Wiederholung: nützliche Funktionen

```
dropWhile :: ( a -> Bool ) -> [ a ] -> [ a ]
dropWhile p xs = case xs of
  []      -> []
  x : xss ->  if p x then (dropWhile p xss) else xss

dropWhile ( < 4 ) [ 1 .. 10 ]

takeWhile :: ( a -> Bool ) -> [ a ] -> [ a ]
takeWhile p xs = case xs of
  []      -> []
  x : xss ->  if p x then x:(takeWhile p xss) else []

takeWhile ( \ x -> mod x 5 < 4) [ 1 .. ]
```

WH: `takeWhile, dropWhile` durch `foldr` definieren

```
take :: Int -> [ a ] -> [ a ]
take 0 _ = []
take _ [] = []
take n ( x : xs ) = x : ( take ( n - 1 ) xs )

take 3 [ 1 .. ]
```

Warum funktioniert (terminiert) `take 3 [ 1 .. ]` ?

Wiederholung – Auswertung von Ausdrücken

Reduktion: Termersetzung durch Funktionsanwendung

Redex: reduzierbarer Teilterm

Normalform: nicht-reduzierbarer Ausdruck
(Ausdruck ohne Redex)

Auswertung: schrittweise Reduktion, bis Normalform erreicht

```
square :: Int -> Int
square x = x * x
```

2 Möglichkeiten,
den Wert von `square (3 + 1)` zu berechnen:

- `square (3+1) = (3+1)*(3+1)=4*(3+1)=4*4 = 16`
- `square (3+1) = square 4 = 4*4 = 16`

Bei beiden Möglichkeiten wird derselbe Wert berechnet.
(Haskell ist nebenwirkungsfrei.)

Auswertungsreihenfolge

```
mult :: Int ->  Int -> Int
mult  = \x y -> x * y
```

Redexe von `mult (1 + 2) (2 + 3)` bei Positionen $0, 01$ und 1

```
data Nat = Z | S Nat
nichtnull :: Nat ->  Bool
nichtnull n = case n of
    Z   -> False
    S _ -> True
```

Auswertung von `nichtnull (S undefined)` bei Position

ε: `nichtnull ( S undefined )`
`== case (S undefined) of Z -> False ; S _ -> True`
`== True`

1: Fehler bei Auswertung von `(S undefined)` (Konstruktor)

Auswertungs-Strategien

innermost Reduktion von Redexen, die keinen Redex enthalten (Parameterübergabe by value)

outermost Reduktion von Redexen, die in keinem Redex enthalten sind (Parameterübergabe by name)

(jeweils so weit links im Baum wie möglich zuerst)
Auswertung von `square (3 + 1)` :

- innermost (bei Position 1):
  `square (3+1) = square 4 =  4*4 = 16`
- outermost (bei Position 0):
  `square (3+1) = (3+1)*(3+1)=4*(3+1)=4*4 = 16`

Auswertung von `nichtnull ( S undefined )` :

- innermost (bei Position 11):
  Fehler bei Auswertung von `undefined`
- outermost (bei Position $\emptyset$):
  `nichtnull ( S undefined ) == True`

Teilterme in λ-Ausdrücken werden nicht reduziert: `(\x->1+2) 1`

Termination

```
inf :: Int
inf = 1 + inf
```

```
fst :: ( a , b ) -> a
fst ( x, y ) = x
```

Auswertung von

```
fst (3, inf)
```

terminiert unter outermost-Strategie (`fst`),
aber nicht unter innermost-Strategie (`inf`),

Satz

Für jeden Ausdruck, für den die Auswertung unter irgendeiner Strategie terminiert, terminert auch die Auswertung unter outermost-Strategie.

Unendliche Datenstrukturen

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

Berechnung von `nats_from 3` terminiert nicht

outermost-Auswertung von:

```
head ( tail ( tail ( nats_from 3 ) ) )

= head ( tail ( tail ( 3 : ( nats_from ( 3 + 1 ))))))
= head ( tail ( nats_from (3 + 1)))
= head ( tail ( (3 + 1)  : nats_from (( 3 + 1 ) + 1 ))
= head ( nats_from ( ( 3 + 1 ) + 1 ) )
= head (((3 + 1) + 1) : nats_from (((3 + 1) + 1) + 1))
= ( 3 + 1 ) + 1
= 4 + 1
= 5
```

Lazyness

- jeder Wert wird erst bei Bedarf ausgewertet.
- Listen sind Streams, der Tail wird erst bei Bedarf ausgewertet.
- Wann die Auswertung stattfindet, lässt sich nicht beobachten.

Die Auswertung hat keine Nebenwirkungen.

Strictness

zu jedem Typ T betrachte $T_\perp = \{\perp\} \cup T$

dabei ist $\perp$ ein „Nicht-Resultat vom Typ T“

- Exception `undefined :: T`
- oder Nicht-Termination `let { x = x } in x`

Definition:
Funktion f heißt strikt, wenn $f(\perp) = \perp$.
Funktion f mit n Argumenten heißt strikt in i, falls
$(x_i = \perp) \Rightarrow f(x_1, \ldots, x_n) = \perp$

in Haskell:

- Konstruktoren (`S` , `Cons` , ...) sind nicht strikt,
- Destruktoren (`head` , `tail` , ...) sind strikt.

Strictness – Beispiele

- `length :: [a] -> Int` ist strikt:

  ```
  length undefined   ==>  exception
  ```

- `(:) :: a->[a]->[a]` ist nicht strikt im 1. Argument:

  ```
  length (undefined : [2,3]) ==> 3
  ```

  d.h. (`undefined : [2,3]`) ist nicht $\perp$

- `(&&)` ist strikt im 1. Argument,
  nicht strikt im 2. Argument

  ```
  undefined && True   ==> (exception)
  False && undefined  ==> False
  ```

Lazy Evaluation – Realisierung

Begriffe:

nicht strikt : nicht zu früh auswerten

lazy : höchstens einmal auswerten

bei jedem Konstruktor- und Funktionsaufruf:

- kehrt sofort zurück
- Resultat ist thunk
- thunk wird erst bei Bedarf ausgewertet
- Bedarf entsteht durch Pattern Matching
- nach Auswertung: thunk durch Resultat überschreiben

Lazy Evaluation (Bedarfsauswertung) =
Outermost-Reduktionsstrategie mit Sharing

Unendliche Datenstrukturen

```
einsen :: [Int]
einsen = 1 : einsen


head einsen
take 3 einsen


walzer :: [Int]
walzer = 1 : 2 : 3 : walzer


nats :: [Int]
nats = 0 : map (+1) nats

takeWhile (<= 5) nats
```

Liste aller Quadratzahlen? Primzahlen?

Motivation: Datenströme

Folge von Daten:

- erzeugen (producer)
- transformieren
- verarbeiten (consumer)

aus softwaretechnischen Gründen:
diese drei Aspekte im Programmtext trennen,

aus Effizienzgründen:
in der Ausführung verschränken

(bedarfsgesteuerte Transformation/Erzeugung)

Rekursive Stream-Definitionen

```
nats = 0 : map (+1) nats

fibonacci = 0
    : 1
    : zipWith (+) fibonacci ( tail fibonacci )

take 10 fibonacci
take 1 $ dropWhile (< 200) fibonacci
```

Welchen Wert hat `bin` ?

```
bin = False
    : True
    : concat ( map ( \ x -> [ x, not x ] )
                   ( tail bin ) )
```

Thue-Morse-Folge $t = 0110100110010110\ldots$
mit vielen interessanten Eigenschaften , z.B.

- $t := \lim_{n\to\infty} \tau^n(0)$ für $\tau : 0 \mapsto 01, 1 \mapsto 10$
- t ist kubikfrei
- Abstandsfolge $v := 210201210120\ldots$
  ist auch Fixpunkt eines Morphismus, quadratfrei

Primzahlen

Sieb des Eratosthenes

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

primzahlen :: [ Int ]
primzahlen = sieb $ nats_from 2

sieb :: [ Int ] -> [ Int ]
sieb (x : xs) =
   x
   : sieb (filter ( \ y -> mod y x /= 0) xs )

take 100 primzahlen
takeWhile (< 100) primzahlen
```