Unendliche Listen: Programmieren mit Strömen

Die folgende Darstellung orientiert sich an...

- Kapitel 14
  Paul Hudak. *The Haskell School of Expression – Learning Functional Programming through Multimedia*, Cambridge University Press, 2000.

- Kapitel 17
  Simon Thompson. *Haskell – The Craft of Functional Programming*, Addison-Wesley, 2nd edition, 1999.

Ströme

Jargon

Strom ...synonym zu *unendliche Liste*
synonym zu *lazy list*

Ströme

- ...erlauben (zusammen mit lazy evaluation), viele Probleme elegant, knapp und effizient zu lösen

- ...bieten bei ungeeigneter Anwendung allerdings auch Klippen und Fallstricke

Mehr dazu im folgenden...

Ströme

Vereinbarung

Anstatt einen speziellen polymorphen Datentyp `Stream` einzuführen...

```
data Stream a = a :* Stream a
```

...werden wir Ströme durch herkömmliche Listen unter Verzicht auf die Benutzung der leeren Liste `[ ]` simulieren.

Motivation dafür:

- Bequemlichkeit/Angemessenheit ...viele vordefinierte (polymorphe) Funktionen auf Listen, die sonst auf dem neuen Datentyp `Stream` neu zu definieren wären, können so wiederverwendet werden

Erste Beispiele für Ströme

- *In Haskell eingebaute Ströme*

```
[3 ..]   = [3,4,5,6,7,...
[3,5 ..] = [3,5,7,9,11,...
```

- *Selbstdefinierte rekursive Listen* (*Ströme*)
  Die unendliche Liste von “Zweien”

  ```
  2,2,2,...
  ```

  In Haskell darstellbar...
  - ...mittels Listenkomprehension: `[2..]`
  - ...als rekursiver Strom: `twos = 2 : twos`

    Veranschaulichung

    ```
    twos => 2 : twos
         => 2 : 2 : twos
         => 2 : 2 : 2 : twos
         => ...
    ```

    ...`twos` repräsentiert eine unendliche Liste, oder kürzer, einen *Strom*

Funktionen auf Strömen

```
head :: [a] -> a
head (x:_) = x
```

Anwendung

```
head twos
  => head (2 : twos)
  => 2
```

Beachte: Normal-order Reduktion (bzw. ihre effiziente Implementierungsvariante *lazy evaluation*) garantiert (in diesem Beispiel) Terminierung.
Die nicht endliche Reduktionsfolge...

```
head twos
  => head (2 : twos)
  => head (2 : 2 : twos)
  => head (2 : 2 : 2 : twos)
  => ...
```

...wird dadurch ausgeschlossen.

Fkt. auf Strömen: Weitere Beispiele

```
addFirstTwo :: [Integer] -> Integer
addFirstTwo (x:y:zs) = x+y
```

Anwendung

```
addFirstTwo twos => addFirstTwo (2:twos)
                 => addFirstTwo (2:2:twos)
                 => 2+2
                 => 4
```


Erinnerung

...wann immer eine terminierende Reduktionsfolge eines Ausdrucks existiert, so terminiert auch die normal-order Reduktionsfolge (Church/Rosser-Theorem)

- Auswertung in *normal-order* Reihenfolge entspricht Auswertung in *leftmost-outermost* Reihenfolge

  Beachte: Im Falle von...

  ```
  ignore :: a -> b -> b
  ignore a b = b
  ```

  ist in beiden Fällen

  – `ignore twos 42`

  – `twos 'ignore' 42`

  der linkeste äußerste Operator durch den Aufruf `ignore` gegeben.

Weitere Beispiele für Ströme

- Selbstdefinierte rekursive Listen/Ströme

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

  fromStep :: Int -> Int -> [Int]
  fromStep n m = n : fromStep (n+m) m
  ```

  Anwendung

  ```
  from 42 => [42, 43, 44,...

  fromStep 3 2 => 3 : fromStep 5 2
               => 3 : 5 : fromStep 7 2
               => 3 : 5 : 7 : fromStep 9 2
               => ...
  ```

Weitere Beispiele

- Die Folge der Potenzen einer ganzen Zahl...

```
powers :: Int -> [Int]
powers n = [n^x | x <- [0 ..]]
```

- Genereller: Die Iterationsfunktion...

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

  Die Iterationsfunktion liefert den Strom

```
[x, f x, (f . f) x, (f . f . f) x, ..
```


Primzahlen: Das Sieb des Eratosthenes 1(4)

Intuition

1. Schreibe die natürlichen Zahlen beginnend mit 2 auf.
2. Die kleinste noch nicht gestrichene Zahl ist eine Primzahl. Streiche alle Vielfachen dieser Zahl
3. Wiederhole Schritt 2 mit der nächstkleinsten nicht gestrichenen Zahl.

Veranschaulichung

```
Schritt 1:   2  3  4  5  6  7  8  9  10  11  12  13...
Schritt 2:   2  3     5     7     9      11      13...
 ("mit 2")
Schritt 2:   2  3     5     7            11      13...
 ("mit 3")
...
```

Primzahlen: Sieb d. Eratosthenes 2(4)

Die Folge der Primzahlen...

```
primes :: [Int]
primes = sieve [2 ..]

sieve :: [Int] -> [Int]
sieve (x:xs) = x : sieve [ y | y <- xs, mod y x > 0 ]
```


Primzahlen: Sieb d. Eratosthenes 3(4)

Veranschaulichung ...durch manuelles Auswerten

```
primes
  => sieve [2 ..]
  => 2 : sieve [ y | y <- [3 ..], mod y 2 > 0 ]
  => 2 : sieve (3 : [ y | y <- [4 ..], mod y 2 > 0 ]
  => 2 : 3 : sieve [ z | z <- [ y | y <- [4 ..], mod y 2 > 0 ],
                                      mod z 3 > 0]
  => ...
  => 2 : 3 : sieve [ z | z <- [5, 7, 9 ..], mod z 3 > 0 ]
  => ...
  => 2 : 3 : sieve [5, 7, 11,...]
  => ...
```

Primzahlen: Sieb d. Eratosthenes 4(4)

- *Anwendung*

```
member primes 7    ...liefert "True"
```

aber

```
member primes 6    ...terminiert nicht!
```

wobei

```
member :: [a] -> a -> Bool
member []     y = False
member (x:xs) y = (x==y) || member xs y
```

- *Frage*(n): Warum? Lässt sich `primes` so in einen Kontext einbetten, dass es festzustellen erlaubt, ob ein bestimmtes Argument prim ist oder nicht?

Zufallszahlen 1(2)

Erzeugung einer Folge von (Pseudo-) Zufallszahlen...

```
nextRandNum :: Int -> Int
nextRandNum n = (multiplier*n + increment) 'mod' modulus

randomSequence :: Int -> [Int]
randomSequence = iterate nextRandNum
```

Für

```
seed       = 17489          increment  = 13849
multiplier = 25173          modulus    = 65536
```

erhält man so folgende Sequenz von (Pseudo-) Zufallszahlen

```
[17489, 59134, 9327, 52468, 43805, 8378,...
```

zwischen `0` und `65536`, wobei alle Zahlen aus diesem Intervall mit gleicher Häufigkeit auftreten.

Zufallszahlen 2(2)

Oft möchte man Zufallszahlen in einem Intervall von Zahlen `p` und `q`, `p<q`, haben.

Dies kann durch entsprechende Skalierung erreicht werden.

```
scale :: Float -> Float -> [Int] -> [Float]
scale p q randSeq = map (f p q) randSeq
      where f :: Float -> Float -> Int -> Float
            f p q n = p + ((n * (q-p)) / (modulus-1))
```

Anwendung

```
scale 42.0 51.0 randomSequence
```


Modularisierungsprinzipien

...im Zusammenhang mit Strömen

- *Generator/Selektor*-Prinzip
  ...z.B. Berechnung der Quadratwurzel, Fibonacci-Zahlen

- *Generator/Transformator*-Prinzip
  ...z.B. “skalierte” Zufallszahlen

Mehr über rekursive Ströme

Erinnerung ...die Folge der Fibonacci-Zahlen

```
1, 1, 2, 3, 5, 8, 13, 21, 34, 55, 89,...
```

ist definiert durch

$fib : \mathbb{N} \to \mathbb{N}$

$$fib(n) =_{df} \begin{cases} 1 & \text{falls } n = 0 \ \vee \ n = 1 \\ fib(n-1) + fib(n-2) & \text{sonst} \end{cases}$$

Die Fibonacci-Zahlen 1(4)

Eine naive Implementierung...

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

...haben wir bereits im letzten Semester als inakzeptabel ineffizient kennen gelernt.

Die Fibonacci-Zahlen 2(4)

Veranschaulichung ...durch manuelle Auswertung

```
fib 0   =>  1  --  1 Aufrufe von fib

fib 1   =>  1  --  1 Aufrufe von fib

fib 2   =>  fib 1 + fib 0
        =>  1 + 1
        =>  2  --  3 Aufrufe von fib

fib 3   =>  fib 2 + fib 1
        =>  (fib 1 + fib 0) + 1
        =>  (1 + 1) + 1
        =>  3  --  5 Aufrufe von fib
```


Die Fibonacci-Zahlen 3(4)

```
fib 4   =>  fib 3 + fib 2
        =>  (fib 2 + fib 1) + (fib 1 + fib 0)
        =>  ((fib 1 + fib 0) + 1) + (1 + 1)
        =>  ((1 + 1) + 1) + (1 + 1)
        =>  5  -- 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 + 1)) + ((1 + 1) + 1)
        =>  (((1 + 1) + 1) + (1 + 1)) + ((1 + 1) + 1)
        =>  8  --  15 Aufrufe von fib
```

Die Fibonacci-Zahlen 4(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
```

...baumartige Rekursion (exponentielles Wachstum!)

Erinnerung: Komplexität 1(3)

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

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

- Sei f eine Funktion $f \;:\; \alpha \to I\!R^+$ 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:

$$\mathcal{O}(f) =_{df} \{h \mid h(n) \leq c * f(n) \text{ für eine positive Konstante } c \text{ und alle } n \geq N_0\}$$

Erinnerung: Komplexität 2(3)

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

Erinnerung: Komplexität 3(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...

Abhilfe: Rekursive Ströme 1(4)

Idee

```
 1  1  2  3  5  8   13  21...    Folge der Fibonacci-Zahlen
 1  2  3  5  8  13  21  34...    Rest der Folge der F.-Zahlen
-----------------------------------------------------------
 2  3  5  8  13 21  34  55...    Rest d. Restes d. F. d. F.-Zahlen
```

Effiziente Realisierung als rekursiver Strom

```
fibs :: [Integer]
fibs = 1 : 1 : zipWith (+) fibs (tail fibs)
```

wobei

```
zipWith :: (a -> b -> c) -> [a] -> [b] -> [c]
zipWith f (x:xs) (y:ys) = f x y : zipWith f xs ys
zipWith f _      _      = []
```

...erinnert an Münchhausens "sich am eigenen Schopfe aus dem Sumpfe ziehen"

Abhilfe: Rekursive Ströme 2(4)

```
fibs => 1 : 1 : 2 : 3 : 5 : 8 : 13 : 21 : 34 : 55 : 89 : ...

take 10 fibs => [1,1,2,3,5,8,13,21,34,55]
```

wobei

```
take :: Integer -> [a] -> [a]
take 0 _            = []
take _ []           = []
take n (x:xs) | n>0 = x : take (n-1) xs
take _ _            = error "PreludeList.take: negative argument"
```


Abhilfe: Rekursive Ströme 3(4)

Insgesamt

```
fib :: Integer -> Integer
fib n = last take n fibs
```

oder noch kürzer

```
fib n = fibs!!n
```

Beachte:

- Auch hier wieder...
  Anwendung des *Generator/Selektor*-Prinzips

Abhilfe: Rekursive Ströme 4(4)

Veranschaulichung ...durch manuelle Auswertung (mit `add` anstelle von `zipWith (+)`)

```
fibs => Ersetzen des Aufrufs von fibs durch den Rumpf von fibs
           1 : 1 : add fibs (tail fibs)
     => // Ersetzen beider Aufrufe von fibs durch den Rumpf von fib
           1 : 1 : add (1 : 1 : add fibs (tail fibs))
                       (tail (1 : 1 : add fibs (tail fibs)))
     => // Anwenden von tail
           1 : 1 : add (1 : 1 : add fibs (tail fibs))
                       (1 : add fibs (tail fibs))
     => ...
```

- *Beobachtung*
  ...der Berechnungsaufwand bleibt exponentiell!
- *Clou*
  ...lazy evaluation: ...gemeinsame Teilausdrücke werden nicht mehrfach berechnet!
  (Im Beispiel gilt dies für `tail` und `fibs`)

Veranschaulichung 1(3)

```
fibs => 1 : 1 : add fibs (tail fibs)

     => // Einfuehren von Abkuerzungen
        1 : tf
        where tf = 1 : add fibs (tail fibs)
     => 1 : tf
        where tf = 1 : add fibs tf

     => // Einfuehren von Abkuerzungen
        1 : tf
        where tf = 1 : tf2
                   where tf2 = add fibs tf

     => // Auffalten von add
        1 : tf
        where tf = 1 : tf2
                   where tf2 = 2 : add tf tf2
```

Veranschaulichung 2(3)

```
=> // Wiederholung dieser Schritte
   1 : tf
   where tf = 1 : tf2
              where tf2 = 2 : tf3
                          where tf3 = add tf tf2
=> 1 : tf
   where tf = 1 : tf2
              where tf2 = 2 : tf3
                          where tf3 = 3 : add tf2 tf3

=> // tf wird nur an einer Stelle gebraucht und kann
   // deshalb eliminiert werden
   1 : 1 : tf2
   where tf2 = 2 : tf3
               where tf3 = 3 : add tf2 tf3
```

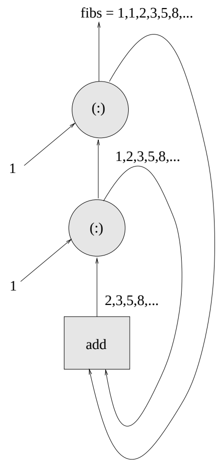

Veranschaulichung 3(3)

```
=> // Schliesslich
   1 : 1 : tf2
   where tf2 = 2 : tf3
               where tf3 = 3 : tf4
                           where tf4 = add tf2 tf3

=> 1 : 1 : tf2
   where tf2 = 2 : tf3
               where tf3 = 3 : tf4
                           where tf4 = 5 : add tf3 tf4

=> 1 : 1 : 2 : tf3
               where tf3 = 3 : tf4
                           where tf4 = 5 : add tf3 tf4
```


Alternativ: Stromdiagramme

Probleme auf Strömen lassen sich oft als (Systeme miteinander verbundener) Prozesse auffassen und visualisieren.

Am Beispiel der Fibonacci-Zahlen...



Ein weiteres Beispiel: Eine Client/Server-Anwendung

Interaktion eines Servers und eines Klienten

```
client :: [Response] -> [Request]
server :: [Request] -> [Response]

reqs  =  client resps
resps =  server reqs
```

Konkretisierung

```
type Request  = Integer
type Response = Integer

client ys = 1 : ys
server xs = map (+1) xs
```


Client/Server-Anwendung (Forts. 1(2))

Beispiel

```
reqs => client resps
     => 1 : resps
     => 1 : server reqs

     => // Einfuehren von Abkuerzungen
        1 : tr
        where tr = server reqs
     => 1 : tr
        where tr = 2 : server tr
     => 1 : tr
        where tr = 2 : tr2
                   where tr2 = server tr
```

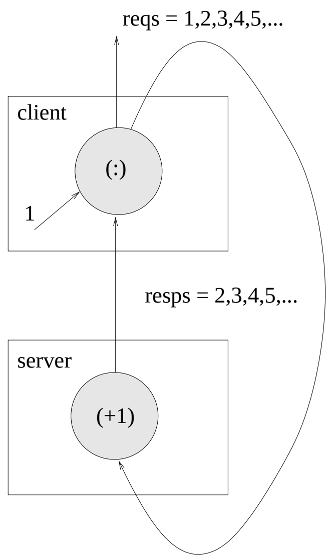

Client/Server-Anwendung (Forts. 2(2))

```
=> 1 : tr
   where tr = 2 : tr2
              where tr2 = 3 : server tr2
=> 1 : 2 : tr2
   where tr2 = 3 : server tr2
=> ...
```

Insbesondere

```
take 10 reqs => [1,2,3,4,5,6,7,8,9,10]
```


Das Client/Server-Bsp. als Stromdiagramm



Klippen überwinden... Lazy Patterns

```
client (y:ys) = if ok y then 1 : (y:ys)
                        else error "Fehlerhafter Server"

where
ok y = True
```

Auswertung...

```
reqs => client resps
     => client (server reqs)
     => client (server (client resps))
     => client (server (client (server reqs)))
     => ...
```

...terminiert nicht!

Lazy Patterns 1(3)

Ad-hoc Abhilfe

```
client ys = 1 : if ok (head ys) then ys
                else error "Fehlerhafter Server"
```

- Ersetzen des pattern matching durch explizite Benutzung der Selektorfunktion `head`
- Verschieben der Fallunterscheidung "in die Liste"

Lazy Patterns 2(3)

Systematische Abhilfe ...lazy patterns

- Syntax: ...voranstellen der Tilde (∼)
- Wirkung: ...wie Benutzung einer expliziten Selektorfunktion

```
client ~(y:ys) = 1 : if ok y then y:ys
                     else error "Fehlerhafter Server"
```

Beachte ...auch bei Benutzung eines lazy patterns muss die Fallunterscheidung noch "verschoben" werden

Lazy Patterns 3(3)

Veranschaulichung ...durch manuelle Auswertung

```
reqs => client resps
     => 1 : if ok y then y : ys
            else error "Fehlerhafter Server"
        where y:ys = resps
     => 1 : (y:ys)
        where y:ys = resps
     => 1 : resps
```

Klippen überwinden... Memo-Tabellen

Achtung ...Teilen/Erkennen gemeinsamer Strukturen hat Grenzen

Folgende Variante der Fibonacci-Funktion...

```
fibsFn :: () -> [Integer]
fibsFn x = 1 : 1 : zipWith (+) (fibsFn ()) (tail (fibsFn ()))
```

...zeigt wieder exponentielles Laufzeit- und Speicherplatzverhalten!

Stichwort:

- *Space* (*Memory*) *Leak* ...der Speicherplatz wird so schnell verbraucht, dass die Programmperformanz spürbar beeinträchtigt wird

Veranschaulichung

```
fibsFn ()
   =>  1 : 1 : add (fibsFn ()) (tail (fibsFn ()))
   =>  1 : tf
       where tf = 1 : add (fibsFn ()) (tail (fibsFn ()))
```

Folgende Vereinfachung wird nicht erkannt

```
   =>  1 : tf
       where tf = 1 : add (fibsFn ()) tf
```

Im Spezialfall wie hier zwar möglich, jedoch nicht generell!

Memo-Funktionen 1(4)

Memo-Funktionen (engl. *Memoization*)

- ...*Konzept* geht zurück auf D. Michie. "*"Memo" Functions and Machine Learning*", Nature, 218, 19-22, 1968.

- ...*Idee*: Ersetze, wo immer möglich, die Berechung einer Funktion gemäß der dafür vorgegebenen Vorschrift (d.h. Funktionsrumpf) durch das Nachschlagen des Funktionswertes in einer Tabelle

Memo-Funktionen 2(4)

- ...*Somit*: Eine Memo-Funktion ist wie eine übliche Funktion, aber "merkt" sich einige oder alle Argumente, auf die sie angewendet wurde und die zugehörigen Funktionsresultate $\leadsto$ Memo-Tabellen.

- ...*Hilfsmittel*: *Memo-Tabellen* – erlauben Wiederberechnung von Werten durch Nachschlagen zu ersetzen
  Korrektheit: Referentielle Transparenz fkt. Programmiersprachen

Memo-Funktionen 3(4)

Berechnung der Fibonacci-Zahlen mithilfe einer Memo-Funktion...

Vorbereitung:

```
flist = [ f x | x <- [0 ..] ]
```

...wobei f Funktion auf ganzen Zahlen. *Anwendung*: Jeder Aufruf von f wird durch Nachschlagen in flist ersetzt.

Am Beispiel der Fibonacci-Zahlen:

```
flist = [ fib x | x <- [0 ..] ]
fib 0 = 1
fib 1 = 1
fib n = flist !! (n-1)  +  flist !! (n-2)
```

anstelle von...

```
fib 0 = 1
fib 1 = 1
fib n = fib (n-1)  +  fib (n-2)
```


Memo-Funktionen 4(4)

Fazit...

- ...Memo-Funktionen: Dazu gedacht, aufwändig zu berechnende Funktionen durch Nachschlag in einer Tabelle zu ersetzen
- ...Beispiel (2^0, 2^1, 2^2, 2^3, ...):

```
power 0 = 1
power i = power (i-1) + power (i-1)
```

Durch Nachschlagen des Ergebnisses des zweiten Aufrufs statt Neuberechnung werden statt $1 + 2^n$ Aufrufen von power nur $1+n$ Aufrufe benötigt: Signifikanter Performanzgewinn

Memo-Tabellen 1(2)

Memo-Funktionen/Tabellen

```
memo :: (a -> b) -> (a -> b)
```

wird so benutzt, dass gilt

```
memo f x = f x
```

Stichwort: Referentielle Transparenz (also Abwesenheit von Seiteneffekten!)

Memo-Tabellen 1(2)

Die Funktion memo...

- i.w. die Identität auf Funktionen, aber...
- memo führt Buch über die Argumente, auf die es angewendet worden ist und die zugehörigen Ergebnisse
  ...statt Wiederberechnung eines Funktionswertes Nachschlagen des bereits berechneten Ergebnisses!
- Memo-Funktionen sind nicht Teil des Haskell-Standards
- Wichtige Entwurfsfrage bei der Implementierung von Memo-Funktionen: ...über wieviele Ergebnisse soll Buch geführt werden? (z.B. memo1 für ein Ergebnis)

Im Beispiel

```
mfibsFn :: () -> [Integer]
mfibsFn x = let mfibs = memo1 mfibsFn
            in 1 : 1 : zipWith (+) (mfibs ()) (tail (mfibs ()))
```

Mehr über Memo-Funktionen...

...und ihre Implementierung

Zum Beispiel in...

- Kapitel 19
  Anthony J. Field, Peter G. Harrison. *Functional Programming*, Addison-Wesley, 1988.

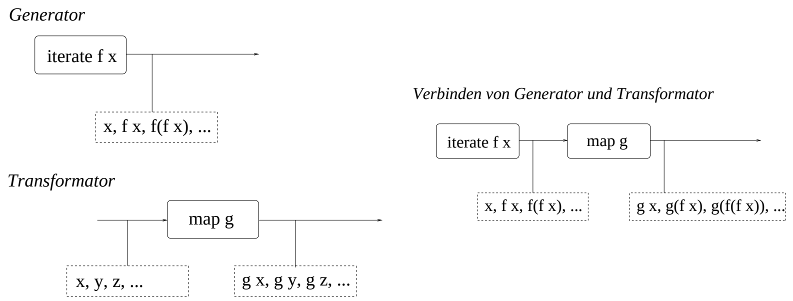
Zusammenfassung

Was spricht für Ströme (und lazy evaluation)?

- *Höhere Abstraktion* ...Einschränkungen auf endliche Listen sind oft aufwändig(er) und zugleich unnatürlich
- *Modularisierung* ...zusammen mit lazy evaluation als Auswertungsstrategie ergeben sich elegante Modularisierungspotenziale. Stichworte sind hier das *Generator/Selektor*- und das *Generator/Transformator*-Prinzip.

Generator/Transformator-Prinzip

Veranschaulichung...



Generator/Selektor-Prinzip

Veranschaulichung...