

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
```

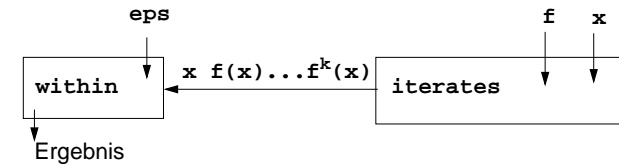
```
fun nextRand seed =  
  let val t = a*seed  
      in t - m * real (Real.floor(t/m))  
      end
```

```
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;
```

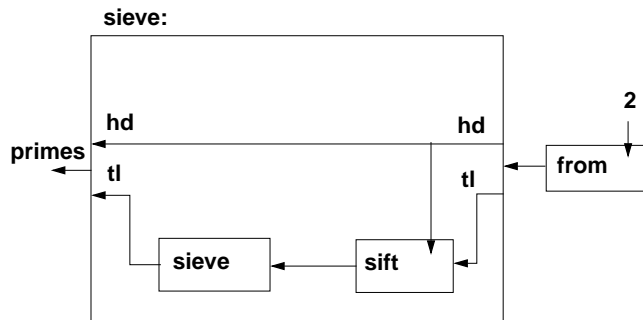
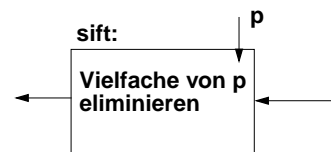
```
fun qroot a =  
  within 1E-12 (Seq.iterates (nextApprox a) 1.0);
```

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())));
```

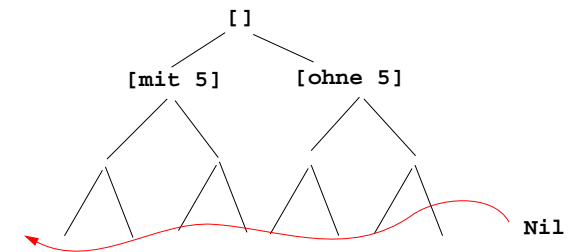
```
val primes = sieve (Seq.from 2);  
Seq.take (primes, 25);
```



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));
```

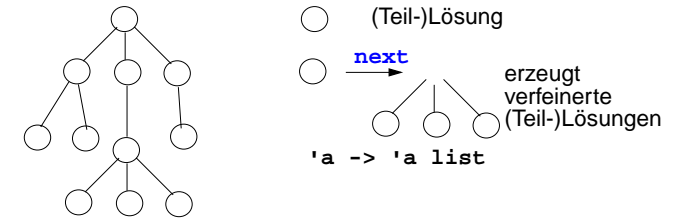
Aufruf mit abbrechender Rest-Funktion:

```
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



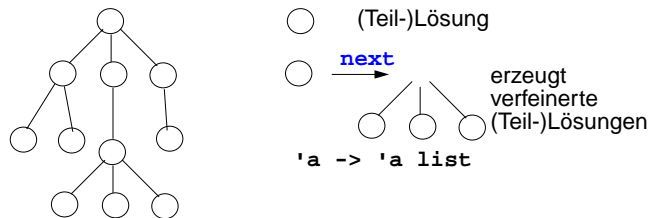
DFS Tiefensuche: effizient; aber terminiert nicht bei unendlichen Teilbäumen
Prädikat **pred** entscheidet, ob eine Lösung vorliegt:

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

Keller:

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
              then Cons (x, fn () => bfs(xs @ next x))
              else bfs (xs @ next x)
  in bfs [root] end;
```

Schlange:

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 = l1Nil | l1Cons of 'a * 'a l1Seq;
datatype 'a l1Seq = Seq of unit -> 'a seqNode;
```