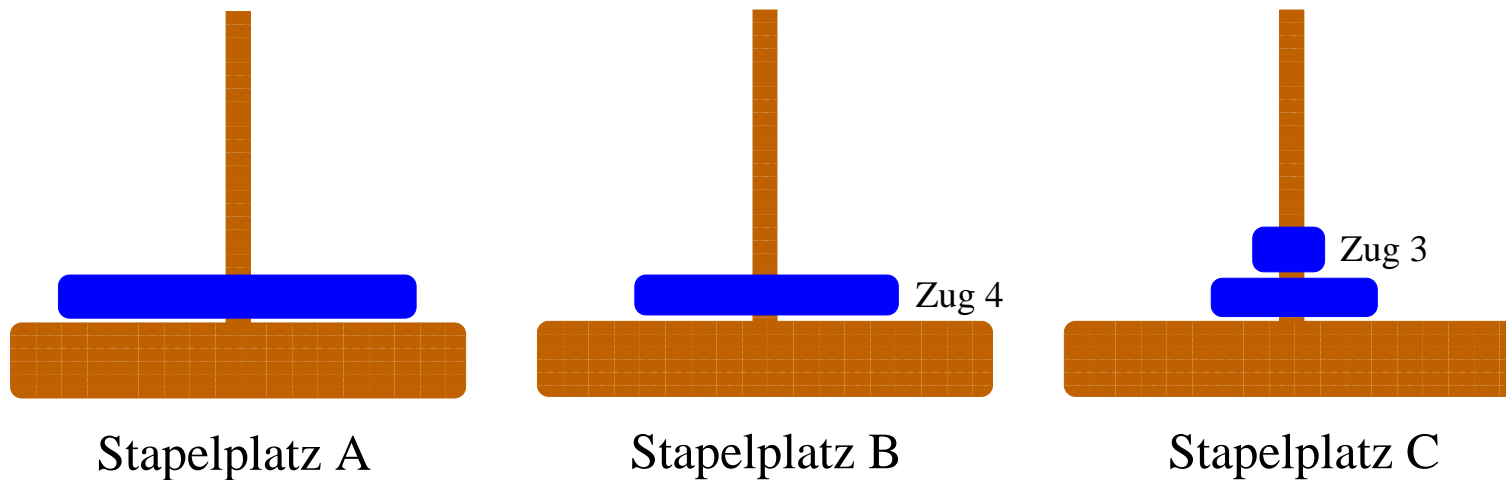
Türme von Hanoi (3)

Nach zwei Zügen:



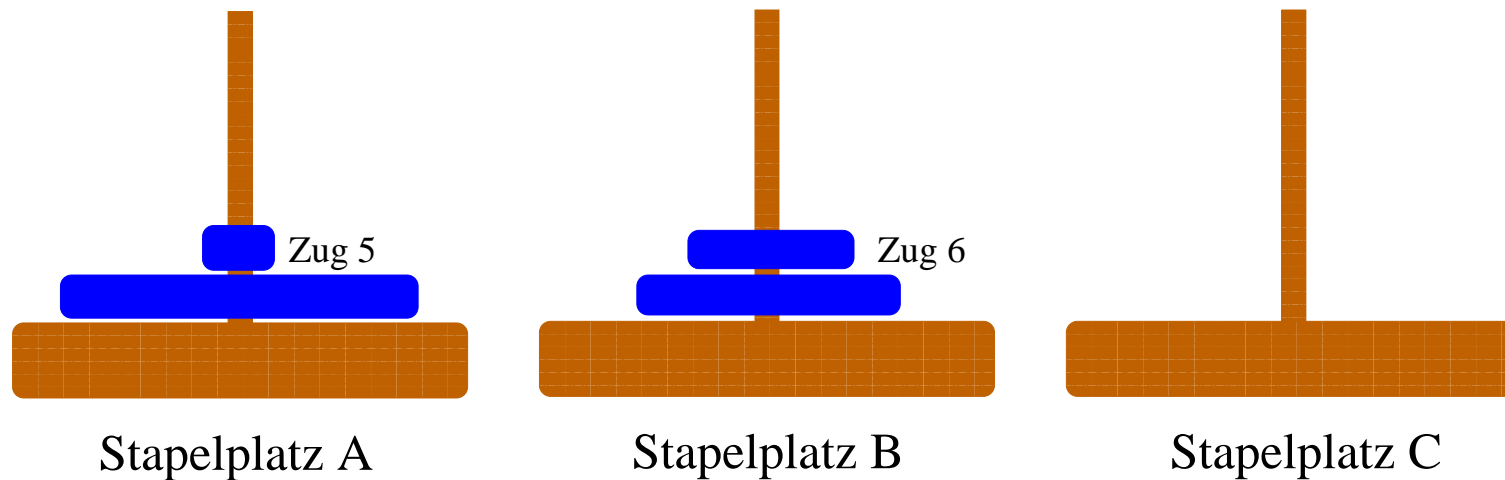
Türme von Hanoi (4)

Nach vier Zügen:



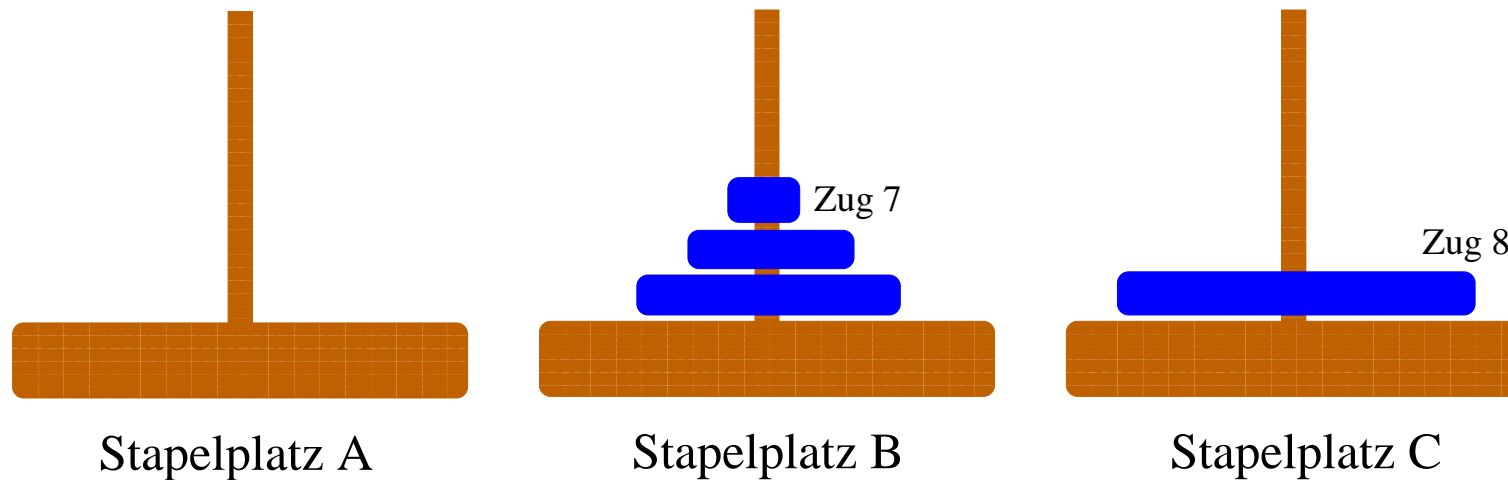
Türme von Hanoi (5)

Nach sechs Zügen:



Türme von Hanoi (6)

Nach acht Zügen:



Klassifikation der Rek.typen (1)

Generell...

...eine Rechenvorschrift heißt *rekursiv*, wenn sie in ihrem Rumpf (direkt oder indirekt) aufgerufen wird.

Dabei können wir unterscheiden...

- *Mikroskopische* Struktur
...betrachtet einzelne Rechenvorschriften und die syntaktische Gestalt der rekursiven Aufrufe
- *Makroskopische* Struktur
...betrachtet Systeme von Rechenvorschriften und ihre gegenseitigen Aufrufe

Rek.typen: Mikroskopische Struktur (2)

Üblich sind folgende Sprechweisen...

1. *Repetitive (schlichte) Rekursion*

...pro Zweig höchstens ein rekursiver Aufruf und zwar jeweils als äußerste Operation

Bsp:

```
ggt :: Integer -> Integer -> Integer
ggt m n
  | n == 0           = m
  | m >= n          = ggt (m-n) n
  | m < n           = ggt (n-m) m
```

Rek.typen: Mikroskopische Struktur (3)

2. *Lineare Rekursion*

...pro Zweig höchstens ein rekursiver Aufruf, jedoch nicht notwendig als äußerste Operation

Bsp:

```
powerThree :: Integer -> Integer
powerThree n
  | n == 0    = 1
  | n > 0    = 3 * powerThree (n-1)
```

Beachte: ...im Zweig $n > 0$ ist "*" die äußerste Operation, nicht powerThree!

Rek.typen: Mikroskopische Struktur (4)

3. *Geschachtelte Rekursion*

...rekursive Aufrufe enthalten rekursive Aufrufe als Argumente

Bsp:

```
fun91 :: Integer -> Integer
fun91 n
  | n > 100    = n - 10
  | n <= 100   = fun91(fun91(n+11))
```

Preisfrage: Warum heißt die Funktion wohl fun91?

Rek.typen: Mikroskopische Struktur (5)

4. Baumartige (kaskadenartige) Rekursion

...pro Zweig können mehrere rekursive Aufrufe nebeneinander vorkommen

Bsp:

```
binom :: (Integer,Integer) -> Integer
```

```
binom (n,k)
```

```
  | k==0 || n==k    = 1
```

```
  | otherwise       = binom (n-1,k-1) + binom (n-1,k)
```

Rek.typen: Makroskopische Struktur (6)

1. *Direkte Rekursion*

...entspricht Rekursion (Präzisierung!)

2. *Indirekte* oder auch *verschränkte (wechselweise) Rekursion*

...zwei oder mehr Funktionen rufen sich wechselseitig auf

Bsp:

```
isEven :: Integer -> Bool
```

```
isEven n
```

```
  | n == 0    = True
```

```
  | n > 0    = isOdd (n-1)
```

```
isOdd :: Integer -> Bool
```

```
isOdd n
```

```
  | n == 0    = False
```

```
  | n > 0    = isEven (n-1)
```

Anm. zu Effektivität & Effizienz (1)

Viele Probleme lassen sich...

- elegant rekursiv lösen (z.B. Türme von Hanoi)
- jedoch nicht immer effizient (\neq effektiv!)

Als Faustregel gilt...

- Unter Effizienzgesichtspunkten ist...
 - repetitive Rekursion am (kosten-) günstigsten
 - geschachtelte und baumartige Rekursion am ungünstigsten

Anm. zu Effektivität & Effizienz (2)

(Oft) folgende Abhilfe bei ineffizienten Implementierungen möglich:

~> Umformulieren! Ersetzen ungünstiger durch günstigere Rekursionsmuster!

Etwa...

- Rückführung *linearer* Rekursion auf *repetitive* Rekursion

Anm. zu Effektivität & Effizienz (3)

...am Beispiel der Fakultätsfunktion:

Naheliegende Formulierung mit *linearem* Rekursionsmuster...

```
fac :: Integer -> Integer
fac n = if n == 0 then 1 else (n * fac(n-1))
```

Effizientere Formulierung mit *repetitivem* Rekursionsmuster...

```
fac :: Integer -> Integer
fac n = facRep (n,1)

facRep :: (Integer,Integer) -> Integer
facRep (p,r) = if p == 0 then r else facRep (p-1,p*r)
```

~> “Trick” ...Rechnen auf Parameterposition!

Aber: Überlagerungen mit anderen Effekten sind möglich, so dass sich der Effizienzgewinn nicht realisiert! (Zur Übung: Wie ist das im obigen Beispiel?)

Kaskaden- oder baumartige Rekursion

...oft anfällig für unnötige Mehrfachberechnungen.

...in der Folge illustriert am Beispiel der Berechnung der Folge der *Fibonacci-Zahlen*:

Die Folge f_0, f_1, \dots der *Fibonacci-Zahlen* ist definiert durch...

$$f_0 = 0, f_1 = 1 \quad \text{und} \quad f_n = f_{n-1} + f_{n-2} \quad \text{für alle } n \geq 2$$

Fibonacci-Zahlen (1)

Die naheliegende Implementierung...

```
fib :: Integer -> Integer
fib n
  | n == 0    = 0
  | n == 1    = 1
  | otherwise = fib (n-1) + fib (n-2)
```

...führt auf kaskaden- bzw. baumartige Rekursion

~> ...und ist sehr, seeehr laaaangsaaaam (ausprobieren!)

Fibonacci-Zahlen (2)

Veranschaulichung ...durch manuelle Auswertung

fib 0 => 0 -- 1 Aufrufe von fib

fib 1 => 1 -- 1 Aufrufe von fib

fib 2 => fib 1 + fib 0
=> 1 + 0
=> 1 -- 3 Aufrufe von fib

fib 3 => fib 2 + fib 1
=> (fib 1 + fib 0) + 1
=> (1 + 0) + 1
=> 2 -- 5 Aufrufe von fib

Fibonacci-Zahlen (3)

```
fib 4 => fib 3 + fib 2
      => (fib 2 + fib 1) + (fib 1 + fib 0)
      => ((fib 1 + fib 0) + 1) + (1 + 0)
      => ((1 + 0) + 1) + (1 + 0)
      => 3 -- 9 Aufrufe von fib
```

```
fib 5 => fib 4 + fib 3
      => (fib 3 + fib 2) + (fib 2 + fib 1)
      => ((fib 2 + fib 1) + (fib 1 + fib 0))
          + ((fib 1 + fib 0) + 1)
      => (((fib 1 + fib 0) + 1) + (1 + 0)) + ((1 + 0) + 1)
      => (((1 + 0) + 1) + (1 + 0)) + ((1 + 0) + 1)
      => 5 -- 15 Aufrufe von fib
```

Fibonacci-Zahlen (4)

```
fib 8 => fib 7 + fib 6
      => (fib 6 + fib 5) + (fib 5 + fib 4)
      => ((fib 5 + fib 4) + (fib 4 + fib 3))
          + ((fib 4 + fib 3) + (fib 3 + fib 2))
      => (((fib 4 + fib 3) + (fib 3 + fib 2))
          + (fib 3 + fib 2) + (fib 2 + fib 1)))
          + (((fib 3 + fib 2) + (fib 2 + fib 1))
              + ((fib 2 + fib 1) + (fib 1 + fib 0)))
      => ... -- 60 Aufrufe von fib
```

Offensichtliche Probleme

- viele Mehrfachberechnungen
- exponentielles Wachstum!

Abhilfe

Programmiertechniken wie

- Dynamische Programmierung
- Memoization

Zentrale Idee:

- Speicherung und Wiederverwendung bereits berechneter (Teil-) Ergebnisse statt deren Wiederberechnung.

Komplexitätsklassen (1)

Nach P. Pepper. *Funktionale Programmierung in OPAL, ML, Haskell und Gofer*, 2. Auflage, 2003, Kapitel 11.

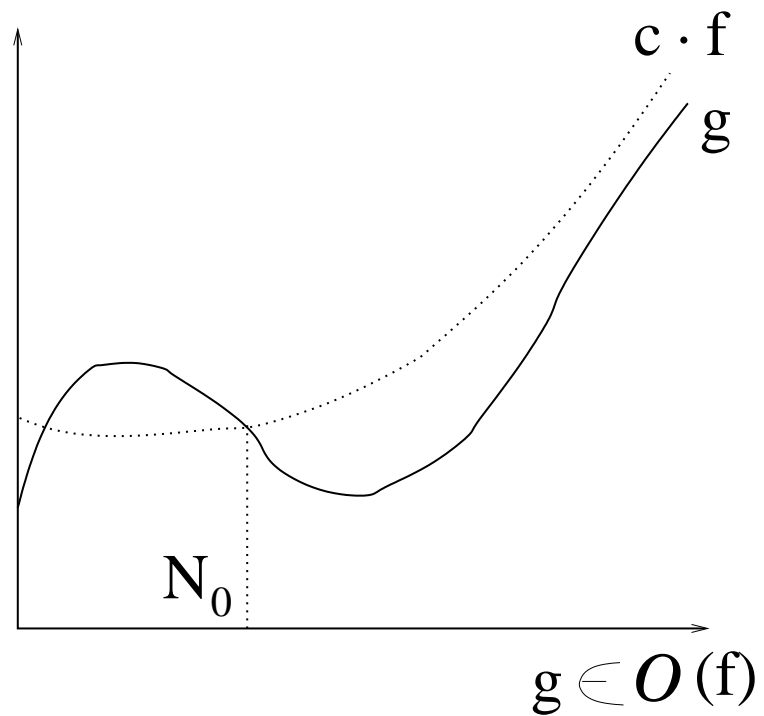
Erinnerung ...O-Notation

- Sei f eine Funktion $f : \alpha \rightarrow \mathbb{R}^+$ von einem gegebenen Datentyp α in die Menge der positiven reellen Zahlen. Dann ist die Klasse $\mathcal{O}(f)$ die Menge aller Funktionen, die “langsamer wachsen” als f :

$$\mathcal{O}(f) =_{df} \{h \mid h(n) \leq c * f(n) \text{ für eine positive Konstante } c \text{ und alle } n \geq N_0\}$$

Komplexitätsklassen (2)

Veranschaulichung:



Komplexitätsklassen (3)

Beispiele häufig auftretender Kostenfunktionen...

Kürzel	Aufwand	Intuition: <i>vertausendfachte Eingabe heißt...</i>
$O(c)$	konstant	... gleiche Arbeit
$O(\log n)$	logarithmisch	...nur zehnfache Arbeit
$O(n)$	linear	...auch vertausendfachte Arbeit
$O(n \log n)$	" $n \log n$ "	...zehntausendfache Arbeit
$O(n^2)$	quadratisch	...millionenfache Arbeit
$O(n^3)$	kubisch	...milliardenfache Arbeit
$O(n^c)$	polynomial	...gigantisch viel Arbeit (für großes c)
$O(2^n)$	exponentiell	...hoffnungslos

Komplexitätsklassen (4)

...und was wachsende Eingaben in realen Zeiten in der Praxis bedeuten können:

n	linear	quadratisch	kubisch	exponentiell
1	1 μ s	1 μ s	1 μ s	2 μ s
10	10 μ s	100 μ s	1 ms	1 ms
20	20 μ s	400 μ s	8 ms	1 s
30	30 μ s	900 μ s	27 ms	18 min
40	40 μ s	2 ms	64 ms	13 Tage
50	50 μ s	3 ms	125 ms	36 Jahre
60	60 μ s	4 ms	216 ms	36 560 Jahre
100	100 μ s	10 ms	1 sec	$4 * 10^{16}$ Jahre
1000	1 ms	1 sec	17 min	sehr, sehr lange...

Fazit

Die vorigen Folien machen deutlich...

- ...Effizienz ist wichtig!
- ...Rekursionsmuster haben einen erheblichen Einfluss darauf (vgl. baumartig-rekursive Implementierung der Fibonacci-Funktion. Beachte aber: Nicht das baumartige Rekursionsmuster ist ein Problem an sich, sondern die unnötigen Mehrfachberechnungen von Werten im Falle der Fibonacci-Funktion!)

Allerdings...

- Baumartig rekursive Funktionsdefinitionen bieten sich zur *Parallelisierung* an!
Stichwort: ...divide and conquer!

Zur Übung empfohlen...

- Wie könnte die Berechnung der Folge der Fibonacci-Zahlen

Struktur von Programmen

Programme funktionaler Programmiersprachen, speziell Haskell-Programme, sind zumeist

- Systeme (*wechselweiser*) *rekursiver* Rechenvorschriften, die sich *hierarchisch* oder/und *wechselweise* aufeinander abstützen.

Um sich über die *Struktur* solcher Systeme von Rechenvorschriften Klarheit zu verschaffen, ist neben der Untersuchung

- der *Rekursionstypen*

der beteiligten Rechenvorschriften insbesondere auch die Untersuchung

- ihrer *Aufrufgraphen*

geeignet.

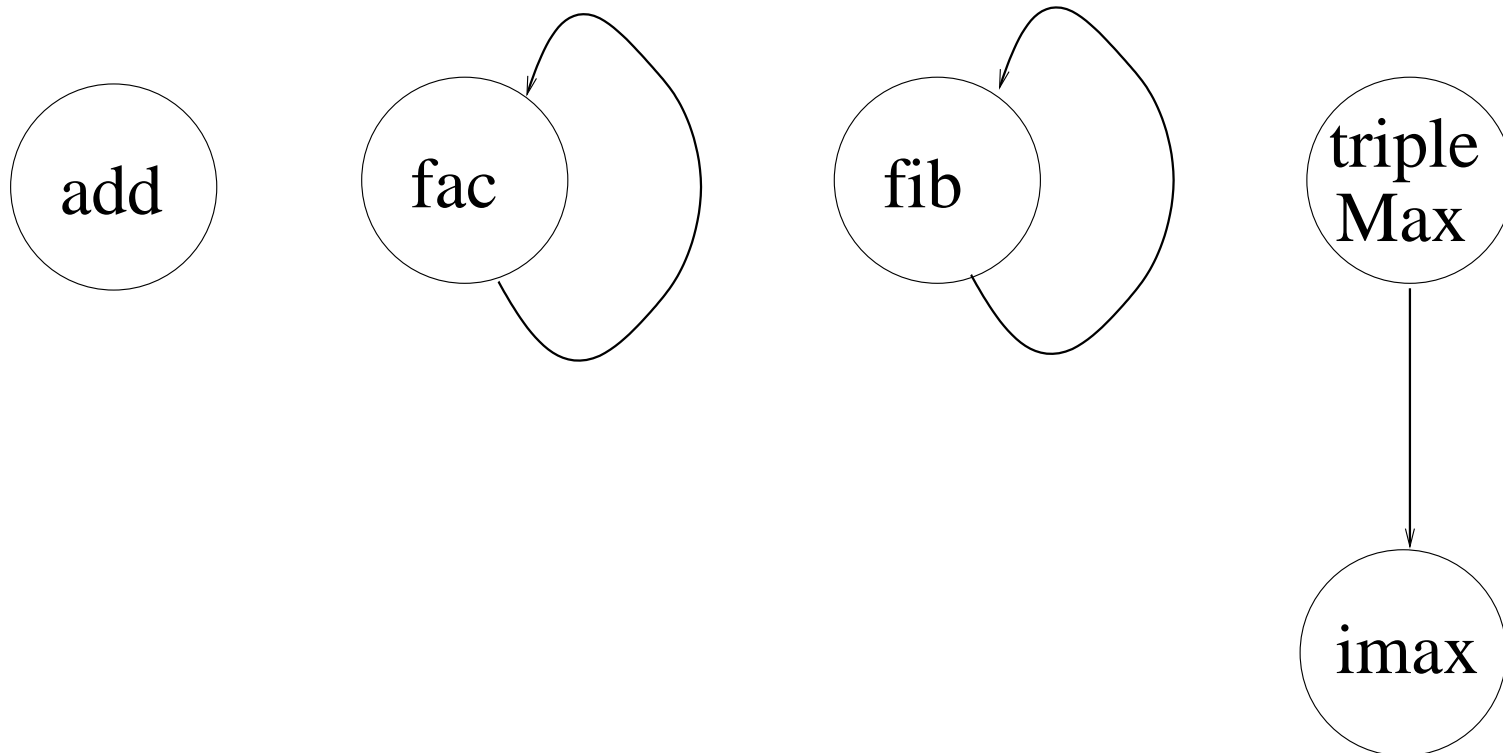
Aufrufgraphen

Der *Aufrufgraph* eines Systems S von Rechenvorschriften enthält

- einen *Knoten* für jede in S deklarierte Rechenvorschrift,
- eine gerichtete *Kante* vom Knoten f zum Knoten g genau dann, wenn im Rumpf der zu f gehörigen Rechenvorschrift die zu g gehörige Rechenvorschrift aufgerufen wird.

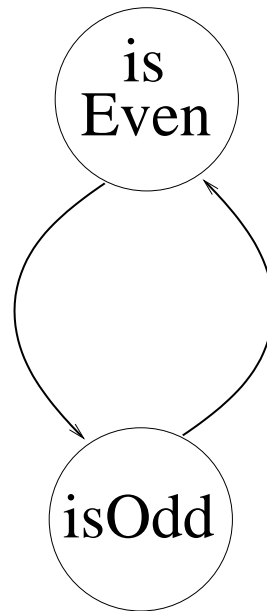
Beispiele von Aufrufgraphen (1)

...die Aufrufgraphen des Systems von Rechenvorschriften der Funktionen `add`, `fac`, `fib`, `imax` und `tripleMax`:



Beispiele von Aufrufgraphen (2)

...die Aufrufgraphen des Systems von Rechenvorschriften der Funktionen `isOdd` und `isEven`:



Beispiele von Aufrufgraphen (3)

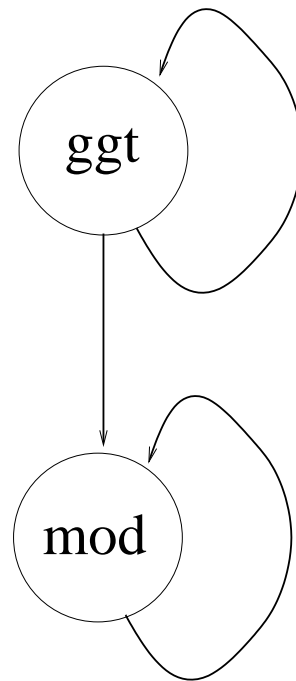
...das System von Rechenvorschriften der Funktionen ggt und mod:

```
ggt :: Int -> Int -> Int
ggt m n
  | n == 0 = m
  | n > 0  = ggt n (mod m n)
```

```
mod :: Int -> Int -> Int
mod m n
  | m < n  = m
  | m >= n = mod (m-n) n
```

Beispiele von Aufrufgraphen (3)

...und sein Aufrufgraph:



Auswertung von Aufrufgraphen

Aus dem Aufrufgraphen eines Systems von Rechenvorschriften ist u.a. ablesbar...

- *Direkte Rekursivität* einer Funktion: "Selbstkringel".
...z.B. bei den Aufrufgraphen der Funktionen `fac` und `fib`.
- *Wechselweise Rekursivität* zweier (oder mehrerer) Funktionen: Kreise (mit mehr als einer Kante)
...z.B. bei den Aufrufgraphen der Funktionen `isOdd` und `isEven`.
- *Direkte hierarchische Abstützung* einer Funktion auf eine andere: Es gibt eine Kante von Knoten f zu Knoten g , aber nicht umgekehrt.
...z.B. bei den Aufrufgraphen der Funktionen `tripleMax` und `imax`.
- *Indirekte hierarchische Abstützung* einer Funktion auf eine andere: Knoten g ist von Knoten f über eine Folge von Kanten erreichbar, aber nicht umgekehrt.
- *Wechselweise Abstützung*: Knoten g ist von Knoten f direkt oder indirekt über eine Folge von Kanten erreichbar und umgekehrt.
- *Unabhängigkeit/Isolation* einer Funktion: Knoten f hat (ggf. mit Ausnahme eines Selbstkringels) weder ein- noch ausgehende Kanten.
...z.B. bei den Aufrufgraphen der Funktionen `add`, `fac` und `fib`.
- ...

Vorschau auf die nächsten Vorlesungstermine... 1(2)

- Do, 25.10.2007: Keine Vorlesung, stattdessen Plenumsübung mit Besprechung von Aufgabenblatt 1.
- Di, 30.10.2007: Vorlesung von 13:00 Uhr s.t. bis 14:00 Uhr im Informatik-Hörsaal
- *Do, 01.11.2006: Keine Vorlesung! (Allerheiligen)*
- Do, 08.11.2007, Vorlesung von 16:30 Uhr bis 18:00 Uhr im Radinger-Hörsaal
- *Do, 15.11.2007: Keine Vorlesung! (Tag des Landespatrons, TU-weit LVA-frei)*
- Di, 20.11.2007, Vorlesung von 13:00 Uhr bis 14:00 Uhr im Informatik-Hörsaal

Vorschau auf die nächsten Vorlesungstermine... 2(2)

- *Do, 22.11.2007: Keine Vorlesung! (epilog, Diplomarbeitpräsentation, FfI-weit LVA-frei ab 15:00 Uhr)*
- Di, 27.11.2007, Vorlesung von 13:00 Uhr bis 14:00 Uhr im Informatik-Hörsaal
- Do, 29.11.2007, Vorlesung von 16:30 Uhr bis 18:00 Uhr im Radinger-Hörsaal