Mehr über Haskell...

- Funktionen
  - ...und wie man sie definieren/notieren kann
    - ⇝ Notationelle Alternativen
    - ⇝ Abseitsregel und Layout-Konventionen
  - Klassifikation von Rekursionstypen
  - Anmerkungen zu Effektivität und Effizienz
- Datenstrukturen: Zusammengesetzte Datentypen
  - Tupel und Listen
  - Funktionen über Tupeln und Listen

Fkt. in Haskell: Notat. Alternativen (1)

...am Beispiel der Fakultätsfunktion.

fact :: Integer -> Integer

(1) *λ-artig*

```
fact = \n -> (if n == 1 then 1 else (n * fact (n - 1)))
```

(2) *Gleichungsorientiert*

```
fact n = if n == 1 then 1 else (n * fact (n - 1))
```

(3) In Form *"bedingter Gleichungen"*

```
fact n
  | n == 1      = 1
  | otherwise   = n * fact (n - 1)
```

⇝ Beachte: (3) "Normalfall" in Haskell

Fkt. in Haskell: Notat. Alternativen (2)

...am Beispiel weiterer Funktionen.

kVA :: Float -> (Float, Float)

Mittels *lokaler Deklarationen*...

(4a) where-Konstrukt

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


Fkt. in Haskell: Notat. Alternativen (3)

bzw...

(4b) let-Konstrukt

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

Fkt.in Haskell: Notat. Alternativen (4)

In einer Zeile...

(5a)

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

(5b)

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


Fkt. in Haskell: Notat. Alternativen (5)

Speziallfall: *binäre* (*zweistellige*) Funktionen...

```
imax :: Integer -> Integer -> Integer
imax p q
  | p >= q     = p
  | otherwise  = q

tripleMax :: Integer -> Integer -> Integer -> Integer
tripleMax p q r
  | (imax p q == p) && (p `imax` r == p)   = p
  | ...
  | otherwise                              = r
```

...`imax` in `tripleMax` als Präfix- und als Infixoperator verwandt

Fkt. in Haskell: Notat. Alternativen (6)

Musterbasiert...

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

capVowels :: Char -> Char
capVowels 'a' = 'A'
capVowels 'e' = 'E'
capVowels 'i' = 'I'
capVowels 'o' = 'O'
capVowels 'u' = 'U'
capVowels  c  =  c
```


Fkt. in Haskell: Notat. Alternativen (7)

Mittels *case*-Ausdrucks...

```
capVowels :: Char -> Char
capVowels letter
  = case letter of
    'a'     -> 'A'
    'e'     -> 'E'
    'i'     -> 'I'
    'o'     -> 'O'
    'u'     -> 'U'
    letter  -> letter
```

```
deCapVowels :: Char -> Char
deCapVowels letter
  = case letter of
    'A'        -> 'a'
    'E'        -> 'e'
    'I'        -> 'i'
    'O'        -> 'o'
    'U'        -> 'u'
    otherwise  -> letter
```

Fkt. in Haskell: Notat. Alternativen (8)

Mittels *Muster* und *"wild cards"*...

```
add :: Integer -> Integer -> Integer
add n 0 = n
add 0 n = n
add m n = m+n

mult :: Integer -> Integer -> Integer
mult _ 0 = 0
mult 0 _ = 0
mult m n = m*n
```


Muster können (u.a.) sein...

- *Werte* (z.B. 0, 'c', True)
  ...ein Argument "passt" auf das Muster, wenn es vom entsprechenden Wert ist.
- *Variablen* (z.B. n)
  ...jedes Argument passt.
- *Wild card* "_"
  ...jedes Argument passt.
- ...

⇝ mehr über Muster und musterbasierte Funktionsdefinitionen später...

Abseitsregel (engl. offside rule)

...layout-abhängige Syntax als notationelle Besonderheit

"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...

```
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 sich beispielsweise:

```
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...

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
```


Klassifikation der Rek.typen (1)

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

Generell...

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

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)
```


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))
```


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 wechselweise 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)

Die Ackermann-Funktion – ein "berühmtberüchtigtes" Bsp.:

```
ack ::  (Integer,Integer) -> Integer
ack (m,n)
  | m == 0                = n+1
  | (m > 0) && (n == 0)   = ack (m-1,1)
  | (m > 0) && (n /= 0)   = ack (m-1,ack(m,n-1))
```

(*Oft*) *folgende Abhilfe möglich*:

$\rightsquigarrow$ Umformulieren! Ersetzen ungünstiger durch günstigere Rekursionsmuster!
Beispiel: Fibonacci-Funktion (siehe Aufgabenblatt 1 und 2!)

Vorschau auf die nächsten Vorlesungstermine...

- **Heute**, Di, 18.11.2003, 18-19 Uhr, Treitl-Hörsaal, 3. Zusatzvorlesung
- Ab Di, 25.11.2003, jeden Dienstag von 13-14 Uhr im Treitl-Hörsaal