1 Lusta lista

Az SML alapvetően mohó kiértékelésű, de lehet vele *lusta listát* definiálni: a lusta lista *farka* függvény, ezáltal *késleltetjük* a kiértékelését. Ily módon *végtelen listákat* hozhatunk létre, amelyeknek előnyős tulajdonságaik mellett hátrányaik, veszélyeik is vannak, pl.

- egy lusta lista *bármely részét* megjeleníthetjük, de *sohasem az egészet;*
- két lusta lista elemeiből páronként képezhetünk egy harmadikat, de például *nem számíthatjuk ki* egy lusta lista *elemeinek összegét,* nem kereshetjük meg benne *a legkisebbet,* nem fordíthatjuk meg az *elemek sorrendjét*;
- úgy kell rekurziót definiálnunk, hogy *nincs alapeset;*
- egy program befejeződése helyett csak azt igazolhatjuk, hogy az eredmény *tetszőleges véges része véges idő alatt* előáll.

Az SML-ben a lusta listákat kezelő függvények bonyolultabbak, mint egy lusta kiértékelést alkalmazó nyelvben, mert a lista farkában *explicite* hivatkoznunk kell a függvényre.
A lusta listát *sorozatnak* (sequence) fogjuk nevezni, és a `seq` típusoperátort használjuk a létrehozására.[1]

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

Egy sorozat fejét, ill. farkát adják eredményül az alábbi `head`, ill. `tail` függvények; mindkettõ abortál, ha üres sorozatra alkalmazzák.

```
- fun head (Cons(x, _)) = x;
> val head = fn :'a seq -> 'a
```

A sorozat farka `unit -> 'a seq` típusú függvény, erre illesztjük az `xf` mintát; `tail` törzsében `xf`-et a `()` argumentumra kell alkalmazni:

```
- fun tail (Cons(_, xf)) = xf();
> val tail = fn :  'a seq -> 'a seq
```

`consq(x, xq)` `x`-et berakja az `xq` sorozatba:

```
- fun consq (x, xq) = Cons (x, fn () => xq);
> val consq = fn :  'a * 'a seq -> 'a seq
```

Ha a `consq` függvényt alkalmazzuk, mondjuk, az `(x,E)` argumentumra, a `consq(x,E)` kifejezést az SML *nem lustán* értékeli ki, hiszen az SML alapvetően mohó kiértékelésű. Ha `E` kiértékelésének eredményét `xq`-val jelöljük, akkor `consq(x,E)` kiértékelése a fenti definíció szerint `Cons(x, fn () => xq)`-t eredményez. A `consq`-beli `fn () => xq` függvény *nem késlelteti* a farok (a példában `E`) kiértékelését `consq` alkalmazásakor. Az SML-ben a lusta kiértékelés érdekében a híváskor is a `Cons(x, fn () => E)` alakot kell használnunk, `consq(x,E)` nem jó. Az *explicit* `fn () => E` *késlelteti* a kiértékelést, és ezzel *szükség szerinti* hivatkozást valósít meg.

[1] Mivel az SML különbséget tesz a kis- és nagybetűk között, a hagyományos lista `nil` és a lusta lista `Nil` konstruktora nem azonosak. A `unit` típus, mint tudjuk, a típusműveletek *egységeleme;* egyetlen elemét `()`-lel jelöljük.

Példaként a korábban megismert `from` és `take` függvények lusta változatait mutatjuk be. A `fromq k` sorozat egészek `k`-tól induló végtelen sorozata:

```
- fun fromq k = Cons(k, fn () => fromq(k+1));
> val fromq = fn :  int -> int seq
- head (fromq 1);
> val it = 1
- fromq 1;
> val it = Cons (1, fn):  int seq;
- tail it;
> val it = Cons (2, fn):  int seq;
- tail it;
> val it = Cons (3, fn):  int seq;
```

`takeq(n, xq)` az `xq` sorozat első `n` eleméből képzett listát adja vissza:

```
- fun takeq (0, xq) = []
    | takeq (n, Cons (x, xf)) = x ::  takeq(n-1, xf())
    | takeq (n, Nil) = [];
> val takeq = fn :  int * 'a seq -> 'a list
```

Nézzünk egy példát `takeq` egyszerűsítésére!

```
takeq(2, fromq 30) →
takeq(2, Cons(30, fn () => fromq(30+1))) →
30::takeq(1, fromq(30+1)) →
30::takeq(1, Cons(31, fn () => fromq(31+1))) →
30::(31::takeq(0, fromq(31+1))) →
30::31::takeq(0, Cons(32, fn () => fromq(32+1))) →
30::31::[] → [30,31]
```

A 32-t az SML ugyan kiszámítja, de sohasem használja fel.
Az `'a seq` típus nem egészen lusta kiértékelésű: egy nemüres sorozat fejét a rendszer mindig feldolgozza, és ezt az SML-ben csak körülményesen lehet elkerülni.[2]

1.1 Elemi feldolgozási műveletek sorozatokkal

A kiszámíthatóság érdekében egy függvény eredményének tetszőleges, véges része az argumentum véges részétől függhet csak. Amikor az eredményre *szükség* van, akkor ez az igény *váltja ki* az argumentum feldolgozását.
Nézzünk egy példát, amelyben egészeket egyesével emelünk négyzetre. Amikor szükség van rá, az eredmény farka - egy függvény - alkalmazza a `squareq` függvényt az argumentum farkára.

```
- fun squareq Nil:  int seq = Nil
    | squareq (Cons (x, xf)) = Cons(x * x, fn () =>
squareq(xf()));
- squareq(fromq 1);
> val it = Cons(1, fn) :  int seq
- takeq(10, it);
> [1, 4, 9, 16, 25, 36, 49, 64, 81, 100] :  int list
```

[2] Lásd L.C. Paulson: SML for the Working Programmer, Cambridge Press, 1991, 168. o.

Két sorozat hasonlóan adható össze:

```
  - fun addq (Cons (x, xf), Cons(y, yf)) =
                                 Cons(x+y, fn () => addq(xf(),
yf()))
      | addq _:  int seq = Nil;
  > val addq = fn :  (int seq * int seq) -> int seq
  - addq(fromq 1000, squareq(fromq 1));
  > val it = Cons(1001, fn) :  int seq
  - takeq (5, it);
  > val it = [1001, 1005, 1011, 1019, 1029] :  int list
```

Az `appendq` függvény addig nem nyúl `yq`-hoz, amíg `xq` ki nem ürül - vagyis csak akkor nyúl hozzá, ha `xq` véges. Véges sorozatot `consq`-val készíthetünk. Most érthetjük meg, hogy miért kellett a típusdefinícióban a `Nil` konstruktorállandót definiálni.

```
  - fun appendq (Cons (x, xf), yq) = Cons(x, fn () => appendq
(xf(), yq))
      | appendq (Nil, yq) = yq;
  > val appendq = fn :  'a seq * 'a seq -> 'a seq

  - val finiteq = consq(25, consq(10, Nil));
  > val it = Cons(25, fn) :  int seq
  - appendq(finiteq, fromq 1526);
  > val it = Cons(25, fn) :  int seq
  - takeq (4, it);
  > val it = [25, 10, 1526, 1527] :  int list
```

1.2 Magasabb rendű függvények sorozatokra

Két jól ismert magasabb rendű függvény - `map` és `filter` - lusta változatát definiáljuk ebben a szakaszban.

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

  - fun filterq p (Cons (x, xf)) = if p x
                                   then Cons(x, fn () => filterq
p (xf()))
                                   else filterq p (xf())
      | filterq p Nil = Nil;
  > val filterq = fn :  ('a -> bool) -> 'a seq -> 'a seq
```

Az előző szakaszban definiált `squareq` sokkal egyszerűbben definiálható `mapq`-val; `f`-ként egészek esetén `sq'i`-t, valósak esetén `sq'r`-et kell használni, pl.

```
  - val squareq = mapq sq'i;
```

Most olyan számsorozatot állítunk elő, amelyben 50-nél nagyobb, 7-esre végződő egész számok vannak:


```
  - filterq (fn n => n mod 10 = 7) (fromq 50);
  > val it = Cons(57, fn) :  int seq
  - takeq(8, it);
  > val it = [57, 67, 77, 87, 97, 107, 117, 127] :  int list
```

Az ugyancsak magasabbrendű `iterateq` függvény a `fromq` egy általánosítása, az
$[x, f(x), f(f(x)), \cdots, f^k(x), \cdots]$
sorozatot állítja elő (v.ö. a korábban definiált `repeat`-tel).

```
  - fun iterateq f x = Cons(x, fn () => iterateq f (f x));
  > val iterateq = fn :  ('a -> 'a) -> 'a -> 'a seq
  - iterateq (secr op/ 2.0) 1.0;
  > val it = Cons (1.0, fn) :  real seq
  - takeq (5, it);
  > val it = [1.0, 0.5, 0.25, 0.125, 0.625] :  real list
```

`fromq`-t `iterateq`-val úgy kaphatjuk meg, hogy `f`-ként a `succ` függvényt alkalmazzuk:

```
  - fun succ k = k + 1;
  - val fromq = iterateq succ
```

1.3 Összetett példák

1.3.1 Álvéletlen-számok

A hagyományos álvéletlenszám-generátorok olyan eljárások, amelyek egy változóban tárolják a `seed` *(mag)* értéket - ebből állítják elő a következő hívásnál a következő álvéletlenszámot. Ha sorozatként valósítjuk meg az álvéletlenszámokat, akkor a következő álvéletlenszám csak ***szükség esetén*** áll elő.

```
  - local val a = 16807.0 and m = 2147483647.0
          fun nextrandom seed =
                let val t = a * seed
                in t - real(floor(t/m)) * m
                end
    in
       fun randseq s = mapq (secr op/ m) (iterateq nextrandom
(real s))
    end;
  > val randseq = fn :  int -> real seq
```

Ha `nextrandom`-ot 1.0 és 21474836467.0 közötti `seed`-értékre alkalmazzuk, ugyanebbe a tartományba eső más értéket állít elő az `a * seed mod m` művelettel. A valós számokat csak a túlcsordulás elkerülésére használjuk.
A sorozat előállítására `iterateq`-et `nextrandom`-ra és `seed` valós számmá alakított kezdőértékére alkalmazzuk. `mapq` gondoskodik arról, hogy a sorozatban minden értéket elosszunk `m`-mel, és így `randseq` 0.0-nál nem kisebb

és 1.0-nél kisebb értékeket adjon eredményül. Látható, hogy a sorozat a megvalósítás részleteit szépen elrejti a felhasználó elől.[3]
Az előállított álvéletlen-számok tehát 0.0-nál nem kisebb és 1.0-nél kisebb valós számok; `mapq`-val alakíthatjuk át őket 0 és 1 közötti egészekké:

```
- mapq (floor o secl 10.0 op* ) (randseq 1);
> val it = Cons(0, fn):  int seq
- takeq(5, it);
> val it = [0, 0, 1, 7, 4]:  int list
```

1.3.2 Prímszámok

Következő SML-programunk az eratosztenészi szita megvalósítása. A jól ismert algoritmust egy példa bemutatásával idézzük föl:

1. Vegyük az egészek 2-vel kezdődő sorozatát: [2, 3, 4, 5, 6, 7, ...]
2. Töröljük az összes 2-vel osztható számot: [3, 5, 7, 9, 11, ...]
3. Töröljük az összes 3-mal osztható számot: [5, 7, 11, 13, 17, 19, ...]
4. Töröljük az összes 5-tel osztható számot: [7, 11, 13, 17, 19, ...]
5. Töröljük az összes ...

Látható, hogy a sorozat első eleme mindig a következő prím. A sorozatban azok a számok maradnak benne, amelyek az eddig előállított prímekkel nem oszthatók.

```
- fun sift p = filterq (fn n => n mod p <> 0);
```

A `sift` (szitál, rostál) segédfüggvény a `p` argumentum többszöröseit törli egy sorozatból. A `sieve` (szita, rosta) függvénynek már csak ismételten alkalmaznia kell `sift`-et a megfelelő sorozatra. Feltehetjük, hogy ez a sorozat sohasem üres, ezért *nem kell* az üres sorozatra illeszkedő változatot írnunk.

```
  - fun sieve (Cons (p, nf)) = Cons(p, fn () => sieve(sift p
(nf())));
  - val primes = sieve (fromq 2);
  > val primes = Cons(2, fn):  int seq
  - takeq(25, primes);
  > val it = [2, 3, 5, 7, ..., 83, 89, 97]:  int list
```

[3] Ezt az algoritmust Park és Miller dolgozta ki 1988-ban. Helyes működéséhez a mantisszának 46 bitesnek (!) kell lennie. Az SML-megvalósítások mantisszája ennél rendszerint rövidebb. Kevésbé jó statisztikai tulajdonságú álvéletlen-számokat kaphatunk, ha `a`-nak és `m`-nek kisebb relatív prímeket választunk.

1.3.3 Numerikus számítások

A példa legyen a négyzetgyökvonás Newton-Raphson módszerrel.

```
- fun nextapprox a x = (a/x + x)/2.0;
```

`nextapprox` x_k-ból x_{k+1}-et számítja ki az $x_{k+1} = \frac{\frac{a}{x_k} + x_k}{2}$ képlet alapján. A befejeződés megállapítására egyszerű tesztet írunk:

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

A `(Cons (y, yf))` és az `xf()` sorozat ugyanaz: az `else`-ágban azért használjuk mégis az előbbit, hogy elkerüljük `xf()` költségesebb meghívását.

```
- fun qroot a = within 1E~6 (iterateq (nextapprox a) 1.0);
- qroot 5.0;
> val it = 2.236067977 :  real
- it * it;
> val it = 5.0 :  real
```

A fenti példa világosan különválasztja a *leállásvizsgálatot* (termination test) *a következő jelölt előállításától.*
A példában az *abszolút különbséget* ($|\, x - y \,| < \varepsilon$) teszteljük, de vizsgálhatnánk pl. a *relatív különbséget* ($|\, x/y - 1 \,| < \varepsilon$) vagy az
$(|\, x - y \,|)/((|\, x \,| + |\, y \,|)/2 + 1) < \varepsilon$ feltételt. A feladat többi része független attól, hogy milyen leállásvizsgálatot alkalmazunk, és így is kell megfogalmazni a megoldást. Minden egyes leállásvizsgálat tehát egy-egy függvény legyen, olyan, amely egy sorozatból egy valós számot állít elő. Egy másik függvény állítja majd elő `real seq -> real seq` típusú függvényként a következő jelöltet. Ha később integrálni, differenciálni stb. akarunk, csak a megfelelő integráló, differenciáló stb. függvényeket kell megírni, a leállásvizsgálatot végző függvényeket felhasználhatjuk.
Most tehát a következő jelölt előállítására írunk függvényt, és ezzel elrejtjük a részleteket:

```
fun approxq a =
      let fun nextapprox x = (a/x + x) / 2.0
      in iterateq nextapprox 1.0
      end;
```

Most már könnyű megírni a `qroot` függvény egy tisztább változatát:

```
val qroot = within 1E~6 o approxq;
```