

Strom := Folge oder Liste von Daten,
unbegrenzt viele Daten-Elemente.

Ein Programm bzw. Funktion kann
Eingabeströme und Ausgabeströme haben

Ströme sind in Haskell als
als (potentiell) unendliche Listen darstellbar
und programmierbar

Modellvorstellung :

- Eingabe-Strom ist nur einmal einlesbar
- nur ein kleiner Teil des Eingabestroms kann gespeichert werden
- Die Ausgabe darf nicht auf das Ende des Stroms warten

Kanäle,

Folge von Zeichen

Lesen eines sequentiellen Files

Liste von Zeichen

*lexikalische Analysephase
eines Compilers*

Folge von Zeichen → Folge von Token

*syntaktische Analysephase
eines Compilers*

Folge von Token → ...

Geeignet sind z.B.: map, filter, nub, zip, scanl
 concat, foldr
 word, unword, lines, unlines

Geeignet für mehrere Ströme: zip, zipWith, merge

Eingeschränkt geeignet: scanr

Nicht geeignet: length, reverse, foldl, sum.
 Sortierfunktionen

mittels verzögerter Auswertung unendlicher Listen
Ausgabe unendlicher Listen

Beispiel [1..] Strom aller natürlichen Zahlen

In Haskell:

Wegen Endrekursions-Optimierung und Garbage-Collector:
Verarbeitung von Strömen in konstantem Speicher möglich

Voraussetzung: keine Referenz auf den Anfang des Stroms
Ausgaben auch ohne das Ende
des Eingabestroms abzuwarten

Beispiel: Verarbeitung unendlicher Listen

```
Prelude> [1..]
```

```
[1,2,3,4,5,6,7,8,9,10,11,12,13,14,15,16,17,18,19,20,21,22,23,24, ...]
```

```
Prelude> map quadrat [1..]
```

```
[1,4,9,16,25,36,49,64,81,100,121,144,169,196,225,256,289, ...]
```

Frage ob mehr als n Elemente im Strom:

```
drop n xs /= []
```

```
length_ge str n = drop n xs /= []
```

```
*Main> length_ge [1..] 2
```

```
True
```

```
*Main> length_ge [1..10] 20
```

```
False
```

Rekursive Funktionen auf Strömen:

Der Nachweis der Terminierung (welche?) und
der Nachweis der Korrektheit:

ist **nicht möglich mittels vollständiger Induktion**,
da es sich um unendliche Listen handelt.

Näherung: die Listen sind sehr groß.

siehe Methodik im Semantik-Kapitel (zweite Semesterhälfte)

Co-Induktion statt Induktion

```
*Main> filter (\x -> x > 100 && x < 1000) [1..]
```

```
[101,102,103,104,105,106,107,108,109, .....  
992,993,994,995,996,997,998,999^CInterrupted.
```

takeWhile oder dropWhile statt filter
zum Extrahieren endlicher Listen aus Strömen.

```
*Main> takeWhile (\x -> x < 1000) [1..]  
[1,2,3,4,5,6,7, ... 96,997,998,999]
```

„Zufällige“ Ströme bzw. Listen (alter Haskell-Standard) erhält man
mit

```
randomInts 1 2  
[2147442192,436925867,1434534200,990707786,1573909306, ...
```

Jedes k -te Element eines Stroms auswählen:

```
strom_ktes xs k = let (y:ys) = drop k xs      in y : ( strom_ktes ys k)
```

```
strom_ktes [1..] 10
```

```
[11,22,33,44,55,66,77,88,99,110,121,132,143 ..
```

Die Funktion `zipWith` wendet eine Funktion f
auf die jeweiligen ersten Elemente von zwei Strömen an:

```
zipWith (+) [1..] [2,4..]
```

```
[3,6,9,12,15,18,21,24,27,30,33,36,39,...
```

nicht synchrone Verarbeitung:

```
zipWith (\ x y -> take x (repeat y)) [1..] ['a'..]  
["a", "bb", "ccc", "dddd", "eeee", "ffffff", "ggggggg", "hhhhhhh",
```

Vermehrung der Elemente einer Liste ergibt nicht-synchrone Verarbeitung:

```
str_kopiere str1 :: [Int] -> [b] -> [b]  
str_kopiere str1 str2 =  
    concat (zipWith (\ x y -> take x (repeat y)) str1 str2)
```

```
str_kopiere (map (\x-> x 'rem'3) (randomInts 1 2)) ['a'..]  
"bbccddefffgghiijjkppqqstuu .....
```

Ausgabe kann nicht die lexikographische Ordnung einhalten,
wenn die Ausgabe fair sein soll.

Kreuzprodukt in lexikographischer Reihenfolge,
wenn die Eingabe sortierte endliche Mengen sind:

`kreuzprodukt xs ys = [(x,y) | x <- xs, y <- ys]`

Kreuzprodukt von zwei Strömen mit fairer Abzählung:

```
-- tails (x:xs) = (x:xs) : tails xs
strKreuz xs (y:ys) = concat (stKreuzr (tails xs) [y:ys] ys)
strKreuzr xstails ystailsr [] =
    [(zip (map head xstails) (map head ystailsr))]
strKreuzr xstails ystailsr (y:ysend) =
    (zip (map head xstails) (map head ystailsr))
    : (strKreuzr xstails ((y:ysend) : ystailsr) ysend)
```

```
stKreuz [1..] [1..]
[(1,1), (1,2), (2,1), (1,3), (2,2), (3,1), (1,4), (2,3), (3,2), (4,1), (1,5), (2,4),
(3,3), (4,2), (5,1), (1,6), (2,5), (3,4), (4,3), (5,2), (6,1),
(1,7), (2,6), (3,5), (4,4), (5,3), (6,2), (7,1),    ....]
```

Beispiel: scanl, scanr

scanl berechnet das foldl jeweils der ersten n Elemente:

```
scanl (+) 0 [1..]
```

```
[0,1,3,6,10,15,21,28,36,45,55,66,78,91, ...
```

```
scanl (*) 1 [1..]
```

```
[1,1,2,6,24,120,720,5040,40320,362880,3628800,39916800,479001600,6227020800,87178291200,1307674368000,.....
```

```
scanl (\x y-> y:x) [] (repeat 'a')
```

```
["", "a", "aa", "aaa", "aaaa", "aaaaa", "aaaaaa", "aaaaaaa", "aaaaaaaa",  
"aaaaaaaaa", "aaaaaaaaaaa", "aaaaaaaaaaaa", "aaaaaaaaaaaaa", "aaaaaaaaaaaaaa",
```

scanr nur in Spezialfällen geeignet

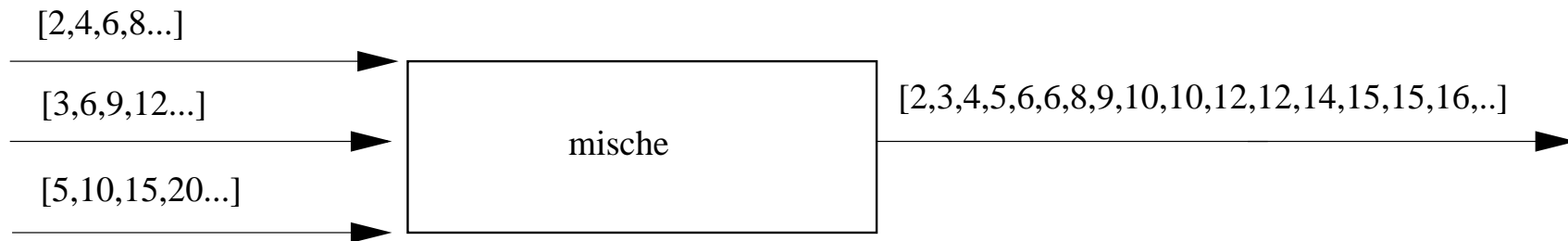
Mischen von 2 aufsteigend sortierten Strömen:

```
mische [] ys = ys
mische xs [] = xs
mische (x:xs) (y:ys) =
    if x <= y then x: (mische xs (y:ys))
    else y: (mische (x:xs) ys)
```

```
mische [1,3..] [2,4..]
[1,2,3,4,5,6,7,8,9,10,11,12,13,14,15,16,17,18,19,20,21,22,23,
24,25,26,27,28,29,30,31,32,33,34,35,36,37,
38,39,40,41,42,43,44,45,46,47,48,49,50,51,52,
53,54,55,56,57,58,59,60,61,62,63,64,65,66,67,68,....
```

```
> mische [2,4..] [3,6..]
[2,3,4,6,6,8,9,10,12,12
```

Mischen der Vielfachen von 2,3,5:



```
Main> mische (mische (map (2*) [1..]) (map (3*) [1..])) (map (5*) [1..])
[2,3,4,5,6,6,8,9,10,10,12,12,14,15,15,16,18,18,20,20,21,22,24,24,25,26,27
28,30,30,30,32,33,34,35,36,36,38,39,40,40,42,42,44,45,45,46,48,48,50,50,..
```

Ausgabe enthält doppelte Elemente.

Entfernen: mit `nub`

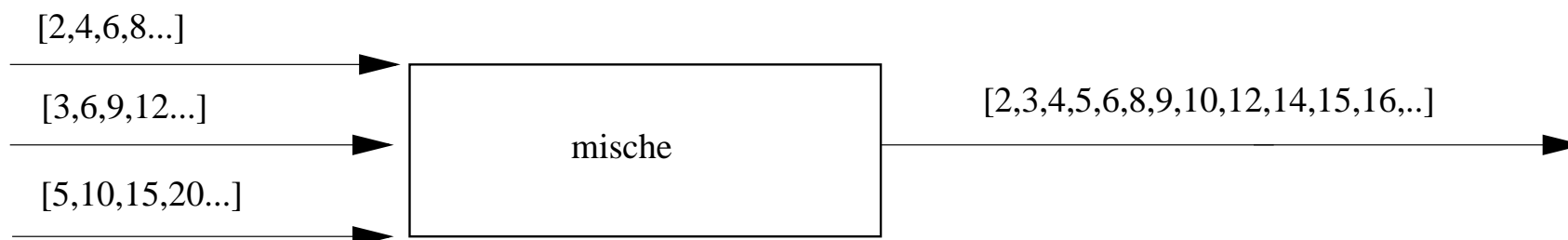
oder `mische` umprogrammieren

Entfernen doppelter Zahlen:

```
*Main> nub (mische (mische (map (2*) [1..]) (map (3*) [1..]))  
            (map (5*) [1..]) )
```

```
[2,3,4,5,6,8,9,10,12,14,15,16,18,20,21,22,24,25,26,27,28,30,32,33,34,...
```

Sortierte Ströme: Mischen ohne Doppelte



```
mischeNub [] ys = ys
mischeNub xs [] = xs
mischeNub (x:xs) (y:ys) =
  if x == y then (mischeNub xs (y:ys))
  else
    if x <= y then x: (mischeNub xs (y:ys))
    else y: (mischeNub (x:xs) ys)
```

```
*Main> mischeNub (mischeNub (map (2*) [1..]) (map (3*) [1..]))
  (map (5*) [1..])
[2,3,4,5,6,8,9,10,12,14,15,16,18,20,21,22,24,25,26,27,28,30,32,33,34,35,
```

Differenz von zwei Strömen wenn
die Eingabeströme aufsteigend sortierte Zahlen enthalten

```
strom_minus xs [] = xs
strom_minus [] ys = []
strom_minus (x:xs) (y:ys) = if x == y then strom_minus xs (y:ys)
                             else if x > y then strom_minus (x:xs) ys
                             else x: (strom_minus xs (y:ys))
```

```
> strom_minus [1..]
      (nub (mische (map (2*) [1..])
                 (mische (map (3*) [1..]) (map (5*) [1..]) )))
[1,7,11,13,17,19,23,29,31,37,41,43,
```

Schnitt zweier Strömen aus aufsteigend sortierte Zahlen:

```
strom_schnitt xs [] = []
strom_schnitt [] ys = []
strom_schnitt (x:xs) (y:ys) = if x == y then x: strom_schnitt xs ys
                               else if x > y then strom_schnitt (x:xs) ys
                               else strom_schnitt xs (y:ys)
```

```
*Main> strom_schnitt [2,4..] [3,6..]
[6,12,18,24,30,36,42,48,54,60,66,72,78,84,...]
```

Verallgemeinerung der Funktionen für variable Ordnung

```
mischeNubGen ord [] ys = ys
mischeNubGen ord xs [] = xs
mischeNubGen ord (x:xs) (y:ys) =
    if x 'ord' y && y 'ord' x then (mischeNubGen ord xs (y:ys))
    else
    if x 'ord' y then x: (mischeNubGen ord xs (y:ys))
    else y: (mischeNubGen ord (x:xs) ys)
```

Für absteigende Ströme: `mischeNubGen (>=)`

```
primes = 2: [x | x <- [3,5..],  
            and (map (\t-> x `mod` t /= 0)  
                (takeWhile (\y -> y^2 <= x) primes))]
```

Der Vergleich mit einer durch den Fermatschen Primzahltest erzeugten Liste ergibt:

```
primesDifference = strom_minus (2:[x | x <- [3,5..], fermat_test_naiv x]  
                               primes
```

```
*Main> primesDifference  
[561,1105,1729,2465,2821,6601^C
```

foldr geeignet;

foldl **ungeeignet**: Nichtterminierung bei unendlichen Strömen

```
*Main> foldl (\y x -> x:x:y) [] [1..10]
[10,10,9,9,8,8,7,7,6,6,5,5,4,4,3,3,2,2,1,1]
```

```
*Main> foldl (\y x -> x:x:y) [] [1..]
^CInterrupted.
```

```
*Main> foldr (\x y -> x:x:y) [] [1..10]
[1,1,2,2,3,3,4,4,5,5,6,6,7,7,8,8,9,9,10,10]
```

```
foldr (\x y -> x:x:y) [] [1..]
[1,1,2,2,3,3,4,4,5,5,6,6.....
```

```
*Main>
```

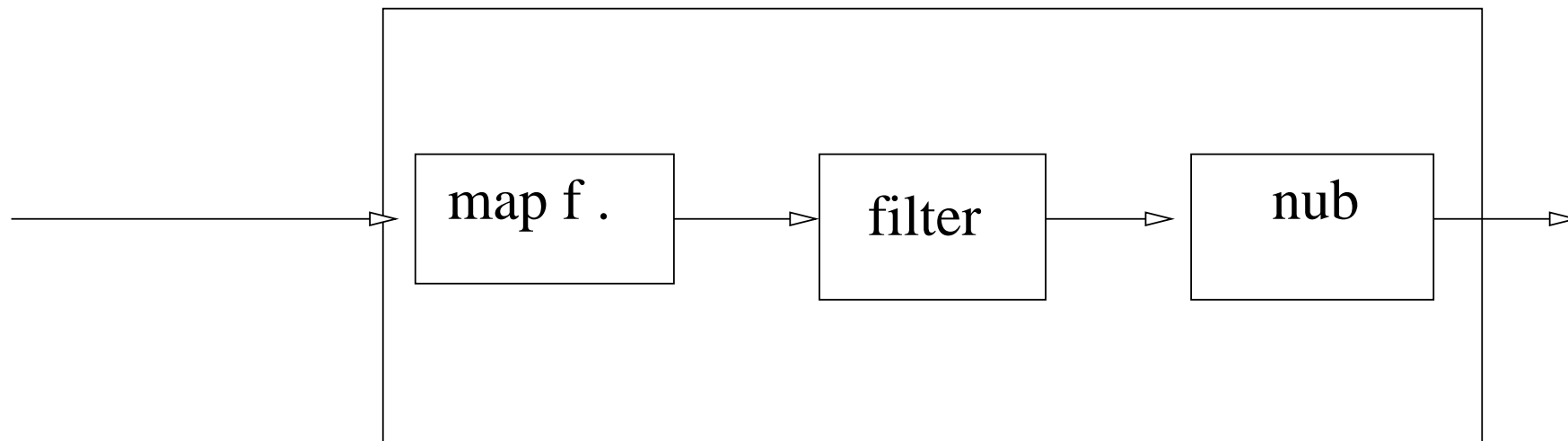
Funktionen zur Bearbeitung langer Strings:

Vordefinierte Funktionen in Haskell:

```
words :: String -> [String]
unwords :: [String] -> String
lines :: String -> [String]
unlines :: [String] -> String
```

verzögerte Auswertung erlaubt:
Programmierung kombinierter Anfragen,
die den Strom **nur einmal lesen**.

```
nub (filter p (map f xs))
```



Falle: Speicher-Blockaden

z.B. durch Definition eines Stromes als Konstante im Programm

Z.B. `primes`,

```
fileLesen fu = do
  putStr "File-Name:?"
  fname <-getLine
  contents <-(readFile fname)
  putStr (fu contents)
```

Mengen als aufsteigend sortierte Listen:

Schnitt, Vereinigung, und Differenz sind

einfach und sehr effizient

Ergebnis ebenfalls aufsteigend sortierte Liste