7. Unendliche Listen (lazy lists), Übersicht

Paradigma: Strom von Werten
Produzent und Konsument getrennt entwerfen
Konsument entscheidet über Abbruch (Terminierung)

Beispiele:		
	Zahlenfolge	summieren
	iteratives Näherungsverfahren	Abbruchkriterium
	Zufallszahlen generieren	benutzen
	Lösungsraum aufzählen	über Lösung entscheiden

Technik:

Liste: Paar aus Element und Rest

Strom: Paar aus Element und **Funktion, die den Rest liefert** (parameterlose Funktion)

```
datatype 'a seq = Nil | Cons of 'a * (unit -> 'a seq);

fun  Head (Cons (x, xf)) = x
|    Head Nil            = raise Empty;
fun  Tail (Cons (x, xf)) = xf ()
|    Tail Nil            = raise Empty;
```


Beispiele für Stromfunktionen (1)

Produzent eines Zahlenstromes: `int -> int seq`

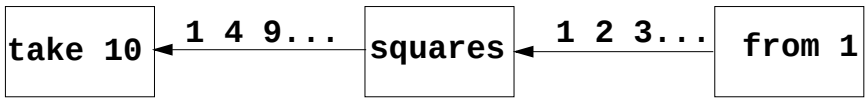
```
fun from k = Cons (k, fn()=> from (k+1));
```

Konsument: erste n Elemente als Liste: `'a seq * int -> 'a list`

```
fun take (xq, 0)          = []
|   take (Nil, n)         = raise Empty
|   take (Cons(x, xf), n) = x :: take (xf (), n - 1);
```

Transformer: `int seq -> int seq`

```
fun squares Nil = Nil
|   squares (Cons (x, xf)) = Cons (x * x, fn() => squares (xf()));

        take          (squares          (from 1), 10);
```



Beispiele für Stromfunktionen (2)

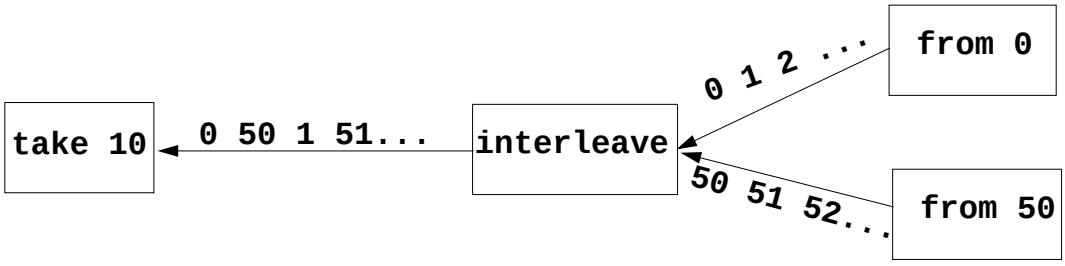
zwei Ströme addieren: `int seq * int seq -> int seq`

```
fun add (Cons(x, xf),Cons(y, yf)) =
            Cons (x+y, fn() => add (xf(), yf()))
|   add _ = Nil;
```

zwei Ströme verzahnen: `'a seq * 'a seq -> 'a seq`

```
fun interleave (Nil, yq) = yq
|   interleave (Cons(x, xf), yq) =
            Cons (x, fn () => interleave(yq, xf ()));
```

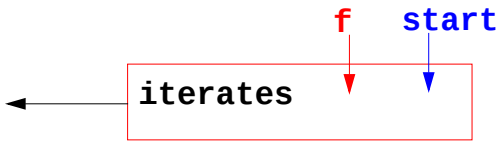
```
take (interleave (from 0, from 50), 10)
```



Funktionale für Ströme

Generator-Schema: wiederholte Anwendung einer Funktion auf einen Startwert

```
fun iterates f x =Cons (x, fn() => iterates f (f x));
        ('a -> 'a) -> 'a -> 'a seq
```




```
fun from k = iterates (secl 1 op+) k;
```

Transformer-Schema:


```
    ('a -> 'b) -> 'a seq -> 'b seq
fun map f Nil  = Nil
|   map f (Cons(x,xf)) = Cons (f x, fn () => map f (xf()));
```

Filter-Schema:


```
    ('a -> bool) -> 'a seq -> 'a seq
fun filter pred Nil = Nil
|   filter pred (Cons(x,xf)) =
        if pred x then Cons (x, fn()=> filter pred (xf()))
                  else filter pred (xf());
```

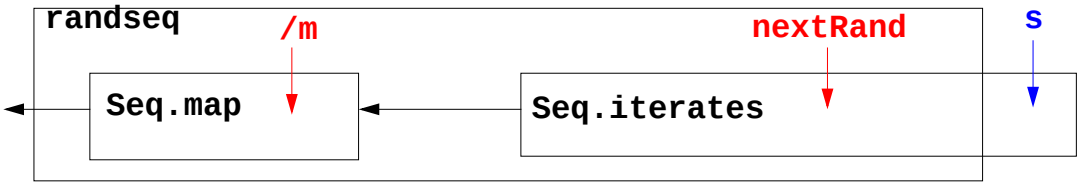
Stromfunktionen im Modul Seq

Funktionen für Ströme sind im Modul Seq zusammengefasst:

```
Seq.hd, Seq.tl, Seq.null, Seq.take, Seq.drop, Seq.@,
Seq.interleave, Seq.map, Seq.filter, Seq.iterates,
Seq.from, Seq.fromlist, Seq.tolist
```

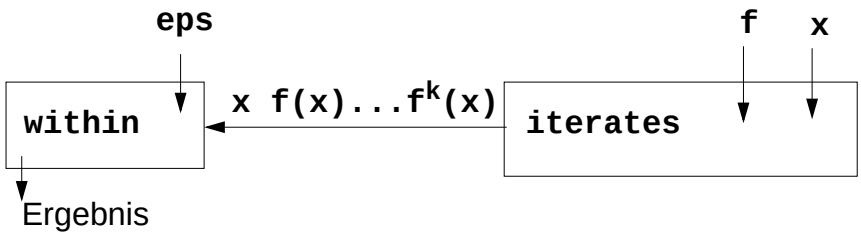
Beispiel: Strom von Zufallszahlen:

```
localval a = 16807.0  and  m = 2147483647.0

     fun nextRand seed =
          let val t = a*seed
          in  t - m * real (Real.floor(t/m))
          end

in   fun randseq s = Seq.map(secr op/ m)
                        (Seq.iterates nextRand (real s))
end;
```



Ströme zusammensetzen

Schema: Konvergenzabbruch für iterierte Funktion

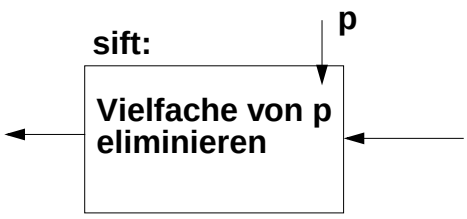


Beispiel: Quadratwurzel iterativ berechnen:

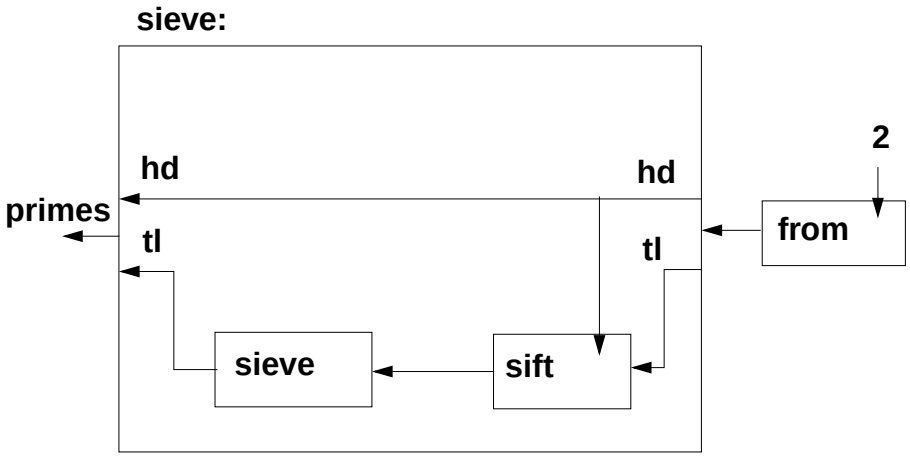
```
fun nextApprox a x = (a/x + x) / 2.0;


fun within (eps:real) (Cons(x,xf)) =
   let val Cons (y,yf) = xf()
   in   if Real.abs (x-y) < eps
        then y
        else within eps (Cons (y,yf))
   end;


fun qroot a =
     within  1E~12    (Seq.iterates (nextApprox a) 1.0);
```

Ströme rekursiv zusammensetzen

```
fun sift p =
   Seq.filter (fn n => n mod p <> 0);

fun sieve (Cons(p,nf)) =
   Cons (p, fn() => sieve (sift p (nf())));

val primes = sieve (Seq.from 2);
Seq.take (primes, 25);
```

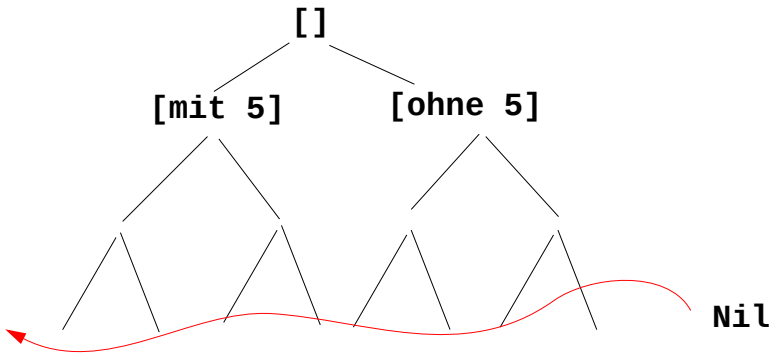


Primzahlen mit dem Sieb des Eratosthenes berechnen:



Strom aller Lösungen im Baum-strukturierten Lösungsraum

Beispiel **Münzwechsel**: Strom von Lösungen der Form [5, 2, 1, 1] berechnen



- endliche Zahl von Lösungen: abbrechender Strom
- **Listenkonkatenation** @ darf **nicht in Stromkonkatenation** Seq.@ geändert werden! Strom würde dann **vollständig ausgewertet**!
- Funktion akkumuliert Strom elementweise
- akkumulierender Parameter berechnet Restfunktion des Stromes mit **`Cons (x, xf)`**

Beispiel Münzwechsel mit Strömen

Signatur:

```
int list * int list * int * (unit -> int list seq) -> int list seq
```

Funktionsdefinition seqChange:

```
fun
```

neue Lösung **coins** in den Strom geben:

```
   seqChange (coins, coinvals, 0, coinsf) = Seq.Cons (coins, coinsf)
```

ist keine Lösung, Strom bleibt unverändert:

```
|  seqChange (coins, [],  amount, coinsf) = coinsf ()
|  seqChange (coins, c::coinvals, amount, coinsf)=
      if amount<0
```

ist keine Lösung, Strom bleibt unverändert:

```
      then coinsf ()
      else seqChange
```

erster Zweig „mit Münze c“:

```
             (c::coins, c::coinvals, amount-c,
```

zweiter Zweig „ohne Münze c“, lazy:

```
              fn() => seqChange (coins, coinvals, amount, coinsf));
```

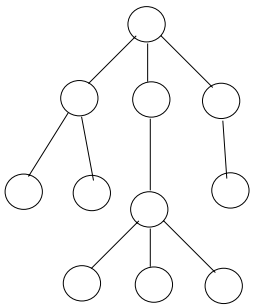
Aufruf mit abbrechender Rest-Funktion:

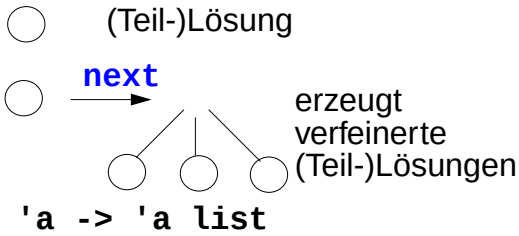
```
   seqChange ([], gb_coins, 99, fn () => Seq.Nil);
```

liefert die erste Lösung im Paar `Seq.Cons ([...], f);` die nächste mit `Seq.tl it`

Funktional für Tiefensuche in Lösungsbäumen

- Strom entkoppelt Erzeuger und Verwender der Lösungen
- Funktional bestimmt die Suchstrategie des Erzeugers
- Die Aufgabe wird durch **next** und **pred** bestimmt



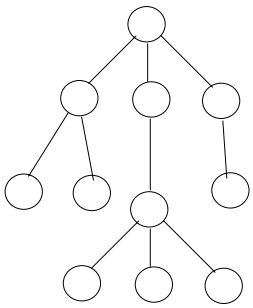


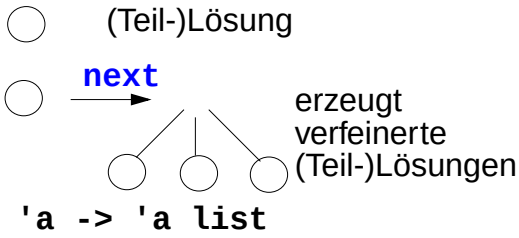
DFS Tiefensuche: effizient; aber terminiert nicht bei unendlichen Teilbäumen
Prädikat **pred** entscheidet, ob eine Lösung vorliegt:

```
fun depthFirst (next, pred) root =
   let  fun dfs [] = Nil
        |   dfs (x::xs) =
              if pred x                                  Keller:
                 then Cons (x, fn () =>dfs ((next x) @ xs))
                 else dfs ((next x) @ xs)
   in dfs [root] end;
```

Funktional für Breitensuche in Lösungsbäumen

- Strom entkoppelt Erzeuger und Verwender der Lösungen
- Funktional bestimmt die Suchstrategie des Erzeugers
- Die Aufgabe wird durch **`next`** und **`pred`** bestimmt





BFS Breitensuche: vollständig; aber speicheraufwendig:

```
fun breadthFirst (next, pred) root =
   let  fun bfs [] = Nil
        |   bfs (x::xs) =
             if pred x                                  Schlange:
                 then Cons (x, fn () => bfs(xs @ next x))
                 else bfs (xs @ next x)
   in bfs [root] end;
```


Funktionale anwenden für Münzwechsel

Knoten des Lösungsbaumes sind Tripel
(ausgezahlte Münzen, verfügbare Münzwerte, zu zahlender Betrag):

```
fun predCoins (paid, coinvals, 0) = true
|   predCoins _                   = false;

fun nextCoins (paid, coinvals, 0) = []
|   nextCoins (paid, nil, amount) = []
|   nextCoins (paid, c::coinvals, amount) =
      if amount < 0
         then []
         else [ (c::paid, c::coinvals, amount-c),
                (paid, coinvals, amount)];

val euro_coins = [200, 100, 50, 20, 10, 5, 2, 1];
val coins52Dep = depthFirst (nextCoins, predCoins) ([],[5,2], 30);
val coins52Bre = breadthFirst (nextCoins, predCoins) ([],[5,2], 30);
val coinsEuroBre = ([], euro_coins, 30);
```

Funktionale anwenden erzeugung von Palindromen

Ein Knoten des Lösungsbaumes ist eine Liste von Zeichen:

```
fun nextChar l = [#"A"::l, #"B"::l, #"C"::l];
fun isPalin l = (l = rev l);

val palinABCBre = breadthFirst (nextChar, isPalin) [];
val palinABCDep = depthFirst (nextChar, isPalin) [];
```


Weiter verzögerte Auswertung

Datentyp **lazySeq** berechnet ein **Paar erst, wenn es gebraucht wird**:

```
datatype 'a lazySeq = LazyNil | LazyCons of unit -> 'a * 'a lazySeq

   fun from k = LazyCons (fn () => (k, from (k + 1)));


   fun  take (xq, 0)           =  nil
   |    take (LazyNil, n)      =  raise Seq.Empty
   |    take (LazyCons xf, n)  =  let val (x, xt) = xf ()
                                  in x :: take (xt, n - 1))
                                  end;
```

noch weiter verzögert: leerer oder nicht-leerer Strom wird erst entschieden, wenn nötig.

```
datatype 'a seqNode = llNil | llCons of 'a * 'a llSeq;

datatype 'a llSeq = Seq of unit -> 'a seqNode;
```