Programmiersprachen

Alexandru Berlea

Institut für Informatik
TU München

Wintersemester 2006/2007

Überblick

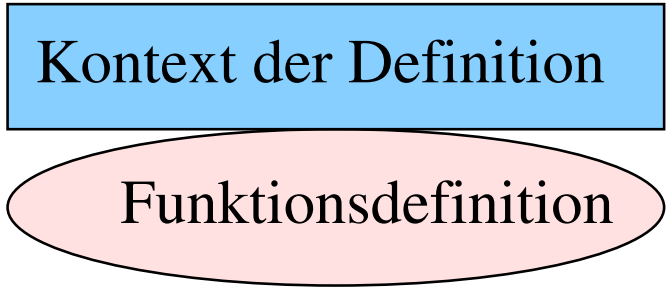
1. **Funktionale Abschlüsse**
   Currying
   Partielle Anwendung
   Funktionen höherer Ordnung

Der Funktionstyp-Operator

- **Der Funktionstyp-Operator:** $- > : MT \times MT \mapsto MT$
  - $\boxed{\alpha \; - > \; \beta} = \{f : \alpha \mapsto \beta \mid \alpha, \beta \in MT\}$
  - rechtsassoziativ: $\alpha \mapsto \beta \mapsto \gamma \equiv \alpha \mapsto (\beta \mapsto \gamma)$

Funktionale Abschlüsse

- Eine Funktion besteht aus der Funktionsdefinition und der Kontext des Programmpunktes an welchen die Funktion definiert ($\equiv$ konstruiert) wird.



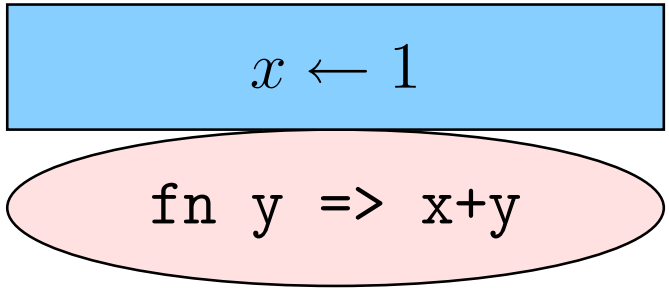
- Funktionen schließen ihren Kontext zum Zeitpunkt ihrer Definition ab.

$\Longrightarrow$ Eine Funktion wird auch **funktionaler Abschluss** (*closure*) genannt.

Funktionaler Abschluss: Beispiel

```
val x = 1
• val f = fn y => x + y
val v1 = f 3;
val v1 = 4 : int

val x =2
val v2 = f 3;
val v2 = 4 : int
```



Curry-Funktionen (*curried functions*)

Currying = Methode mit der man Funktionen von mehreren Argumenten konstruieren kann.

- genannt nach dem Erfinder, Haskell B. Curry

```
val sum = fn x => (fn y => x+y);
val sum = fn : int -> int -> int
```

`sum` erwartet ein Argument `x` und liefert eine Funktion zurück, die wiederum ein Argument `y` erwartet und `x+y` zurückliefert.

- Wegen der Rechtsassoziativität von => kann man auch schreiben:

```
val sum = fn x => fn y => x+y;
val sum = fn : int -> int -> int
```

Curry-Funktionen

- **Funktionsanwendung** (Aufruf):

```
val sum = fn x => fn y => x+y;
(sum 2) 3;
val it = 5 : int
```

- Die Funktionsanwendung ist linksassoziativ:

$$\texttt{f x1 x2 x3} \equiv \texttt{(((f x1) x2) x3)}$$

Deshalb kann man auch schreiben:

```
sum 2 3;
val it = 5 : int
```

Verkürzte Syntax

- Statt:

```
val sum = fn x => fn y => x+y;
val sum = fn : int -> int -> int
```

kann man mit verkürzter Syntax die Funktion `sum` so definieren:

```
fun sum x y = x+y;
val sum = fn : int -> int -> int
```

- I.a. ist:

```
fun f x1 x2 ... xn = expr
```

das selbe wie:

```
val f = fn x1 => fn x2 => ... fn xn => expr
```

Partielle Anwendung

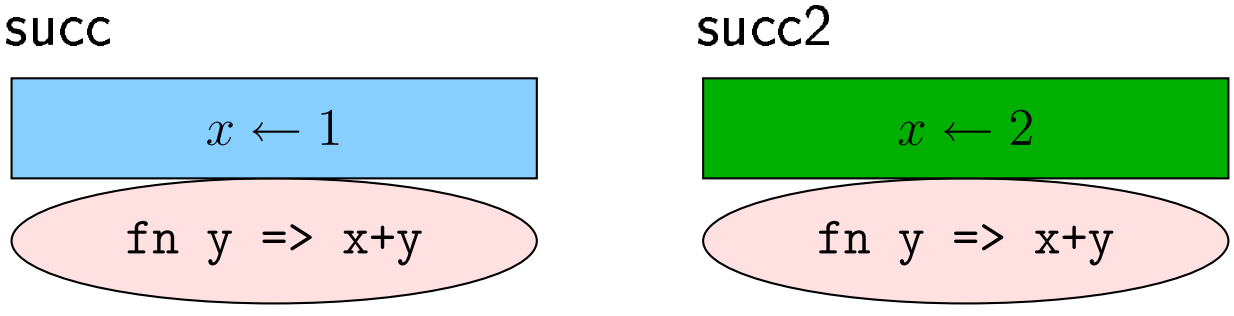
- Curry-Funktionen können **unterversorgt sein**, d.h. auf weniger Argumente als in der Deklaration angewendet werden.

- Liefern dann eine Funktion zurück, die den Rest der Argumente erwartet.
  ($\Longrightarrow$ Curry-Funktionen sind Funktionen höherer Ordnung)

```
fun f x y z t = x+y+z+t;
val f = fn : int -> int -> int -> int -> int
- val f1 = f 10;
val f1 = fn : int -> int -> int -> int
- val f2 = f1 20 30;
val f2 = fn : int -> int
- val v = f2 40;
val v = 100 : int
```

Partielle Anwendung: Beispiel

```
- val sum = fn x => fn y => x + y;
val sum = fn : int -> int -> int
- • val succ = sum 1;
val succ = fn : int -> int
- succ 16;
val it = 17 : int
- • val succ2 = sum 2;
val succ2 = fn : int -> int
- succ2 16;
val it = 18 : int
```



Funktionen höherer Ordnung

- ...bekommen Funktionen als Argumente (heißen auch Funktionale)...
- Bsp.: `map f l` wendet `f` auf jedes Element aus `l` an und liefert die Liste der Ergebnisse.

```
- fun map f l =
    case l of nil => nil
            | h::r => (f h)::map f r
val map = fn : ('a -> 'b) -> 'a list -> 'b list

- map (fn x => x+1) [1,2,3,4];
val it = [2,3,4,5] : int list
```

Funktionen höherer Ordnung

- Bsp.: `filter p l` wendet `p` auf jedes Element `x` aus `l` an und liefert die Liste aller `x`, für welche `p x` den Wert `true` hat.

```
- fun filter p l =
    case l of   nil => nil
              | h::r => if p h then h::(filter p r)
                        else (filter p r);
val filter = fn : ('a -> bool) -> 'a list -> 'a list

- fun greaterThan c x = x>c;
val greaterThan = fn : int -> int -> bool
- val greaterThanFive = greaterThan 5;
val greaterThanFive = fn : int -> bool

- filter greaterThanFive [1,6,3,7,9,4,8];
val it = [6,7,9,8] : int list
```

Funktionen höherer Ordnung:

- ...liefern Funktionen als Ergebnisse zurück:

```
fun curry f = fn x => fn y => f (x,y);
val curry = fn : ('a * 'b -> 'c) -> 'a -> 'b -> 'c

Int.max(2,7);
val it = 7 : int

- map (curry Int.max 3) [1,2,3,4,5,6];
val it = [3,3,3,4,5,6] : int list

fun uncurry f = fn (x,y) => f x y;
val uncurry = fn : ('a -> 'b -> 'c) -> 'a * 'b -> 'c

uncurry (fn x=> fn y => x+y);
val it = fn : int * int -> int
```

Das Sieb des Eratosthenes

2	3	4	5	6	7	8	9	10	11	12	13	14	15
2	3		5		7		9		11		13		15
2	3		5		7				11		13		
2	3		5		7				11		13		
2	3		5		7				11		13		

Funktionen höherer Ordnung: Beispiele

Das Sieb des Eratosthenes:

```
fun list_n i n = if i>n then nil
                 else i::(list_n (i+1) n)
val firstTen = list_n 2 10;
val firstTen = [2,3,4,5,6,7,8,9,10] : int list
```

Funktionen höherer Ordnung: Beispiele

Das Sieb des Eratosthenes:

```
fun list_n i n = if i>n then nil
                 else i::(list_n (i+1) n)
val firstTen = list_n 2 10;
val firstTen = [2,3,4,5,6,7,8,9,10] : int list


fun sieve n = filter (fn x => x mod n <> 0)
val sieve = fn : int -> int list -> int list
val l1 = sieve 2 firstTen
val it = [3,5,7,9] : int list
```

Funktionen höherer Ordnung: Beispiele

Das Sieb des Eratosthenes:

```
fun list_n i n = if i > n then nil
                 else i::(list_n (i+1) n)
val firstTen = list_n 2 10;
val firstTen = [2,3,4,5,6,7,8,9,10] : int list


fun sieve n = filter (fn x => x mod n <> 0)
val sieve = fn : int -> int list -> int list
val l1 = sieve 2 firstTen
val it = [3,5,7,9] : int list


fun iter l primes = case l of nil => primes
                  | h::r => iter (sieve h r) (h::primes)
val iter = fn : int list -> int list -> int list
```

Funktionen höherer Ordnung: Beispiele

Das Sieb des Eratosthenes:

```
fun list_n i n = if i>n then nil
                 else i::(list_n (i+1) n)
val firstTen = list_n 2 10;
val firstTen = [2,3,4,5,6,7,8,9,10] : int list


fun sieve n = filter (fn x => x mod n <> 0)
val sieve = fn : int -> int list -> int list
val l1 = sieve 2 firstTen
val it = [3,5,7,9] : int list


fun iter l primes = case l of nil => primes
                  | h::r => iter (sieve h r) (h::primes)
val iter = fn : int list -> int list -> int list

fun eratostenes n = iter (list_n 2 n) nil
eratostenes 10;
val it = [7,5,3,2] : int list
```

Überblick

2. Verzögerte Auswertung

Auswertungsstrategien

Benutzer-kontrollierte Auswertung

Unendliche Datenstrukturen

Auswertungsstrategien

- Wann werden Ausdrücke ausgewertet?
- Die meisten Sprachen legen sich auf einer Strategie fest:
  - **Strikte Auswertung** (*eager-evaluation*, strict evaluation, eifrige/vollständige Auswertung): Ein Ausdruck wird ausgewertet, sobald er an einer Variable gebunden wird.
    $\Longrightarrow$ SML, Java, C
  - **Verzögerte Auswertung** (*lazy-evaluation*, delayed evaluation): Ein Ausdruck wird ausgewertet, sobald er zur Auswertung eines umgebenden Ausdruckes gebraucht wird.
    $\Longrightarrow$ Miranda, Haskell

Parameterübergabe bei “strikter” Auswertung

- Betrachten wir den folgenden SML-Code:

```
- fun f (x,y,z) = if x=0 then y else z;
val f = fn : int * 'a * 'a -> 'a
```

Parameterübergabe bei "strikter" Auswertung

- Betrachten wir den folgenden SML-Code:

```
- fun f (x,y,z) = if x=0 then y else z;
val f = fn : int * 'a * 'a -> 'a
- fun h x = f(x,1,1 div x);
val h = fn : int -> int
```

Parameterübergabe bei "strikter" Auswertung

- Betrachten wir den folgenden SML-Code:

```
- fun f (x,y,z) = if x=0 then y else z;
val f = fn : int * 'a * 'a -> 'a
- fun h x = f(x,1,1 div x);
val h = fn : int -> int
- h 0;
uncaught exception divide by zero raised at: <file stdIn>
```

- Grund: Bei der Auswertung des Aufrufs `f(0,1,1 div 0)` werden die **aktuellen Parameter** 0, 1, `1 div 0` **ausgewertet**, wenn sie zu den **formalen Parameter** `x`, `y`, `z` gebunden werden.

- Diese Art der Übergabe der aktuellen Parameter (wie in SML,Java,C) heißt **Wertübergabe** (*call by value*)

Parameterübergabe "verzögerte" Auswertung

- Nähmen wir verzögerte Auswertung an:

```
- fun f (x,y,z) = if x=0 then y else z;
val f = fn : int * 'a * 'a -> 'a
- fun h x = f(x,1,1 div x);
val h = fn : int -> int
- h 0;
1
```

- Grund: zur Auswertung des Aufrufs `f(0,1,1 div 0)` ist die Auswertung von `1 div 0` nicht nötig.

- Wenn ein Parameter ausgewertet wird, immer wenn sein Wert gebraucht wird $\Longrightarrow$ **call by name**. (Algol)

- Wenn das Ergebnis der ersten Auswertung eines Parameter gemerkt wird, und nachträglich nachgeschlagen, immer wann der Wert gebraucht wird $\Longrightarrow$ **call by need**. (Haskell)

Benutzer-kontrollierte Auswertung

- Mit Hilfe funktionaler Abschlüsse kann man Ausdrücke **kontrolliert auswerten**.
  $\Longrightarrow$ In funktionalen Sprachen kann man eigene Auswertungsstrategien entwickeln

- Simulierung verzögerter Auswertung:

```
- fun f (x,y,z) = if x=0 then y() else z();
val f = fn : int * (unit -> 'a) * (unit -> 'a) -> 'a
- fun h x = f(x,fn () => 1,fn () => 1 div x);
val h = fn : int -> int
- h 0;
val it = 1 : int
```

Unendliche Datenstrukturen

- Ein Vorteil der verzögerten Auswertung: Darstellung unendlicher Datenstrukturen.

- Erster Versuch: Datentyp zur Darstellung unendlicher Folgen (*streams*, **Ströme**):
  - Ein Stream ist ein Paar `Stream(firstTerm,restStream)`:

```
- datatype 'a stream = Stream of 'a * 'a stream
```

Unendliche Datenstrukturen

- Ein Vorteil der verzögerten Auswertung: Darstellung unendlicher Datenstrukturen.

- Erster Versuch: Datentyp zur Darstellung unendlicher Folgen (*streams*, **Ströme**):
  - Ein Stream ist ein Paar `Stream(firstTerm,restStream)`:

```
- datatype 'a stream = Stream of 'a * 'a stream
```

```
fun generateNat n =
    Stream (n, generateNat (n+1));
val generateNat = fn : int -> int stream
```

Unendliche Datenstrukturen

- Ein Vorteil der verzögerten Auswertung: Darstellung unendlicher Datenstrukturen.
- Erster Versuch: Datentyp zur Darstellung unendlicher Folgen (*streams*, **Ströme**):
  - Ein Stream ist ein Paar `Stream(firstTerm,restStream)`:

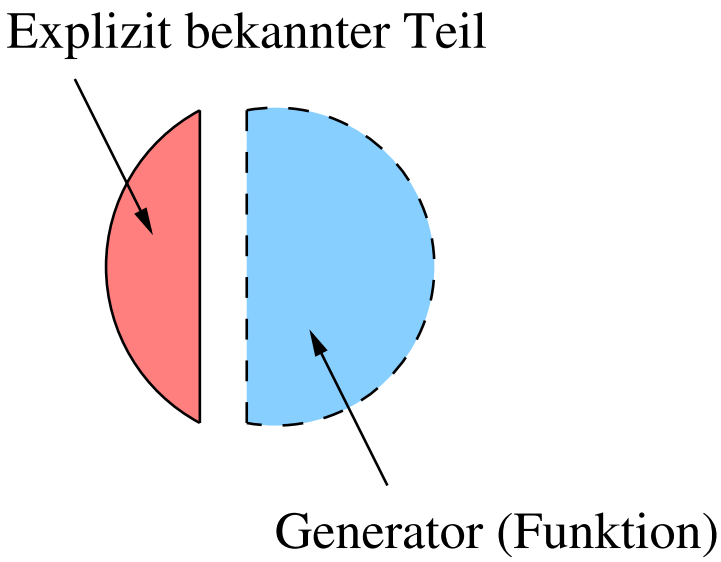
```
- datatype 'a stream = Stream of 'a * 'a stream
```

```
fun generateNat n =
    Stream (n, generateNat (n+1));
val generateNat = fn : int -> int stream
```

  - Wegen der strikten Auswertung terminiert `generateNat 0` nie.

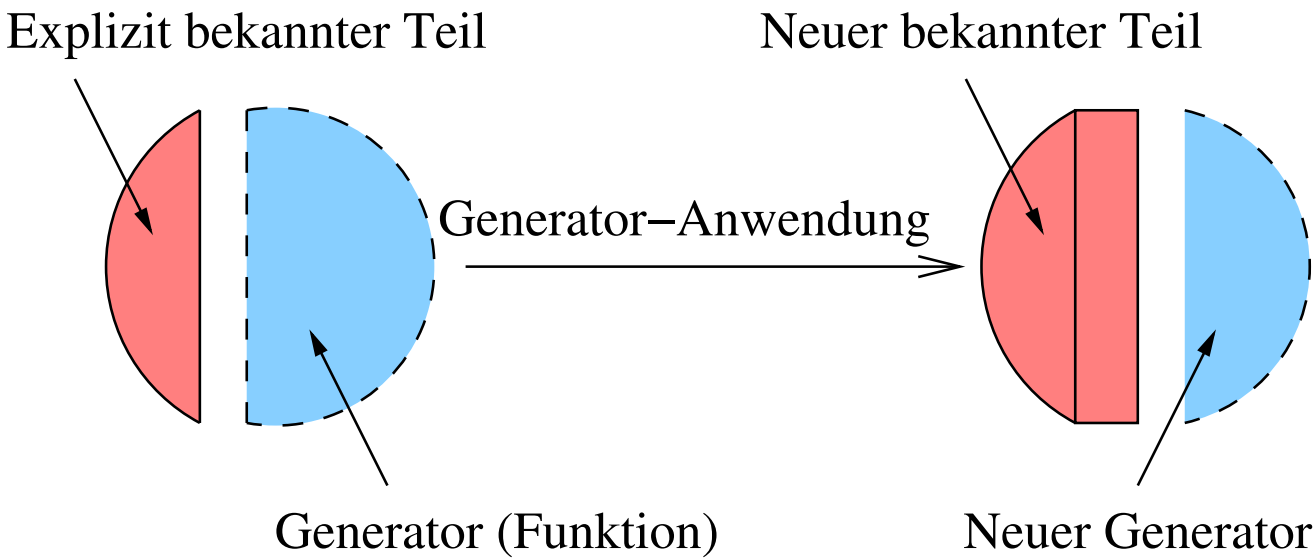
Unendliche Datenstrukturen

- Idee:



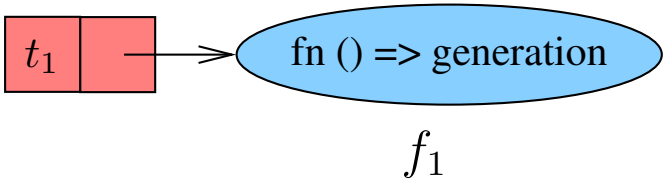
Unendliche Datenstrukturen

- Idee:

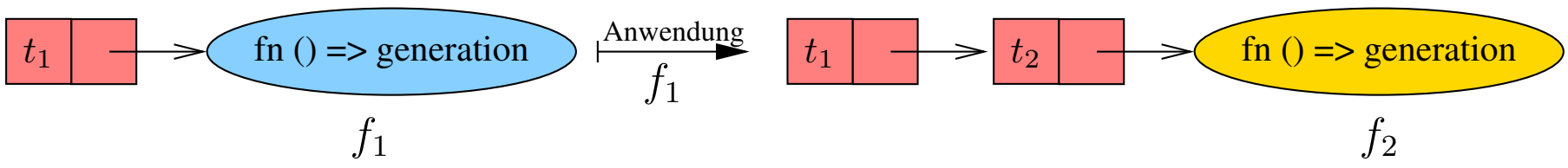


Unendliche Datenstrukturen

Unendliche Datenstrukturen



Unendliche Datenstrukturen



Unendliche Datenstrukturen

- Zweiter Versuch: mit funktionalen Abschlüssen:

```
datatype 'a stream =
    Stream of 'a * (unit -> 'a stream)
```

  - Dieser Datentyp kann keine endlich größen Daten repräsentieren, denn es fehlt einen nicht-rekursiver Konstruktor.
  - Das zweite Argument für `Stream` (Rest der Strömes) ist vom Typ `(unit -> 'a stream)`. Es ist also eine Funktion, die, wenn sie auf `()` angewandt wird, den Rest des Stroms ergibt.

Unendliche Datenstrukturen

- Zweiter Versuch: mit funktionalen Abschlüssen:

```
datatype 'a stream =
    Stream of 'a * (unit -> 'a stream)
```

  - Dieser Datentyp kann keine endlich größen Daten repräsentieren, denn es fehlt einen nicht-rekursiver Konstruktor.

  - Das zweite Argument für `Stream` (Rest der Strömes) ist vom Typ `(unit -> 'a stream)`. Es ist also eine Funktion, die, wenn sie auf `()` angewandt wird, den Rest des Stroms ergibt.

```
fun generateNat n =
      Stream (n,fn () => generateNat (n+1));
val generateNat = fn : int -> int stream
val nats = generateNat 0;
val nats = Stream (0,fn) : int stream
```

  - `generateNat 0` terminiert!

Verarbeitung unendlicher Datenstrukturen

```
- fun sum n (Stream (x,rest)) =
    if n=0 then 0
    else x + sum (n-1) (rest());
val sum = fn : int -> int stream -> int
```

- Der Rest des Stroms wird erzeugt, indem man `rest` auf `()` anwendet. Erst dadurch wird das nächste Element (und die Funktion, die den weiteren Rest des Stroms darstellt) erzeugt.

```
- sum 10 nats;
val it = 45 : int

- sum 1000 nats;
val it = 499500 : int
```

Verarbeitung unendlicher Datenstrukturen

- Analog wie bei Listen kann man eine Reihe von nützlichen Funktionen für Ströme (≡ unendliche Listen) definieren:

```
- fun head (Stream (x,_)) = x
  fun tail (Stream (_,xs)) = xs()
```

Verarbeitung unendlicher Datenstrukturen

- Analog wie bei Listen kann man eine Reihe von nützlichen Funktionen für Ströme (≡ unendliche Listen) definieren:

```
- fun head (Stream (x,_)) = x
  fun tail (Stream (_,xs)) = xs()

- head nats;
val it = 0 : int
- tail nats;
val it = Stream (1,fn) : int stream
- head(tail(tail(tail nats)));
val it = 3 : int
```

Verarbeitung unendlicher Datenstrukturen

- Analog wie bei Listen kann man eine Reihe von nützlichen Funktionen für Ströme (≡ unendliche Listen) definieren:

```
- fun head (Stream (x,_)) = x
  fun tail (Stream (_,xs)) = xs()

- head nats;
val it = 0 : int
- tail nats;
val it = Stream (1,fn) : int stream
- head(tail(tail(tail nats)));
val it = 3 : int

- fun nth n s = if n=0 then head s
                else nth (n-1) (tail s)
```

Verarbeitung unendlicher Datenstrukturen

- Extrahieren einer endlichen Teilliste:

```
- fun take n s =
    if n = 0 then nil
    else (head s)::(take (n-1) (tail s));
val take = fn : int -> 'a stream -> 'a list
- take 10 nats;
val it = [0,1,2,3,4,5,6,7,8,9] : int list
```

Verarbeitung unendlicher Datenstrukturen

- Extrahieren einer endlichen Teilliste:

```
- fun take n s =
    if n = 0 then nil
    else (head s)::(take (n-1) (tail s));
val take = fn : int -> 'a stream -> 'a list
- take 10 nats;
val it = [0,1,2,3,4,5,6,7,8,9] : int list
```

- Funktionen höherer Ordnung (*Funktionale*):

```
fun map f s = Stream (f (head s),
                      fn () => map f (tail s))
val map = fn : ('a -> 'b) -> 'a stream -> 'b stream
```

Verarbeitung unendlicher Datenstrukturen

- Extrahieren einer endlichen Teilliste:

```
- fun take n s =
    if n = 0 then nil
    else (head s)::(take (n-1) (tail s));
val take = fn : int -> 'a stream -> 'a list
- take 10 nats;
val it = [0,1,2,3,4,5,6,7,8,9] : int list
```

- Funktionen höherer Ordnung (*Funktionale*):

```
fun map f s = Stream (f (head s),
                      fn () => map f (tail s))
val map = fn : ('a -> 'b) -> 'a stream -> 'b stream

fun filter f s =
   if f (head s) then
      Stream(head s,fn () => filter f (tail s))
   else filter f (tail s)
val filter = fn : ('a -> bool) -> 'a stream -> 'a stream
```

Verarbeitung unendlicher Datenstrukturen

- Jetzt können wir z.B. die unendliche Liste aller geraden Zahlen oder aller Quadratzahlen berechnen:

```
- take 10 (filter (fn x => x mod 2=0) nat);
val it = [0,2,4,6,8,10,12,14,16,18] : int list


- take 10 (map (fn x => x*x) nat);
val it = [0,1,4,9,16,25,36,49,64,81] : int list
```

Unendliche Datenstrukturen

- So können wir auch die Liste aller Primzahlen berechnen (Sieb des Eratosthenes):

```
fun all_primes () =
 let
   fun sieve (Stream (n,ns)) =
     Stream
      (n,
       fn() => sieve
                (filter (fn x => x mod n<>0) (ns())))
      )
 in
   sieve (generateNat 2)
 end;
```

Unendliche Datenstrukturen

- So können wir auch die Liste aller Primzahlen berechnen (Sieb des Eratosthenes):

```
fun all_primes () =
 let
   fun sieve (Stream (n,ns)) =
     Stream
      (n,
       fn() => sieve
                (filter (fn x => x mod n<>0) (ns())))
      )
 in
   sieve (generateNat 2)
 end;

- take 10 (all_primes());
val it = [2,3,5,7,11,13,17,19,23,29] : int list
```

Unendliche Datenstrukturen

```
take 200 (all_primes());
val it =
[2,3,5,7,11,13,17,19,23,29,31,37,41,43,47,53,59,61,67,71,73,79,83,89,97,101,
 103,107,109,113,127,131,137,139,149,151,157,163,167,173,179,181,191,193,197,
 199,211,223,227,229,233,239,241,251,257,263,269,271,277,281,283,293,307,311,
 313,317,331,337,347,349,353,359,367,373,379,383,389,397,401,409,419,421,431,
 433,439,443,449,457,461,463,467,479,487,491,499,503,509,521,523,541,547,557,
 563,569,571,577,587,593,599,601,607,613,617,619,631,641,643,647,653,659,661,
 673,677,683,691,701,709,719,727,733,739,743,751,757,761,769,773,787,797,809,
 811,821,823,827,829,839,853,857,859,863,877,881,883,887,907,911,919,929,937,
 941,947,953,967,971,977,983,991,997,1009,1013,1019,1021,1031,1033,1039,1049,
 1051,1061,1063,1069,1087,1091,1093,1097,1103,1109,1117,1123,1129,1151,1153,
 1163,1171,1181,1187,1193,1201,1213,1217,1223] : int list
```