

8. Lazy Evaluation

eager

vs.

lazy

allgemeines Prinzip (auch in der SWT):

Erst werden **alle evtl. benötigten Werte** berechnet, dann die Operation darauf ausgeführt.

Strikte Auswertung: Wenn ein Operand **bottom** liefert (nicht terminiert), dann liefert auch der Ausdruck **bottom**.

Eine **Berechnung** wird erst dann ausgeführt, **wenn ihr Ergebnis benötigt** wird.

Zusammengesetzte Ergebnisse werden **nur so tief wie nötig ausgewertet**.

Mehrfach benötigte **Ergebnisse** werden nur einmal berechnet und dann **wiederverwendet**.

Parameterübergabe: call-by-value

call-by-name, call-by-need

Datenstrukturen: Listen

Ströme

Sprachsemantik: SML, Lisp

Haskell, Miranda

Einführung in Notationen von Haskell

Definitionen von Funktionen:

```
add :: Int -> Int -> Int
add x y = x + y
```

vorangestellte Signatur ist guter Stil,
aber nicht obligatorisch

```
inc1 :: Int -> Int
inc1 = add 1
```

```
inc2 :: Int -> Int
inc2 = (+2)
```

entspricht (`secr op+ 2`) in SML

```
sub :: Int -> Int -> Int
sub = \x y -> x - y
```

Lambda-Ausdruck in Haskell

Funktionen über Listen:

```
lg :: [a] -> Int
lg [] = 0
lg (_:xs) = 1 + lg xs
```

```
xmap f [] = []
xmap f (x:xs) = (f x) : (xmap f xs)
```

Aufruf z. B.: `xmap (+2) [1,2,3]`

```
quicksort [] = []
quicksort (x:xs) =
  quicksort [y | y <- xs, y < x] ++
  [x] ++
  quicksort [y | y <- xs, y >= x]
```

Lazy-Semantik in Haskell

Die Semantik von Haskell ist **konsequent lazy**,
nur elementare Rechenoperationen (+, *, ...) werden strikt ausgewertet.

Beispiele:

```
inf = inf
```

ist wohldefiniert; aber die Auswertung würde nicht terminieren.

```
f x y = if x == 0 then True else y
```

Parameterübergabe call-by-need:

```
f 0 inf                                liefert True
```

```
f inf False                            terminiert nicht, liefert bottom
```

Lazy Listen in Haskell

Listen in Haskell haben Lazy-Semantik - wie alle Datentypen.

Definition einer **nicht-endlichen Liste** von 1en:

```
ones :: [Int]
ones = 1 : ones
```

```
take 4 ones          liefert [1, 1, 1, 1]
```

Funktionsaufrufe brauchen nicht zu terminieren:

```
numsFrom :: Int -> [Int]
numsFrom n = n : numsFrom (n+1)
```

```
take 4 (numsFrom 3) liefert [3, 4, 5, 6]
```

Listen als Ströme verwenden

Listen können unmittelbar wie Ströme verwendet werden:

```
squares :: [Int]
squares = map (^2) (numsFrom 0)

take 5 squares                liefert [0, 1, 4, 9, 16]
```

Paradigma Konvergenz (vgl. FP-7.7):

```
within :: Float -> [Float] -> Float
within eps (x1:(x2:xs)) =
    if abs(x1-x2)<eps then x2 else within eps (x2:xs)
```

```
myIterate :: (a->a) -> a -> [a]
myIterate f x = x : myIterate f (f x)
```

```
nextApprox a x = (a / x + x) / 2.0
```

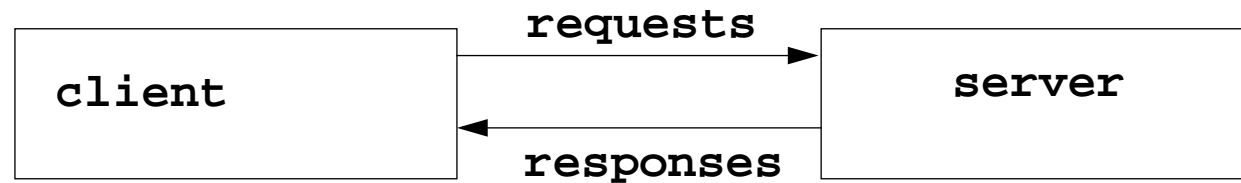
```
groot a = within 1e-8 (myIterate (nextApprox a) 1.0)
```

Strom von Fibonacci-Zahlen:

```
fib :: [Int]                zip erzeugt Strom von Paaren
fib = 1 : 1 : [ a+b | (a,b) <- zip fib (tail fib) ]
```

```
fibs :: [Int]              zipWith verknüpft die Elemente zweier Ströme
fibs = 1 : 1 : (zipWith (+) fibs (tail fibs))
```

Simulation zyklischer Datenflüsse



```

reqs      = client csInit resps
resps     = server reqs
  
```

```

server :: [Int] -> [Int]
server (req:reqs) = process req : server reqs
  
```

```

client :: Int -> [Int] -> [Int]
  
```

```

-- client init (resp:resps) = init : client (next resp) resps
  
```

```

-- Fehler: das zweite Pattern wird zu früh ausgewertet
  
```

```

-- client init resps = init : client (next (head resps)) (tail resps)
  
```

```

-- funktioniert: Das zweite Pattern wird erst bei Benutzung ausgewertet
  
```

```

client init ~(resp:resps) = init : client (next resp) resps
  
```

```

-- Das zweite Pattern wird erst bei Benutzung ausgewertet
  
```

```

csInit      = 0
  
```

```

next resp   = resp
  
```

```

process req  = req+1
  
```

Beispiel: Hamming-Folge

Erzeuge eine Folge $X = x_0, x_1, \dots$ mit folgenden Eigenschaften:

1. $x_{i+1} > x_i$ für alle i
2. $x_0 = 1$
3. Falls x in der Folge X auftritt, dann auch $2x$, $3x$ und $5x$.
4. Nur die durch (1), (2) und (3) spezifizierten Zahlen treten in X auf.

Funktion zum Verschmelzen zweier aufsteigend sortierten Listen zu einer ohne Duplikate:

```
setMerge :: Ord a => [a] -> [a] -> [a]
setMerge allx@(x:xs) ally@(y:ys) -- allx ist Name für das gesamte Pattern
  | x == y      = x : setMerge xs    ys
  | x <  y      = x : setMerge xs    ally
  | otherwise   = y : setMerge allx  ys
```

Funktion für die Hamming-Folge, wie definiert:

```
hamming :: [Int]
hamming = 1 : setMerge (map (*2) hamming)
                  (setMerge (map (*3) hamming)
                           (map (*5) hamming))
```