Partielle Anwendung

- Curry-Funktionen können unterversorgt sein d.h. auf weniger Argumente angewendet werden als in der Deklaration
- Liefern dann eine Funktion zurück, die den Rest der Argumente erwartet

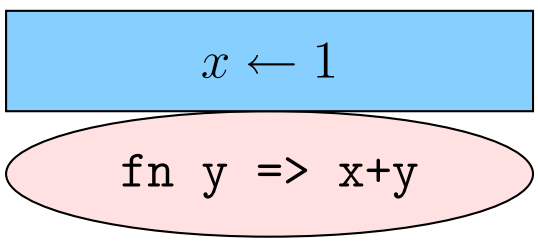
($\Longrightarrow$ Curry-Funktionen sind Funktionen höherer Ordnung)

```
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
```

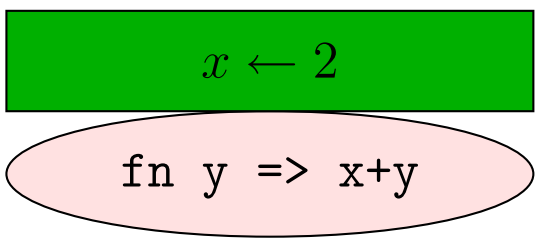
Partielle Anwendung: Beispiel

```
- 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
```

succ



succ2



4.7.4 Funktionen höherer Ordnung: Beispiele

... 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 zurück.

```
- 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: Beispiele

... bekomen Funktionen als Argumente:

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

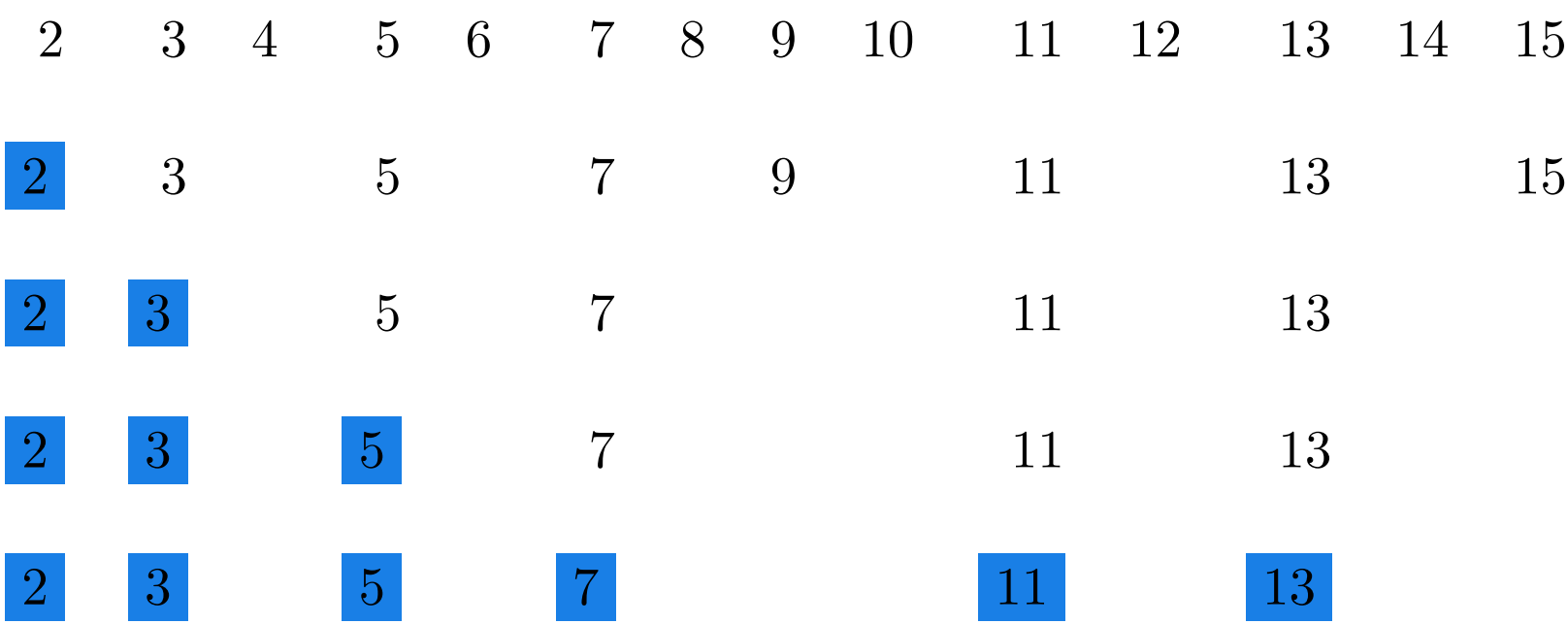
```
- fun filter p l =
    case l of   nil => nil
              | h::r => if p h then h::(filter p r)
                        else (filter p r);
val filter = fn : ('a -> bool) -> 'a list -> 'a list
- fun greaterThan c x = x>c;
val greaterThan = fn : int -> int -> bool
- val greaterThanFive = greaterThan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

... 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 de Eratosthenes



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
```

4.8 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
- 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 Aurufes `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 Aufrufes `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 $\Longrightarrow$ **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)

4.8.1 Benutzer-kontrollierte Auswertung

Mit Hilfe funktionaller 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
```

4.8.2 Unendliche Datenstrukturen

Ein Vorteil der verzögerten Auswertung: Darstellung infiniter Datenstrukturen.

Erster Versuch: Datentyp zur Darstellung unendlicher Folgen (*streams*, Ströme):

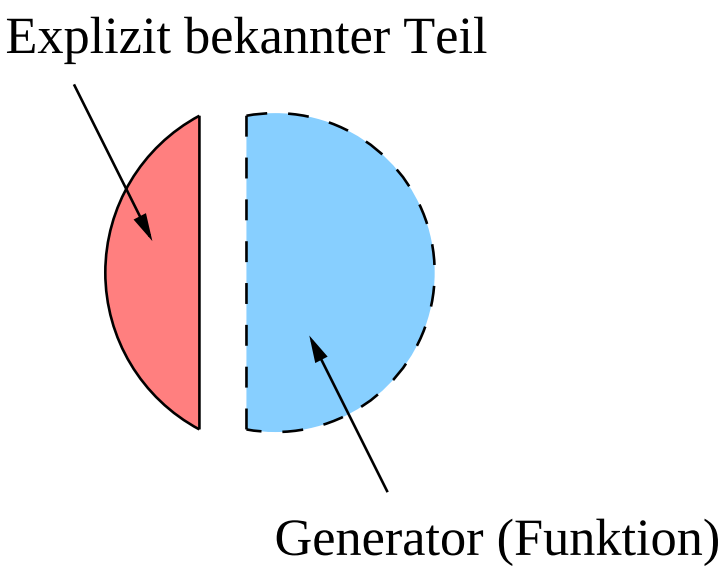
```
- datatype 'a stream = Stream of 'a * 'a stream
  fun generateNat n = Stream (n,generateNat (n+1));
val generateNat = fn : int -> int stream
```

Ein Stream ist ein Paar `Stream(firstTerm,restStream)`:

Wegen der strikten Auswertung terminiert `generateAllNat 0` nie.

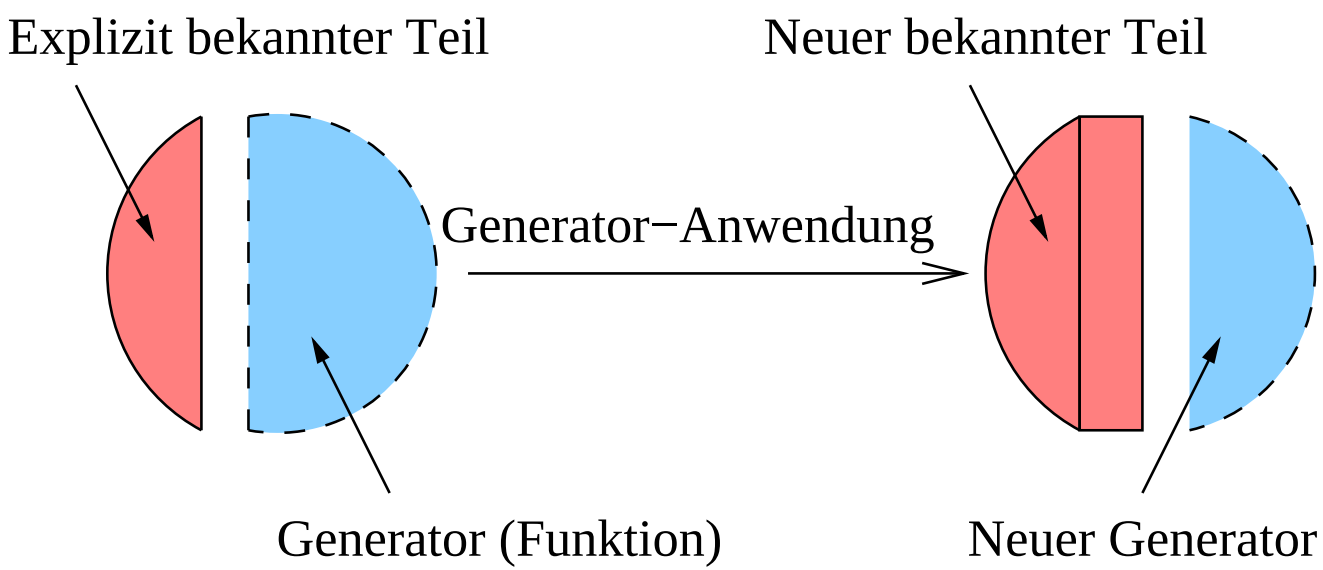
Unendliche Datenstrukturen

Idee:



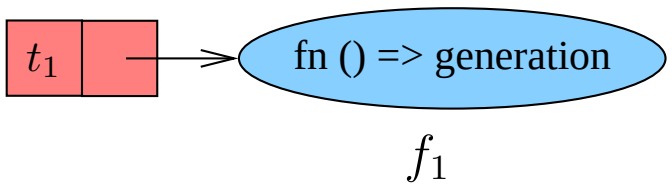
Unendliche Datenstrukturen

Idee:

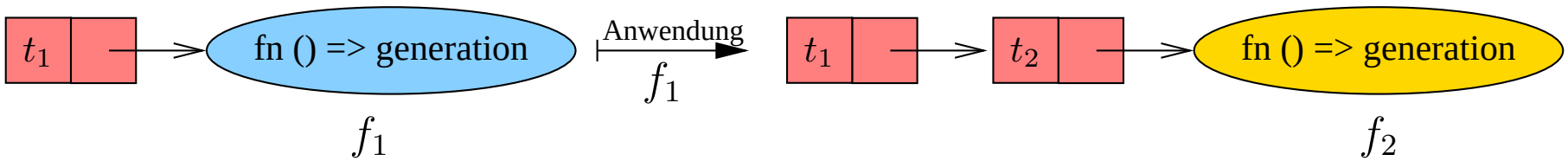


Unendliche Datenstrukturen

Unendliche Datenstrukturen



Unendliche Datenstrukturen



Unendliche Datenstrukturen

Zweiter Versuch: mit funktionallen Abschlüssen:

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

- Dieser Datentyp kann keine endlich großen 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()
  fun nth n s = if n=0 then head s else nth (n-1) (tail s)
- 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

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
```