Heutiges Thema...

Mehr über Haskell...

- Funktionen
  - ...und wie man sie definieren/notieren kann
    - $\leadsto$ Notationelle Alternativen
    - $\leadsto$ Abseitsregel und Layout-Konventionen
  - Klassifikation von Rekursionstypen
  - Anmerkungen zu Effektivität und Effizienz
  - Komplexitätsklassen

Fkt. in Haskell: Notat. Alternativen (1)

...am Beispiel der Fakultätsfunktion.

fact :: Integer -> Integer

(1) *λ-artig*

fact = \n -> (if n == 0 then 1 else (n ∗ fact (n − 1)))

(2) *Gleichungsorientiert*

fact n = if n == 0 then 1 else (n ∗ fact (n − 1))

(3) In Form *"bedingter Gleichungen"*

```
fact n
  | n == 0     = 1
  | otherwise  = n * fact (n - 1)
```

$\leadsto$ *Beachte*: (3) "Normalfall" in Haskell

Fkt. in Haskell: Notat. Alternativen (2)

...am Beispiel weiterer Funktionen.

kVA :: Float -> (Float, Float) – Berechnung von Volumen (V) und Fläche (A) einer Kugel (k)

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)

Spezialfall: *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 (sinnvoll für nicht zum zum Ergebnis beitragende Argumente)
- ...

$\leadsto$ mehr über Muster und musterbasierte Funktionsdefinitionen später...

Abseitsregel (engl. offside rule) (1)

...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 (1)...

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

square x = x * x
```

Abseitsregel (2)

Graphische Veranschaulichung...

```
------------------------------------------------
|
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)

(*Oft*) *folgende Abhilfe bei ineffizienten Implementierungen möglich*:

$\leadsto$ Umformulieren! Ersetzen ungünstiger durch günstigere Rekursionsmuster!

Etwa...

- Rückführung *linearer* Rekursion auf *repetitive* Rekursion

Anm. zu Effektivität & Effizienz... (3)

...am Beispiel der Fakultätsfunktion:

Klassische Formulierung mit *linearem* Rekursionsmuster...

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

Alternative 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)
```

$\leadsto$ "*Trick*" ...Rechnen auf Parameterposition!

Kasken- oder baumartige Rekursion

...komplexitätsmäßig ein ungünstiges Rekursionsmuster

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

Folge $f_0, f_1, \ldots$ der *Fibonacci-Zahlen* 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

$\rightsquigarrow$ ...und ist sehr, seehr, 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
```

...exponentielles Wachstum!

Komplexitätsklassen (1)

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

Erinnerung ...$\mathcal{O}$-Notation

- Sei f eine Funktion $f \ : \ \alpha \to IR^+$ 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)

Beispiele häufig auftretender Kostenfunktionen...

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

Komplexitätsklassen (3)

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

n	linear	quadratisch	kubisch	exponentiell
1	1 μs	1 μs	1 μs	2 μs
10	10 μs	100 μs	1 ms	1 ms
20	20 μs	400 μs	8 ms	1 s
30	30 μs	900 μs	27 ms	18 min
40	40 μs	2 ms	64 ms	13 Tage
50	50 μs	3 ms	125 ms	36 Jahre
60	60 μs	4 ms	216 ms	36 560 Jahre
100	100 μs	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?

Nächster Vorlesungstermin...

...möglicherweise vom üblichen Rhythmus abweichend am

- Di, 23.11.2003, **18-19 Uhr**

Falls der Termin am 23.11.2004 auf die spätere Uhrzeit verlegt wird, werden Ort und Zeit auf Webseite und in newsgroup bekanntgegeben! Achten Sie deshalb bitte in den kommenden Tagen auf Webseite und in newsgroup auf entsprechende Hinweise. Vielen Dank!