AZ SML-MODULNYELV

Struktúra-deklaráció átlátszó szignatúra-kötéssel: `ORDERED`, `MyString`

- Átlátszó szignatúra-kötéssel csökkenthető az explicit típusspecifikációk száma a szignatúrában. De az se jó, ha túl sokat használjuk, ui. a típusinformációk láthatóvá tételével csökken a modulok függetlensége.
- Az átlátszó szignatúra-kötés tipikusan arra való, hogy egy struktúra *nézetét* állítsuk elő vele. E nézet célja, hogy elrejtse azokat a komponenseket, amelyek az adott szövegkörnyezetben feleslegesek, de ne rejtse el azokat a típusdefiníciókat, amelyekre szükség van.
- Az `ORDERED` szignatúra specifikálja a `t` típust és a `t` típusú értékekből álló párokra alkalmazható `lt` összehasonlító műveletet.

```
signature ORDERED =
sig
    type t
    val lt : t * t -> bool
end
```

- Az ilyen szignatúrát, mint láttuk, *csak átlátszóan* érdemes egy struktúrához kötni, különben nem tudnánk `t` típusú értékeket létrehozni.

Struktúra-deklaráció átlátszó szignatúra-kötéssel: ORDERED, MyString (folyt.)

- Nézzük a következő példát:

```
structure MyString : ORDERED =
struct
    type t = string
    val clt = Char.<
    fun lt (s, t) = ... clt ...
end
```

- `MyString` a füzérek összehasonlítását karakterek összehasonlítására vezeti vissza, `clt`-t elrejti. `String.t` a külvilág számára is ekvivalens `string`-gel, bár ez az `ORDERED` szignatúrából nem látszik: `MyString` tényleges, *látható* szignatúrája ugyanis:

```
ORDERED where type t = string
```

- Arra is érdemes átlátszó szignatúra-kötést használni, hogy *dokumentáljuk* egy típus jelentését (anélkül, hogy absztrakttá tennénk).

Struktúra-deklaráció átlátszó szignatúra-kötéssel: ORDERED, IntLt, IntDiv

- Tegyük föl, hogy egészeket kétféle alapon akarunk összehasonlítani, de nem akarjuk elrejteni, hogy egészekről van szó:
- Összehasonlítás aritmetikai alapon

```
structure IntLt : ORDERED =
struct
    type t = int
    val lt = (op <)
end
```

- Összehasonlítás oszthatóság alapján

```
structure IntDiv : ORDERED =
struct
    type t = int
    fun lt (m, n) = (n mod m = 0)
end
```

- Mind `IntLt.t`, mind `IntDiv.t` ekvivalens `int`-tel.

Átlátszóság, áttetszőség, függőség

- Az átlátszó szignatúra-kötés a típuslevezetéshez hasonlóan megkönnyíti a programozó dolgát: kevesebbet kell írnia. *De ára van:*
- átlátszó szignatúra-kötés esetén a szignatúra önmagában nem fordítható le, csak a struktúrával együtt: csak így állítható elő a struktúra tényleges, *látható* szignatúrája.
- Vagyis az összes olyan programrész, amely e struktúra látható szignatúrájára hivatkozik, *függ* e struktúra *megvalósításától!*
- Amíg az átlátszó szignatúra-kötés függőséget okoz, az áttetsző szignatúra-kötés kiküszöböli a függőséget.
- Ha egy struktúrához áttetsző módon kötjük a szignatúrát, a rá hivatkozó programrészek megbízhatnak a szignatúrában (a struktúra látható szignatúrája ui. ekvivalens az áttetsző szignatúrával).
- A megvalósítástól való függés gátolja, nehezíti a modularitást. A modularitás célja ui. az, hogy elszigetelje egymástól az egyes programrészeket, csökkentse az egyes programrészek hatását a többire. Ekkor egymástól függetlenül legyenek fejleszthetők, módosíthatók. Minél kevésbé függnek egymástól a modulok, annál könnyebben rakhatók össze a végén egyetlen rendszerré.

MODULHIERARCHIA

Modulhierarchia

- A modulok egymásba skatulyázhatók; a beágyazott struktúrát *alstruktúrának* (substructure) nevezzük.
- Egy struktúra más struktúra-deklarációkat tartalmazhat (akár átlátszó, akár áttetsző szignatúra-kötéssel).
- Egy szignatúrában egy struktúra `structure` *`strid`* `:` *`sigexp`* alakban specifikálható (szignatúráról lévén szó, itt nincs különbség átlátszó és áttetsző kötés között).
- Az alstruktúrákra a struktúrák típusellenőrzési és a kiértékelési szabályait rekurzív módon alkalmazza a fordítóprogram.
- Ebben a részben arról lesz szó, hogyan lehet alstruktúrákkal kifejezni az egyes absztrakciók egymástól való függését.

- A következő példák egy polimorf szótárat megvalósító programból valók.

Polimorf szótár `string` típusú keresési kulccsal

- Az első változatban a *keresési kulcs* `string` típusú. A szignatúra és lehetséges megvalósítása:

```
signature MY_STRING_DICT =
sig
    type 'a dict
    val empty : 'a dict
    val insert : 'a dict * string * 'a -> 'a dict
    val lookup : 'a dict * string -> 'a option
end

structure MyStringDict :> MY_STRING_DICT =
struct
    datatype 'a dict = Empty
                     | Node of 'a dict * string * 'a * 'a dict
    val empty = Empty
    fun insert (d, k, v) = ...
    fun lookup (d, k) = ...
end
```

- A hiányzó függvénydefiníciók a füzérek lexikografikus összehasonlító műveleteit használják.

Polimorf szótár `int` típusú keresési kulccsal

- A második változatban a *keresési kulcs* `int` típusú. A szignatúra és lehetséges megvalósítása:

```
signature MY_INT_DICT =
sig
    type 'a dict
    val empty : 'a dict
    val insert : 'a dict * int * 'a -> 'a dict
    val lookup : 'a dict * int -> 'a option
end

structure MyIntDict :> MY_INT_DICT =
struct
    datatype 'a dict = Empty
                     | Node of 'a dict * int * 'a * 'a dict
    val empty = Empty
    fun insert (d, k, v) = ...
    fun lookup (d, k) = ...
end
```

- A hiányzó függvénydefiníciók az egészek aritmetikai összehasonlító műveleteit használják.

Polimorf szótár *absztrakt* típusú keresési kulccsal

- A két változat, a típustól és az összehasonlító műveletektől eltekintve, azonos.
- A harmadik változatban a *keresési kulcs absztrakt* típusú. A *generikus* szignatúra és két leszármazottja (példánya, instanciája):

```
signature MY_GEN_DICT =
sig
    type key
    type 'a dict
    val empty : 'a dict
    val insert : 'a dict * key * 'a -> 'a dict
    val lookup : 'a dict * key -> 'a option
end

signature MY_STRING_DICT =
    MY_GEN_DICT where type key = string

signature MY_INT_DICT =
    MY_GEN_DICT where type key = int
```

Polimorf szótár `string` típusú keresési kulccsal (folyt.)

- A szignatúra egy megvalósítása `string` típusú kulcsokra:

```
structure MyStringDict :> MY_STRING_DICT =
struct
    type key = string
    datatype 'a dict = Empty
                     | Node of 'a dict * key * 'a * 'a dict
    val empty = Empty
    fun insert (None, k, v) = Node (Empty, k, v, Empty)
    fun lookup (Empty, _) = NONE
      | lookup (Node (dl, l, v, dr), k) =
          if k < l then      (* string comparison *)
              lookup (dl, k)
          else if k > l then (* string comparison *)
              lookup (dr, k)
          else
              SOME v
end
```

- A `MY_INT_DICT` szignatúrájú `MyIntDict` hasonlóan valósítható meg.

Polimorf szótár `int` típusú kulccsal, *oszthatóságon alapuló* összehasonlítással

```
structure MyIntDivDict :> MY_INT_DICT =
struct
    type key = int
    datatype 'a dict = Empty
                     | Node of 'a dict * key * 'a * 'a dict
    fun divides (k, l) = (l mod k = 0)
    val empty = Empty
    fun insert (None, k, v) = Node (Empty, k, v, Empty)
    fun lookup (Empty, _) = NONE
      | lookup (Node (dl, l, v, dr), k) =
          if divides (k, l) then      (* divisibility test *)
              lookup (dl, k)
          else if divides (l, k) then (* divisibility test *)
              lookup (dr, k)
          else
              SOME v
end
```

- Függetlenítsük a megvalósítást a keresési kulcs típusától és az összehasonlító műveletektől!

Egy rendezett absztrakt típus és néhány megvalósítása

- A `t` típus és két összehasonlító művelet

```
signature ORDERED =
sig
    type t
    val lt : t * t -> bool
    val eq : t * t -> bool
end
```

- Füzérek lexikografikus összehasonlítása

```
structure LexString : ORDERED =
struct
    type t = string
    val lt = (op <)
    val eq = (op =)
end
```

- Egészek aritmetikai összehasonlítása

```
structure LessInt : ORDERED =
struct
    type t = int
    val lt = (op <)
    val eq = (op =)
end
```

- Egészek oszthatóságon alapuló összehasonlítása

```
structure DivInt : ORDERED =
struct
    type t = int
    fun lt (m, n) = (n mod m = 0)
    fun eq (m, n) = lt (m, n)
                    andalso lt (n, m)
end
```

- Ezekben a példákban indokolt az átlátszó szignatúra-kötés alkalmazása.

Polimorf szótár generikus szignatúrája

- A `DICT` szignatúra „paramétere” az `ORDERED` szignatúrájú `Key` absztrakt keresési kulcs (a `DICT` szignatúra *örökli* a keresési kulcs `ORDERED` szignatúráját!):

```
signature DICT =
sig
    structure Key : ORDERED
    type 'a dict
    val empty : 'a dict
    val insert : 'a dict * Key.t * 'a -> 'a dict
    val lookup : 'a dict * Key.t -> 'a option
end
```

- A szignatúra két specializált változata segíti az absztrakciót:

```
signature STRING_DICT =
     DICT where type Key.t=string

signature INT_DICT =
    DICT where type Key.t=int
```

Polimorf szótár: a specializált szignatúra megvalósítása `string` kulccsal

```
structure StringDict :> STRING_DICT =
struct
    structure Key : ORDERED = LexString
    datatype 'a dict = Empty
                     | Node of 'a dict * Key.t * 'a * 'a dict
    val empty = Empty
    fun insert (None, k, v) = Node (Empty, k, v, Empty)
    fun lookup (Empty, _) = NONE
      | lookup (Node (dl, l, v, dr), k) =
          if Key.lt (k, l) then
              lookup (dl, k)
          else if Key.lt (l, k) then
              lookup (dr, k)
          else
              SOME v
end
```

(Félkövér szedéssel e változat és a következő két változat közötti **különbséget** emeljük ki.)

Polimorf szótár: a specializált szignatúra megvalósítása `int` kulccsal

```
structure LessIntDict :> INT_DICT =
struct
    structure Key : ORDERED = LessInt
    datatype 'a dict = Empty
                     | Node of 'a dict * Key.t * 'a * 'a dict
    val empty = Empty
    fun insert (None, k, v) = Node (Empty, k, v, Empty)
    fun lookup (Empty, _) = NONE
      | lookup (Node (dl, l, v, dr), k) =
          if Key.lt (k, l) then
              lookup (dl, k)
          else if Key.lt (l, k) then
              lookup (dr, k)
          else
              SOME v
end
```

Polimorf szótár: a specializált szignatúra megvalósítása `int` kulccsal

```
structure DivIntDict :> INT_DICT =
struct
    structure Key : ORDERED = DivInt
    datatype 'a dict = Empty
                     | Node of 'a dict * Key.t * 'a * 'a dict
    val empty = Empty
    fun insert (None, k, v) = Node (Empty, k, v, Empty)
    fun lookup (Empty, _) = NONE
      | lookup (Node (dl, l, v, dr), k) =
          if Key.lt (k, l) then
              lookup (dl, k)
          else if Key.lt (l, k) then
              lookup (dr, k)
          else
              SOME v
end
```

Később (a funktorok tárgyalásakor) látni fogjuk, hogyan írhatjuk meg e három struktúra közös generikus (azaz paraméterezhető) változatát.

LUSTA LISTÁK

Lusta lista (folyt.)

Az előző előadáson bemutattuk a lusta lista egy definícióját:

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

valamint a `head : 'a seq -> 'a` és a `tail : 'a seq -> 'a seq` függvényt.
Most további függvényeket definiálunk. `consq(x, xq)` az `x`-et berakja az `xq` sorozatba:

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

- Ha a `consq` függvényt alkalmazzuk, mondjuk, az `(x, E)` argumentumra, az SML a `consq(x, E)` kifejezést *nem lustán* értékeli ki, hiszen 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.
- A lusta kiértékelés érdekében a híváskor is a `Cons(x, fn () => E)` alakot kell használnunk, `consq(x,E)` nem jó.
- Az explicit `fn () => E` alak késlelteti a kiértékelést: *szükség szerinti hivatkozást* valósít meg.

Lusta lista (folyt.)

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

```
(* fromq : int -> int seq
*)
fun fromq k = Cons(k, fn () => fromq(k+1))
```

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

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

- Az `'a seq` típus nem egészen lusta kiértékelésű: ***egy nemüres sorozat fejét a futtatórendszer mindig feldolgozza***.

Egyszerű függvények lusta listákra

- 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.
- Első példánkban egészeket egyesével emelünk négyzetre. Amikor szükség van rá, az eredmény farka (egy függvény) alkalmazza a `squareq` függvényt az argumentum farkára.

```
(* squareq : int seq -> int seq
*)
fun squareq Nil: int seq = Nil
  | squareq (Cons (x, xf)) = Cons(x * x, fn () => squareq(xf()))
```

- Két lusta lista hasonlóan adható össze.

```
(* addq : (int seq * int seq) -> int seq
*)
fun addq (Cons (x, xf), Cons(y, yf)) =
                            Cons(x+y, fn () => addq(xf(), yf()))
  | addq _: int seq = Nil
```


Egyszerű függvények lusta listákra (folyt.)

- 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.

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

- Most érthetjük meg, hogy miért kellett a típusdefinícióban a `Nil` konstruktorállandót definiálni.

Magasabb rendű függvények lusta listákra

- A `map` lusta változata:

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

- A `filter` lusta változata:

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


Magasabb rendű függvények lusta listákra (folyt.)

- `squareq` a korábban látottnál sokkal egyszerűbben definiálható `mapq`-val:

```
val squareq = mapq (fn i => i * i)
```

- Olyan számsorozatot állítunk elő, amelyben 50-nél nagyobb, 7-esre végződő egészek vannak:

```
filterq (fn n => n mod 10 = 7) (fromq 50)
```

- Az `iterateq` függvény – a `fromq` egy általánosítása – a következő sorozatot állítja elő: $[x, f(x), f(f(x)), \cdots, f^k(x), \cdots]$.

```
(* iterateq : ('a -> 'a) -> 'a -> 'a seq
*)
fun iterateq f x = Cons(x, fn () => iterateq f (f x))
```

- `fromq`-t `iterateq`-val így definiálhatjuk:

```
(* fromq : int -> int seq
*)
val fromq = iterateq (fn i => i+1)
```

Álvéletlen számok

- Hagyományos álvéletlenszám-generátorok: olyan eljárások, amelyek egy *frissíthető változóban* tárolják a *seed* (mag) értéket – ebből állítják elő egy következő hívásnál a következő álvéletlen számot.
- Lusta listaként megvalósítva: a következő álvéletlen szám csak szükség esetén áll elő.

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


Álvéletlen számok (folyt.)

- Ha a `nextrandom`-ot 1.0 és 21474836467.0 közötti `seed`-re alkalmazzuk, ugyanebbe a tartományba eső más értéket állít elő az `a * seed mod m` művelettel. (A valós számokat a túlcsordulás elkerülésére használjuk.)
- A lusta lista előállítására `iterateq`-t `nextrandom`-ra és `seed` valós számmá alakított kezdőértékére alkalmazzuk. `mapq` gondoskodik arról, hogy a lusta listában minden értéket elosszunk `m`-mel, és így `randseq` 0.0-nál nem kisebb és 1.0-nél kisebb értékeket adjon eredményül. Látható, hogy a lusta lista a megvalósítás részleteit szépen elrejti a felhasználó elől.
- Az előállított álvéletlen-számok 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 (fn x => 10.0 * x)) (randseq 1)
```

Prímszámok előállítása *eratosztenészi szitával*

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 ...

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

```
(* sift : int -> int seq -> int seq *)
fun sift p = filterq (fn n => n mod p <> 0)
```

- A `sift` a `p` argumentum többszöröseit törli egy lusta listából.
- A `sieve`-nek már csak ismételten alkalmaznia kell `sift`-et a megfelelő lusta listára. Mivel ez a lusta lista sohasem üres, nem kell az üres lusta listára illeszkedő változatot írnunk.

```
(* sieve : int seq -> int seq *)
fun sieve Nil = Nil
  | sieve (Cons (p, nf)) = Cons(p, fn () => sieve(sift p (nf())));

takeq(sieve(fromq 2), 10)
```


Négyzetgyökvonás Newton-Raphson módszerrel

Az előadáson nem hangzott el, csak olvasmány, nem vizsgaanyag!

- `nextapprox` x_k-ból x_{k+1}-et számítja ki az $x_{k+1} = \frac{\frac{a}{x_k}+x_k}{2}$ képlet alapján.

```
(* nextapprox : real -> real -> real
*)
fun nextapprox a x = (a/x + x)/2.0
```

- A befejeződés megállapítására egyszerű tesztet írunk:

```
(* within : real -> real seq -> real
*)
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()` lusta lista ugyanaz: az `else`-ágban azért használjuk az elsőt, mert `xf()` meghívása költségesebb.

Négyzetgyökvonás Newton-Raphson módszerrel (folyt.)

- Ezzel

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

- A példában világosan különválasztjuk a leállásvizsgálatot (termination test) a következő jelölt előállításától.
- Most az abszolút különbséget ($|\ x - y\ | < \varepsilon$) teszteljük, de vizsgálhatnánk pl. a relatív különbséget ($|\ \frac{x}{y} - 1\ | < \varepsilon$) vagy az $\frac{|x-y|}{\frac{|x|+|y|}{2}+1} < \varepsilon$ feltételt.
- A feladat többi része független attól, hogy milyen leállásvizsgálatot alkalmazunk, és így is kell megfogalmazni a megoldást.

Négyzetgyökvonás Newton-Raphson módszerrel (folyt.)

- Írjunk függvényt a következő jelölt előállítására, és rejtsük el a részleteket:

```
(* approxq : real -> real seq
*)
fun approxq a =
      let (* nextapprox : real -> real
          *)
      fun nextapprox x = (a/x + x) / 2.0
      in iterateq nextapprox 1.0
      end
```

- Ezzel `qroot` egy „tisztább” változata:

```
(* qroot : real -> real
*)
val qroot = within 1E~6 o approxq
```

Keresztszorzatokból álló lista

Az előadáson nem hangzott el, csak olvasmány, nem vizsgaanyag!

- Legyen `xq` és `yq` egy-egy sorozat. Képezzünk új sorozatot az $(\texttt{x}_i, \texttt{y}_j)$ párokból, ahol $\texttt{x}_i \in \texttt{xq}$ és $\texttt{y}_j \in \texttt{yq}$!
- Először hagyományos listákra oldjuk meg a feladatot `map` és `pair` alkalmazásával.
- `xs` és `ys` egy-egy lista. Képezzünk listát az $(\texttt{x}_i, \texttt{y}_j)$ párokból, ahol $\texttt{x}_i \in \texttt{xs}$ és $\texttt{y}_j \in \texttt{ys}$!
- `map`-et, `pair`-t és `List.concat`-ot alkalmazva juthatunk el a keresett függvényhez.

```
(* pair : 'a -> 'b -> ('a * 'b)
*)
fun pair x y = (x, y)
```

- A `pair`-t a `map`-pel az `ys` lista elemeire alkalmazva olyan párokból álló listát kapunk eredményül, amelyben a párok első tagja a rögzített `x` érték, a második tagja pedig az `ys` egy-egy eleme.

```
map (pair x) ys
```


Keresztszorzatokból álló lista (folyt.)

- Hogyan érhetjük el, hogy az `x` végigfusson az `xs` lista összes elemén? Az eddig szabad `x`-et kössük le egy függvény argumentumaként:

```
fn x => map (pair x) ys
```

  majd alkalmazzuk újból a `map`-et erre a függvényre és `xs`-re:

```
map (fn x => map (pair x) ys) xs
```

- Listák listáját kapjuk eredményül, mert a belső `map` már listát adott vissza, amelynek minden eleméből újabb listát képeztünk a külső `map`-pel. `List.concat` elvégzi a szükséges simítást:

```
(* pairs : 'a list -> 'b list -> ('a * 'b) list
*)
fun pairs xs ys = List.concat (map (fn x => map (pair x) ys) xs)
```

Keresztszorzatokból álló lusta lista

- A `pairss`-hez hasonlóan állíthatjuk elő párok lusta listájának lusta listáját:

```
(* pairqq : 'a seq -> 'b seq -> ('a * 'b) seq seq
*)
fun pairqq xq yq = mapq (fn x => mapq (pair x) yq) xq
```

- Az eredmény véges része kiíratható `takeqq`-val, amely a bal felső saroktól számított első `m` sorból és `n` oszlopból álló téglalapot jeleníti meg az `xqq` lusta listából:

```
(* 'a takeqq : 'a seq seq * (int * int) -> 'a list list
*)
fun takeqq (xqq, (m, n)) = map (fn yq => takeq(yq, n)) (ta-
keq(xqq, m))
```

- Példa: olyan lusta lista, amelyben a párok első tagja az egymás után következő egészek 30-tól kezdve, második tagja pedig a prímszámok 2-től kezdve:

```
- pairqq (fromq 30) (sieve(fromq 2));
> val it = Cons (Cons ((30, 2), fn), fn): (int * int) seq seq
```


Keresztszorzatokból álló lusta lista (folyt.)

- ```
  - takeqq(pairqq (fromq 30) (sieve(fromq 2)), (3, 5));
  > val it = [[(30, 2), ...,  (30, 11)],
              [(31, 2), ...,  (31, 11)],
              [(32, 2), ...,  (32, 11)]] : (int * int) list list
  ```
- Ha ki akarjuk símítani a lusta listát, egy `List.concat`-hoz hasonló, lusta listákra alkalmazható függvénnyel nem megyünk semmire:

  ha `xq` végtelen, `appendq (xq, yq) = xq`.

  Azonban két lusta lista elemei páronként egymásba ékelhetők:

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

- `interleaveq` a rekurzív hívásban váltogatja a két lusta listát.
- ```
  - takeq(interleaveq(fromq 0, fromq 50), 10);
  > val it = [0, 50, 1, 51, 2, 52, 3, 53, 4, 54] : int list
  ```

Keresztszorzatokból álló lusta lista (folyt.)

- `enumerate`: lusta listák lusta listájából egyetlen lusta listát állít elő. Legyen a kétszeres mélységű lusta lista feje `xq` és a farka `xqf`; alkalmazzuk `enumerate`-et rekurzívan `xqf`-re, majd az eredményt ékeljük `xq`-ba:

```
(* enumerate : 'a seq seq -> 'a seq
*)
fun enumerate Nil = Nil
  | enumerate (Cons (xq, xqf)) =
                              interleaveq (xq, enumerate(xqf()))
```

- Ez a „megoldás” nem jó, mert a „ végtelen” lusta lista miatt a rekurzió nem ér véget: az SML-ben, amely alapvetően mohó kiértékelésű, a rekurzív hívást késleltetni kell. Több esetet kell megkülönböztetnünk:

```
fun enumerate Nil = Nil
  | enumerate (Cons (Nil, xqf)) = enumerate (xqf())
  | enumerate (Cons (Cons (x, xf), xqf)) =
                Cons(x, fn () =>
                            interleaveq(enumerate(xqf()),xf()))
```


Keresztszorzatokból álló lusta lista (folyt.)

- Ha a bemenő lusta lista üres, készen vagyunk. Ha nem üres, meg kell vizsgálni a lusta lista fejét: ha ez üres, akkor folytatni kell a rekurzív hívást, ha nem üres, akkor az explicit `fn () => ...` függvénydefinícióval *késleltetni kell* a rekurziót.
- Példa: pozitív egészekből álló párok egy lusta listáját!

```
- val posintqq = pairqq (fromq 1) (fromq 1);
> val posintqq = Cons (Cons ((1, 1), fn), fn):(int * int) seq seq
- takeq(enumerate posintqq, 15);
> val it = [(1,1), (2,1), (1,2), (3,1), (1,3), (2,2),
            (1,4), (4,1), (1,5), (2,3), (1,6), (3,2),
            (1,7), (2,4), (1,8)] : (int * int) list
```