

---

# Verlängerte Abgabetermine

...aufgrund technischer Probleme auf der b1 werden die Abgabetermine für Aufgabenblatt 1 und 2 wie folgt verlängert:

- Aufgabenblatt 1:
  - Zweitabgabe, Di, 15.11.2005, 12:00 Uhr
- Aufgabenblatt 2:
  - Erstabgabe, Di, 15.11.2005, 12:00 Uhr
  - Zweitabgabe, Di, 22.11.2005, 12:00 Uhr

Weitere Informationen finden Sie auf der homepage zur LVA:

[http://www.complang.tuwien.ac.at/knoop/fp185161\\_ws0506](http://www.complang.tuwien.ac.at/knoop/fp185161_ws0506)

---

# Heutiges Thema...

Mehr über Haskell, insbesondere über...

- Funktionen
  - ...und darüber wie man sie definieren/notieren kann
    - ~> *Notationelle Alternativen (siehe Vorlesungsteil 1)*
    - ~> *Klammereinsparungsregeln in Funktionssignaturen*
    - ~> Ergänzungen zu Funktionssignaturen
    - ~> Abseitsregel und Layout-Konventionen
  - Klassifikation von Rekursionstypen
  - Anmerkungen zu Effektivität und Effizienz
  - Komplexitätsklassen

*Hinweis:* Die beiden kursiv hervorgehobenen Punkte sind bereits in der Vorlesung am 24.10.2005 besprochen worden.

---

# Erinnerung

Die Einsicht in den Unterschied

- von

`f :: Int -> Int -> Int -> Int`

...aufgrund der *Rechtsassoziativität* von  $\rightarrow$  abkürzend und gleichbedeutend mit der vollständig, aber nicht überflüssig geklammerten Version

`f :: (Int -> (Int -> (Int -> Int)))`

- und von

`f :: (((Int -> Int) -> Int) -> Int)`

ist *essentiell* und von absolut *zentraler* Bedeutung!

*Bewusst pointiert...*

Ohne diese Einsicht ist erfolgreiche Programmierung (speziell) im funktionalen Paradigma

- nicht möglich
- oder allenfalls Zufall!

---

# Motivation (1)

- Voriges Mal:  
...Konzentration auf Funktionsdeklarationen und ihre *Signaturen* bzw. *Typen*
- Dieses Mal:  
...Konzentration auf *Funktionsterme* und ihre *Signaturen* bzw. *Typen*

---

# Motivation (2)

## Tatsache:

Wir sind gewohnt, mit Ausdrücken der Art

`add 2 3`

umzugehen. (Auch wenn wir gewöhnlich  $2+3$  statt `add 2 3` schreiben.)

## Frage:

- Warum könnte es sinnvoll sein, auch mit (*scheinbar unvollständigen*) Ausdrücken wie

`add 2`

umzugehen?

- Entscheidend für die Antwort: Können wir einem Ausdruck wie `add 2` sinnvoll eine Bedeutung geben und wenn ja, welche?

---

# Funktionsterme und ihre Typen (1)

Betrachten wir die Funktion `add` zur Addition ganzer Zahlen noch einmal im Detail:

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

Dann sind die Ausdrücke `add`, `add 2` und `add 2 3` von den Typen:

```
add :: Int -> Int -> Int
add 2 :: Int -> Int
add 2 3 :: Int
```

---

# Funktionsterme und ihre Typen (2)

*Erinnerung:*

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

entspricht wg. vereinbarter Rechtsassoziativität von  $\rightarrow$

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

Somit *verbal* umschrieben:

- `add :: Int -> Int -> Int`  
...bezeichnet eine Funktion, die ganze Zahlen auf Funktionen von ganzen Zahlen in ganze Zahlen abbildet (*Rechtsassoziativität von  $\rightarrow$ !*).
- `add 2 :: Int -> Int`  
...bezeichnet eine Funktion, die ganze Zahlen auf ganze Zahlen abbildet.
- `add 2 3 :: Int`  
...bezeichnet eine ganze Zahl (nämlich 5).

---

# Funktionsterme und ihre Typen (3)

Damit haben wir eine Antwort auf unsere Ausgangsfrage...

- Warum könnte es sinnvoll sein, auch mit (*scheinbar unvollständigen*) Ausdrücken wie

add 2

umzugehen?

- Entscheidend für die Antwort: Können wir einem Ausdruck wie add 2 sinnvoll eine Bedeutung geben und wenn ja, welche?

## Nämlich:

Es ist sinnvoll, mit Ausdrücken der Art add 2 umzugehen, weil

- wir ihnen sinnvoll eine Bedeutung zuordnen können!
- im Falle von add 2:  
...add 2 bezeichnet eine Funktion auf ganzen Zahlen, die angewendet auf ein Argument dieses Argument um 2 erhöht als Resultat liefert.



---

## Funktionsterme und ihre Typen (4)

Betrachte auch folgendes Beispiel vom letzten Mal unter dem neuen Blickwinkel auf Funktionsterme und ihre Typen:

```
k :: (Int -> Int -> Int) -> Int -> Int -> Int
k h x y = h x y
```

Dann gilt:

```
k :: (Int -> Int -> Int) -> Int -> Int -> Int
k add :: Int -> Int -> Int
k add 2 :: Int -> Int
k add 2 3 :: Int
```

*Zur Übung:*

- Ausprobieren! In Hugs lässt sich mittels des Kommandos `:t <Ausdruck>` der Typ eines Ausdrucks bestimmen!

Bsp.: `:t k add 2` liefert `k add 2 :: Int -> Int`

---

# Funktionsterme und ihre Typen (5)

**Beachte:**

Der Ausdruck (Funktionsterm)

`k add 2 3`

steht kurz für

`((k add) 2) 3`

Analog stehen die Ausdrücke (Funktionsterme)

`k add`

`k add 2`

kurz für

`(k add)`

`((k add) 2)`

---

# Funktionsterme und ihre Typen (6)

*Beobachtung* (anhand des vorigen Beispiels):

- Funktionen in Haskell sind grundsätzlich *einstellig*!
- Wie die Funktion `k` zeigt, kann dieses Argument komplex sein, bei `k` z.B. eine Funktion, die ganze Zahlen auf Funktionen ganzer Zahlen in sich abbildet.

*Beachte:*

Die Sprechweise, Argument der Funktion `k` sei eine zweistellige Funktion auf ganzen Zahlen, ist *lax* und *unpräzise*, gleichwohl (aus Gründen der Einfachheit und Bequemlichkeit) üblich.

---

# Funktionsterme und ihre Typen (7)

*Konsequenz* aus voriger Beobachtung:

- Wann immer man nicht durch Klammerung etwas anderes erzwingt, ist (aufgrund der vereinbarten Rechtsassoziativität des Typoperators  $\rightarrow$ ) das “eine” Argument der in Haskell grundsätzlich einstelligen Funktionen von demjenigen Typ, der links vor dem ersten Vorkommen des Typoperators  $\rightarrow$  in der Funktionssignatur steht.
- Wann immer dies nicht erwünscht ist, muss dies durch explizite Klammerung in der Funktionssignatur ausgedrückt werden.

---

# Funktionsterme und ihre Typen (8)

*Beispiele:*

- *Keine Klammerung* ( $\rightsquigarrow$  Konvention greift!)

$f :: \text{Int} \rightarrow \text{Tree} \rightarrow \text{Graph} \rightarrow \dots$

f ist einstellige Funktion auf ganzen Zahlen, nämlich Int, die diese abbildet auf...

- *Explizite Klammerung* ( $\rightsquigarrow$  Konvention aufgehoben, wo gewünscht!)

$f :: (\text{Int} \rightarrow \text{Tree}) \rightarrow \text{Graph} \rightarrow \dots$

f ist einstellige Funktion auf Abbildungen von ganzen Zahlen auf Bäume, nämlich  $\text{Int} \rightarrow \text{Tree}$ , die diese abbildet auf...

*Hinweis:* Wie wir Bäume und Graphen in Haskell definieren können, lernen wir bald.

---

---

# Funktionsterme und ihre Typen (9)

Auch noch zu...

- ...
- Wann immer dies nicht erwünscht ist, muss dies durch explizite Klammerung in der Funktionssignatur erzwungen werden.

*Beispiele:*

- *Keine Klammerung*

$f :: \text{Int} \rightarrow \text{Tree} \rightarrow \text{Graph} \rightarrow \dots$

f ist einstellige Funktion auf ganzen Zahlen, nämlich Int, die diese abbildet auf...

- *Explizite Klammerung*

$f :: (\text{Int}, \text{Tree}) \rightarrow \text{Graph} \rightarrow \dots$

f ist einstellige Funktion auf Paaren aus ganzen Zahlen und Bäumen, nämlich (Int,Tree), die diese abbildet auf...

---

# Funktionsterme und ihre Typen (10)

Noch einmal zurück zum Beispiel der Funktion  $k$ :

$$k :: (\text{Int} \rightarrow \text{Int} \rightarrow \text{Int}) \rightarrow \text{Int} \rightarrow \text{Int} \rightarrow \text{Int}$$

... $k$  ist eine einstellige Funktion, die eine zweistellige Funktion auf ganzen Zahlen als *Argument* erwartet (*lax!*) und auf eine Funktion abbildet, die ganze Zahlen auf Funktionen ganzer Zahlen in sich abbildet.

Zur Deutlichkeit die Signatur von  $k$  auch noch einmal vollständig, aber nicht überflüssig geklammert:

$$k :: ((\text{Int} \rightarrow (\text{Int} \rightarrow \text{Int})) \rightarrow (\text{Int} \rightarrow (\text{Int} \rightarrow \text{Int})))$$

---

# Funktionsterme und ihre Typen (11)

Das Beispiel von `k` fortgesetzt:

```
k add :: Int -> Int -> Int
```

...`k add` ist eine einstellige Funktion, die ganze Zahlen auf Funktionen ganzer Zahlen in sich abbildet.

Zur Deutlichkeit auch hier noch einmal vollständig, aber nicht überflüssig geklammert:

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



---

# Funktionsterme und ihre Typen (12)

Das Beispiel von `k` weiter fortgesetzt:

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

...`k add 2` ist eine einstellige Funktion, die ganze Zahlen in sich abbildet.

Zur Deutlichkeit auch hier wieder vollständig, aber nicht überflüssig geklammert:

```
((k add) 2) :: (Int -> Int)
```

---

## Funktionsterme und ihre Typen (13)

Das Beispiel von `k` abschließend fortgesetzt:

```
k add 2 3 :: Int
```

`k add 2 3` bezeichnet ganze Zahl; in diesem Falle 5.

Zur Deutlichkeit auch dieser Funktionsterm vollständig, aber nicht überflüssig geklammert:

```
((k add) 2) 3 :: Int
```

---

# Wichtige Vereinbarungen in Haskell

Wenn in Haskell durch Klammerung nichts anderes ausgedrückt wird, gilt für

- Funktionssignaturen *Rechtsassoziativität*, d.h.

```
k :: (Int -> Int -> Int) -> Int -> Int -> Int
```

*steht für*

```
k :: ((Int -> (Int -> Int)) -> (Int -> (Int -> Int)))
```

- Funktionsterme *Linksassoziativität*, d.h.

```
k add 2 3 :: Int
```

*steht für*

```
((k add) 2) 3 :: Int
```

als vereinbart!

---

# Zum Abschluss des Signaturthemas (1)

Frage:

- Warum mag uns ein Ausdruck wie

`add 2`

“unvollständig” erscheinen?

---

## Zum Abschluss des Signaturthemas (2)

...weil wir im Zusammenhang mit der Addition tatsächlich weniger an Ausdrücke der Form

add 2 3

als vielmehr an Ausdrücke der Form

add' (2,3)

gewohnt sind!

*Erinnern Sie sich?*

$$+ : \mathbb{Z} \times \mathbb{Z} \rightarrow \mathbb{Z}$$

---

## Zum Abschluss des Signaturthemas (3)

Der Unterschied liegt in den Signaturen der Funktionen `add` und `add'`:

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

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

Mit diesen Signaturen von `add` und `add'` sind einige Beispiele...

- *korrekter* Aufrufe:

```
add 2 3          => 5 :: Int
add' (2,3)       => 5 :: Int
add 2           :: Int -> Int
```

- *inkorrekt* Aufrufe:

```
add (2,3)
add' 2 3    -- beachte: add' 2 3 steht kurz fuer (add' 2) 3
add' 2
```

---

## Zum Abschluss des Signaturthemas (4)

Mithin...

- ...die Funktionen `+` und `add'` sind echte *zweistellige* Funktionen

wohingegen...

- ...die Funktion `add` einstellig ist und nur aufgrund der Klammereinsparungsregeln scheinbar ebenfalls "zweistellige" Aufrufe zulässt:

`add 17 4`

*Aber:* `add 17 4` steht kurz für `(add 17) 4`. Die geklammerte Variante macht deutlich: Ein Argument nach dem anderen und nur eines zur Zeit...

---

# Fazit zum Signaturthema (1)

Wir müssen nicht nur sorgfältig

- zwischen

$f :: \text{Int} \rightarrow \text{Int} \rightarrow \text{Int}$

...aufgrund der *Rechtsassoziativität* von  $\rightarrow$  abkürzend und gleichbedeutend ist mit

$f :: \text{Int} \rightarrow (\text{Int} \rightarrow \text{Int})$

- und

$f :: (\text{Int} \rightarrow \text{Int}) \rightarrow \text{Int}$

unterscheiden, sondern ebenso sorgfältig auch

- zwischen

$f :: (\text{Int}, \text{Int}) \rightarrow \text{Int}$

- und

$f :: \text{Int} \rightarrow (\text{Int}, \text{Int})$

und nicht zuletzt zwischen allen vier Varianten insgesamt!



---

## Fazit zum Signaturthema (2)

Mithin, schreiben Sie

$f :: \text{Int} \rightarrow \text{Int} \rightarrow \text{Int}$

nur, wenn Sie auch wirklich

$f :: \text{Int} \rightarrow (\text{Int} \rightarrow \text{Int})$

meinen und nicht etwa

$f :: (\text{Int} \rightarrow \text{Int}) \rightarrow \text{Int}$

oder

$f :: (\text{Int}, \text{Int}) \rightarrow \text{Int}$

oder

$f :: \text{Int} \rightarrow (\text{Int}, \text{Int})$

*Es macht einen Unterschied!*

---

## Und deshalb die Bitte:

- Gehen Sie die vorausgegangenen Beispiele noch einmal Punkt für Punkt durch und vergewissern Sie sich, dass Sie sie im Detail verstanden haben.

Das ist wichtig, weil...

- dieses Verständnis und der aus diesem Verständnis heraus mögliche kompetente und selbstverständliche Umgang mit komplexen Funktionssignaturen und Funktionstermen essentiell für alles weitere ist!

---

## Ein kurzer Ausblick

Wir werden auf die Unterschiede und die Vor- und Nachteile von Deklarationen in der Art von

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

und

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

im Verlauf der Vorlesung unter den Schlagwörtern *Funktionen höherer Ordnung*, *Currifizierung*, *Funktionen als "first class citizens"* wieder zurückkommen.

Behalten Sie die Begriffe im Hinterkopf und blättern Sie zu gegebener Zeit in Ihren Unterlagen wieder hierher zurück.

---

# 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.

---

# Türme von Hanoi (1)

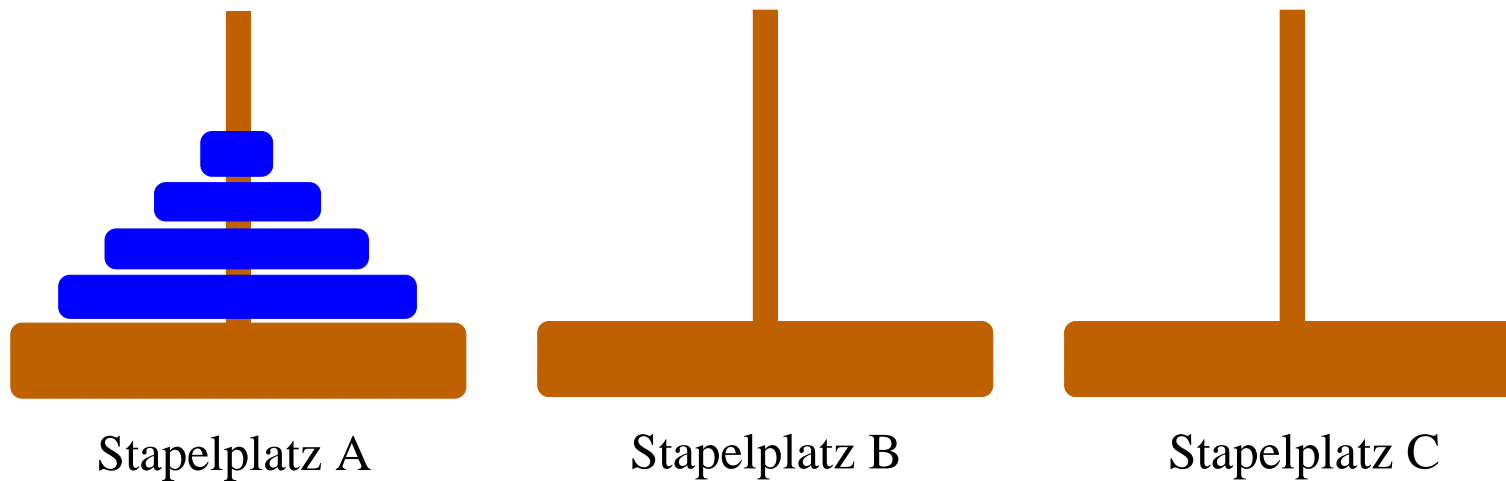
- *Ausgangssituation:*  
Gegeben sind drei Stapelplätze A, B und C. Auf Platz A liegt ein Stapel unterschiedlich großer Scheiben, die ihrer Größe nach sortiert aufgeschichtet sind, d.h. die Größe der Scheiben nimmt von unten nach oben sukzessive ab.
- *Aufgabe:* Verlege unter Zuhilfenahme von Platz B 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)

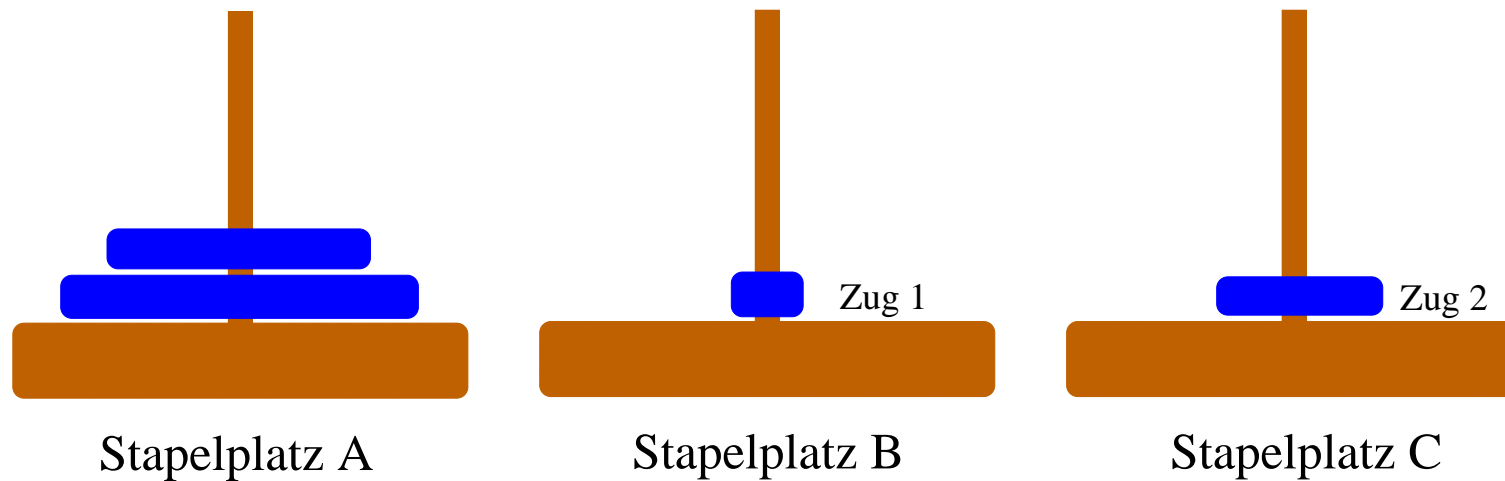
Veranschaulichung:



---

# Türme von Hanoi (3)

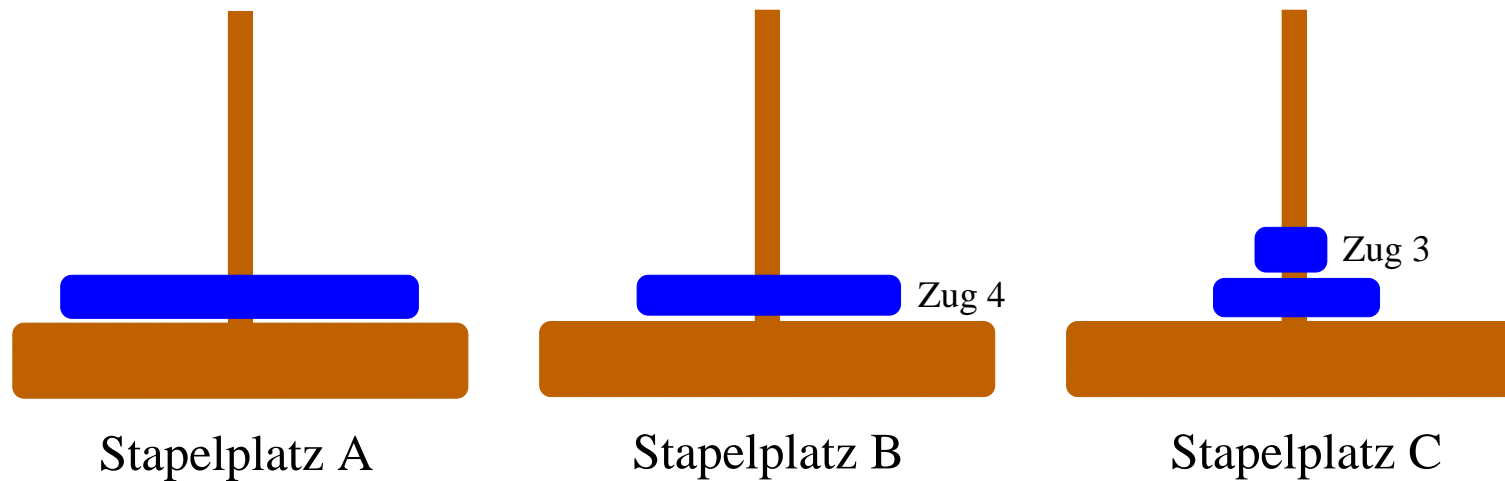
Nach zwei Zügen:



---

# Türme von Hanoi (4)

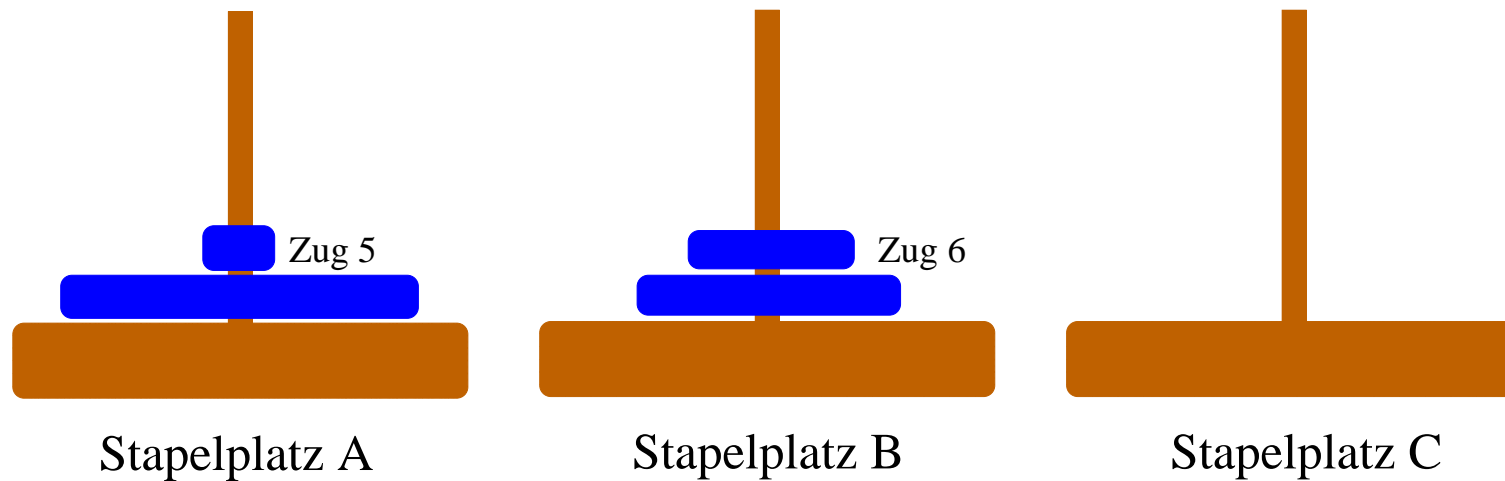
Nach vier Zügen:



---

# Türme von Hanoi (5)

Nach sechs Zügen:

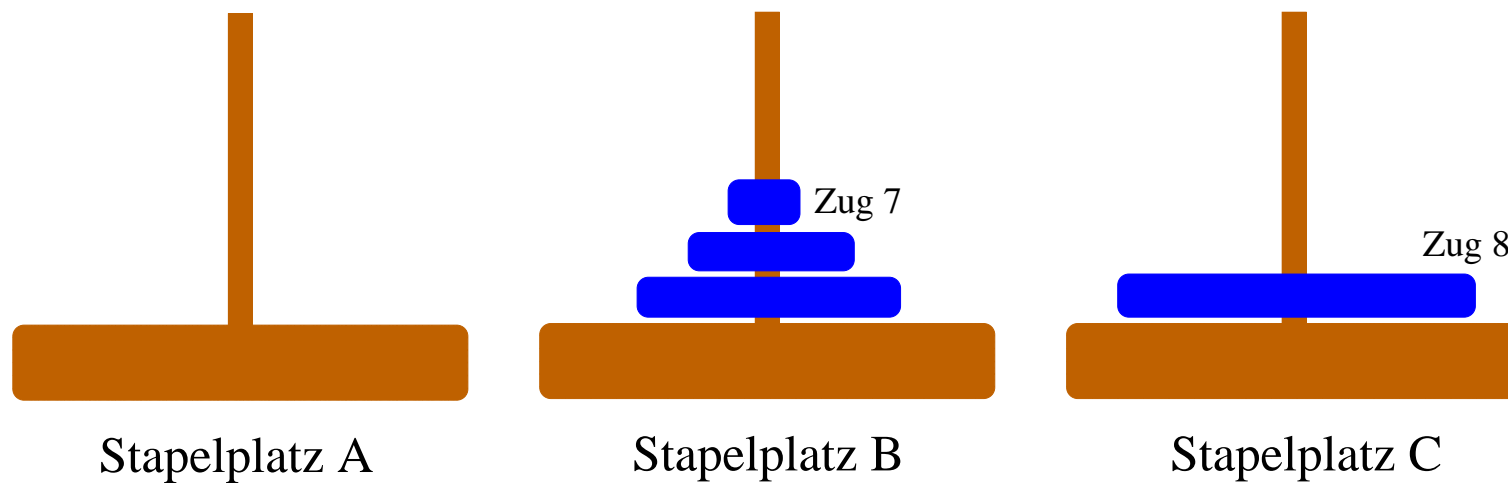




---

# Türme von Hanoi (6)

Nach acht Zügen:



---

# Quicksort

...auch 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 ]
```

---

# 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!

---

# Kaskaden- oder baumartige Rekursion

...komplexitätsmäßig ein ungünstiges Rekursionsmuster

...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!



---

# Komplexitätsklassen (1)

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

*Erinnerung ...O-Notation*

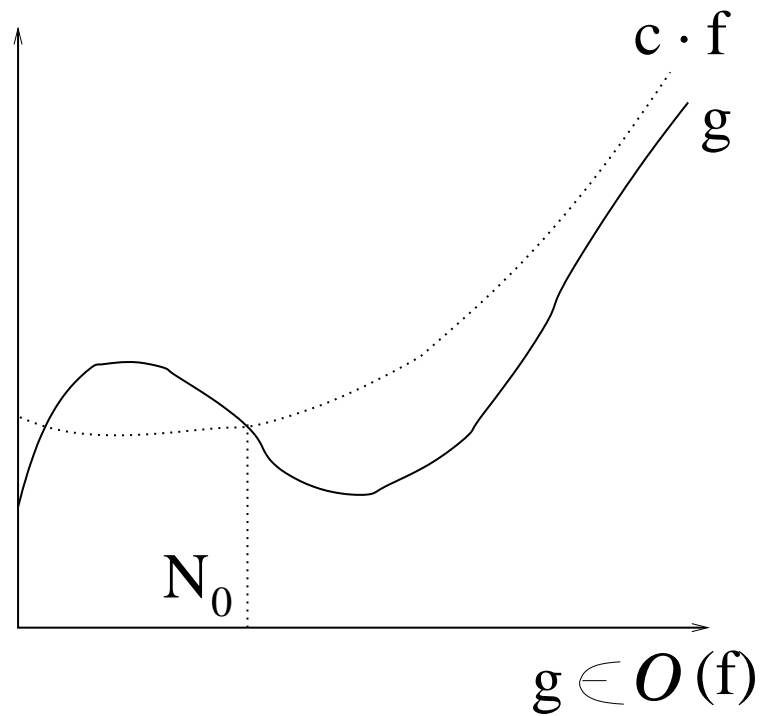
- Sei  $f$  eine Funktion  $f : \alpha \rightarrow \mathbb{R}^+$  von einem gegebenen Datentyp  $\alpha$  in die Menge der positiven reellen Zahlen. Dann ist die Klasse  $\mathcal{O}(f)$  die Menge aller Funktionen, die “langsamer wachsen” als  $f$ :

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

---

# 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 $\mu$ s	1 $\mu$ s	1 $\mu$ s	2 $\mu$ s
10	10 $\mu$ s	100 $\mu$ s	1 ms	1 ms
20	20 $\mu$ s	400 $\mu$ s	8 ms	1 s
30	30 $\mu$ s	900 $\mu$ s	27 ms	18 min
40	40 $\mu$ s	2 ms	64 ms	13 Tage
50	50 $\mu$ s	3 ms	125 ms	36 Jahre
60	60 $\mu$ s	4 ms	216 ms	36 560 Jahre
100	100 $\mu$ 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 (siehe baumartige Rekursion am Bsp. der Fibonacci-Zahlen)

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 effizienter realisiert werden?

---

# 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

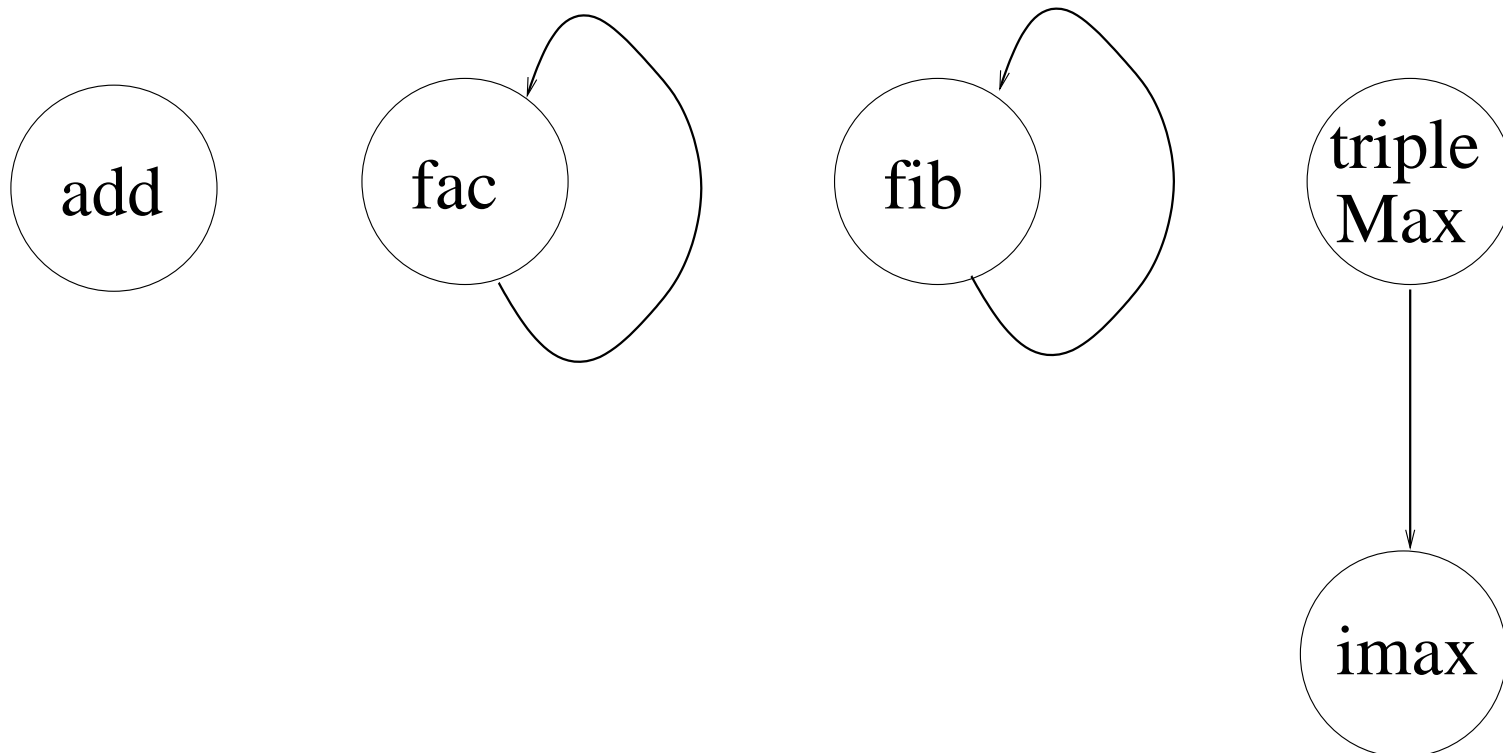
Der *Aufrufgraph* eines Systems  $S$  von Rechenvorschriften enthält

- 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`:

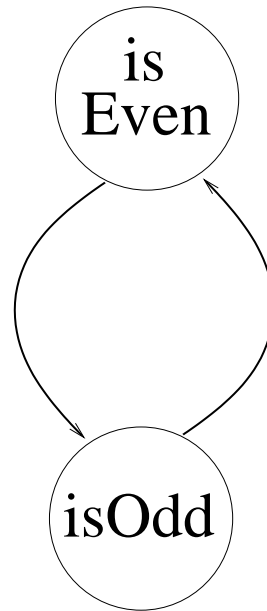




---

## 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:

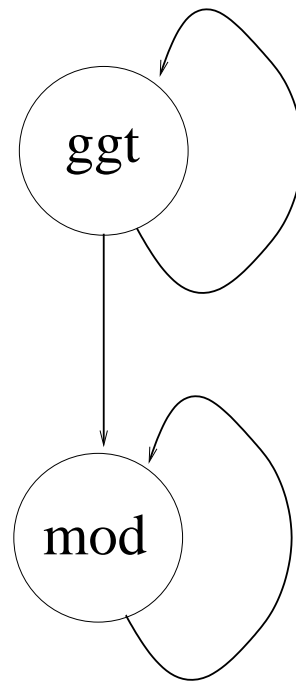
```
ggt :: Int -> Int -> Int
ggt m n
  | n == 0 = m
  | n > 0  = ggt n (mod m n)
```

```
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`.
- ...

---

# Verlängerte Abgabetermine

...aufgrund technischer Probleme auf der b1 werden die Abgabetermine für Aufgabenblatt 1 und 2 wie folgt verlängert:

- Aufgabenblatt 1:
  - Zweitabgabe, Di, 15.11.2005, 12:00 Uhr
- Aufgabenblatt 2:
  - Erstabgabe, Di, 15.11.2005, 12:00 Uhr
  - Zweitabgabe, Di, 22.11.2005, 12:00 Uhr

Weitere Informationen finden Sie auf der homepage zur LVA:

[http://www.complang.tuwien.ac.at/knoop/fp185161\\_ws0506](http://www.complang.tuwien.ac.at/knoop/fp185161_ws0506)

---

## Zum dritten Aufgabenblatt...

- ...erhältlich ab morgen im Web unter folgender URL  
[http://www.complang.tuwien.ac.at/knoop/fp185161\\_ws0506.html](http://www.complang.tuwien.ac.at/knoop/fp185161_ws0506.html)
- Erstabgabe: Mi, den 16.11.2005, 12:00 Uhr
- Zweitabgabe: Mi, den 23.11.2005, 12:00 Uhr

### **Vorschau:**

Ausgabe des...

- vierten Aufgabenblatts: Mi, den 16.11.2005  
...Abgabetermine: Mi, den 23.11.2005, und Mi, den 30.11.2005
- fünften Aufgabenblatts: Mi, den 23.11.2005  
...Abgabetermine: Mi, den 30.11.2005, und Mi, den 07.12.2005

---

## Vorschau auf die nächsten Vorlesungstermine...

- *Do, 10.11.2005: Keine Vorlesung!*
- Do, 17.11.2005, Vorlesung von 16:30 Uhr bis 18:00 Uhr im Radinger-Hörsaal
- Do, 24.11.2005, Vorlesung von 16:30 Uhr bis 18:00 Uhr im Radinger-Hörsaal
- Do, 01.12.2005, Vorlesung von 16:30 Uhr bis 18:00 Uhr im Radinger-Hörsaal