LUSTA KIÉRTÉKELÉS, LUSTA LISTA

Mohó kiértékelés, lusta kiértékelés

Idézzük föl:

- *Mohó* (eager) vagy *applikatív sorrendű* (applicative order) kiértékelésnek nevezzük azt a kiértékelési sorrendet, amikor egy összetett kifejezésben először az operátort és argumentumait értékeljük ki, majd ezután alkalmazzuk az operátort az argumentumokra.
- *Lusta* (lazy), *szükség szerinti* (by need) vagy *normál sorrendű* (normal order) kiértékelésnek nevezzük azt a kiértékelési sorrendet, amikor a kiértékelést addig halogatjuk, ameddig csak lehetséges.
  - A lusta kiértékelés hatékonysága javítható azzal, hogy az azonos részkifejezéseket megjelöljük, és amikor egy részkifejezést először kiértékelünk, *az eredményét megjegyezzük*, majd további előfordulásakor a már kiszámított eredményt vesszük elő. A módszer hátránya a nyilvántartás szükségessége. Ma általában ezt nevezik *lusta* kiértékelésnek.
- Igazolható, hogy olyan kifejezések esetén, amelyek jelentésének megértésére a helyettesítési modell alkalmas, azaz amelyek a *hivatkozás tekintetében átlátszóak* (referential transparent), a kétféle kiértékelési sorrend azonos eredményt ad.

A továbbiakban az SML-nyelvvel felülről kompatíbilis Alice-nyelven mutatunk be példákat.

Lusta kifejezés és függvény az Alice-ben

- Egy kifejezés lusta kiértékelése írható elő a `lazy` kulcsszóval:

```
val zs = lazy [1,2,3,4,5,6,7,8,9];
val zs : int list = _lazy
```

- Az ilyen kifejezést lusta kifejezésnek nevezzük. Kiértékelésére csak akkor kerül sor, ha *szükség* van rá, azaz ha egy mohó műveletben argumentumként használjuk.
- Mindig mohó kiértékelésűek a következő kifejezések:
  - mintaillesztésben az illesztett érték,
  - függvényalkalmazásban a függvényérték,
  - kivétel jelzésében a kivételt alkotó érték,
  - primitív műveletben (pl. `op+`, `op=`) az operandus, amelyre a műveletnek szüksége van.
- A következő példában a mohó `hd`-t függvényt alkalmazzuk a lusta `zs`-re:

```
hd zs;
val it : int = 1
```

Lusta kifejezés és függvény (folyt.)

- Első kiértékelése után a lusta kifejezés (esetleg csak egy részkifejezésének) lusta volta megszűnik.

- Az egyszer kiszámított lusta (rész)kifejezés értéke a továbbiakban az *eltárolt* érték lesz:

```
zs;
val it : int list = [1, 2, 3, 4, 5, 6, 7, 8, 9]
```

- A `hd` és a `tl` lusta változata (a név végén a *z* a függvény lusta – lazy – voltára emlékeztet):

```
fun lazy headz (x::_) = x
       | headz [] = raise Empty;
val headz : 'a list -> 'a = _fn

fun lazy tailz (_::xs) = xs
       | tailz [] = raise Empty;
val tailz : 'a list -> 'a list = _fn
```

Lusta kifejezés és függvény (folyt.)

- Példák:

```
headz zs;
val it : int = _lazy

0 + headz zs;
val it : int = 1

zs;
val it : int list = [1, 2, 3, 4, 5, 6, 7, 8, 9]

tailz zs;
val it : int list = _lazy

[] @ tailz zs;
val it : int list = _lazy

tailz zs @ [];
val it : int list = [2, 3, 4, 5, 6, 7, 8, 9]

tailz zs @ [0, 1];
val it : int list = [2, 3, 4, 5, 6, 7, 8, 9]
```

Lusta lista

- A lusta lista olyan *nem korlátos méretű* lista, amelynek előnyős tulajdonságai mellett hátrányai, veszélyei is vannak, pl.
  - egy lusta lista *bármely részét* megjeleníthetjük, de *sohasem az egészet;*
  - két lusta lista elemeiből páronként képezhetünk egy harmadikat, de *nem számíthatjuk ki* egy lusta lista *elemeinek összegét,* nem kereshetjük meg benne *a legkisebbet,* nem fordíthatjuk meg az *elemek sorrendjét* stb.;
  - egy program befejeződése helyett csak azt igazolhatjuk, hogy az eredmény *tetszőleges véges része véges idő alatt* előáll.
- Most a `from` függvény lusta változatát definiáljuk: `fromz k` olyan lusta listát hoz létre, amelynek az elemei `fromz k`-tól kezdve egyesével növekvő számtani sorozatot alkotnak:

```
fun lazy fromz k = k :: fromz(k+1);
val fromz : int -> int list = _fn

val xs = fromz 3;
val xs : int list = _lazy
```

Lusta lista (folyt.)

- Ha például `xs` fejét kiszámíttatjuk, akkor `xs`-nek már csak a farka marad lusta kiértékelésű:

```
xs;
val xs : int list = _lazy

hd xs;
val it : int = 3

xs;
val it : int list = 3 :: _lazy
```

- További példák `headz` és `tailz` használatára:

```
headz(fromz 3);
val it : int = _lazy

headz(fromz 3) + 0;
val it : int = 3
```

Lusta lista (folyt.)

- További példák `headz` és `tailz` használatára (folyt.):

```
tailz(fromz 3);
val it : int list = _lazy

headz(tailz(fromz 3)) + 0;
val it : int = 4

[] @ tailz(fromz 3);
val it : int list = 4 :: _lazy
```

- A `tailz zs @ []` kifejezésben az eredményt az üres lista nem befolyásolja, az értelmező ezt felismeri, az eredménylista lusta marad.

```
tailz(fromz 3) @ [];
val it : int list = _lazy
```

- De vigyázzunk! Veremtúlcsorduláshoz vezet – előbb-utóbb – a következő kifejezés kiértékelése:

```
tailz(fromz 3) @ [1];
```

Lusta lista (folyt.)

- A `List.take` és `List.drop` függvényt lusta listára is alkalmazhatjuk a lusta lista egy részének kiértékelésére.

```
List.take(fromz 1, 5);
val it : int list = [1, 2, 3, 4, 5]

List.drop(fromz 1, 5);
val it : int list = _lazy

hd(List.drop(fromz 1, 5));
val it : int = 6
```

- 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.

Egyszerű függvények lusta listákra

- A következő lusta függvény egy lusta lista egész elemeinek a négyzetét számítja ki.

```
fun lazy squarez [] = []
       | squarez (x::xs) = x*x :: squarez xs;
val squarez : int list -> int list = _fn

squarez(fromz 1);
val it : int list = _lazy

List.take(squarez(fromz 1), 10);
val it : int list = [1, 4, 9, 16, 25, 36, 49, 64, 81, 100]
```

- Két lusta lista hasonlóan adható össze:

```
fun lazy addz (x::xs, y::ys) = x+y :: addz(xs, ys)
       | addz _ = [];
val addz : int list * int list -> int list = _fn

addz(fromz 1000, squarez(fromz 1));
val it : int list = _lazy

List.take(addz(fromz 1000, squarez(fromz 1)), 7);
val it : int list = [1001, 1005, 1011, 1019, 1029, 1041, 1055]
```

Egyszerű függvények lusta listákra (folyt.)

- Az `appendz` függvény addig nem nyúl `ys`-hez, amíg `xs` ki nem ürül – vagyis csak akkor nyúl hozzá, ha `xs` korlátos.

```
fun lazy appendz (x::xs, ys) = x :: appendz (xs, ys)
       | appendz ([], ys) = ys;
val appendz : 'a list * 'a list -> 'a list = _fn

appendz([1,2,3],[4,5,6]);
val it : int list = _lazy

appendz([1,2,3],[4,5,6]) @ [];
val it : int list = [1, 2, 3, 4, 5, 6]

List.take(appendz([1,2,3], fromz 10), 7);
val it : int list = [1, 2, 3, 10, 11, 12, 13]

List.take(appendz(fromz 10, [4,5,6]), 7);
val it : int list = [10, 11, 12, 13, 14, 15, 16]
```

Magasabbrendű függvények lusta listákra

- A `map` lusta változata:

```
fun lazy mapz f [] = []
      | mapz f (x::xs) = f x :: mapz f xs;
val mapz : ('a -> 'b) -> 'a list -> 'b list = _fn
```

- A `filter` lusta változata:

```
fun lazy filterz p [] = []
      | filterz p (x::xs) = if p x
                            then x :: filterz p xs
                            else filterz p xs;
val filterz : ('a -> bool) -> 'a list -> 'a list = _fn
```

- Az előző szakaszban definiált `squarez`-t egyszerű felírni `mapz`-vel:

```
val squarez = mapz (fn x => x*x);
val squarez : int list -> int list = _fn
```

- Az `iteratez` a `fromz` egy általánosítása:

```
fun lazy iteratez f x = x :: iteratez f (f x);
val iteratez : ('a -> 'a) -> 'a -> 'a list = _fn
```

Magasabbrendű függvények lusta listákra (folyt.)

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

```
val sevens = filterz (fn n => n mod 10 = 7) (fromz 50);
val sevens : int list = _lazy

List.take(sevens, 8);
val it : int list = [57, 67, 77, 87, 97, 107, 117, 127]
```

- Egy példa `iteratez` alkalmazására:

```
val zs = iteratez (fn x => x / 2.0) 1.0;
val zs : real list = _lazy

List.take(zs, 5);
val it : real list = [1.0, 0.5, 0.25, 0.125, 0.0625]
```

- `fromz`-t az `iteratez`-vel így definiálhatjuk:

```
val fromz = iteratez (fn k => k+1);
val fromz : int -> int list = _fn

List.take(fromz 3, 6);
val it : int list = [3, 4, 5, 6, 7, 8]
```

Álvéletlenszámok

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

```
local val a = 16807.0 and m = 2147483647.0
      (* nextrandom seed = a következő álvéletlenszám
         nextrandom : real -> real
      *)
      fun nextrandom seed =
            let
                val t = a * seed
            in
                t - real(floor(t/m))*m
            end
in
    fun randomz s = mapz (fn x => x/m) (iteratez nextrandom s)
end;
val randomz : real -> real list = _fn
```

Álvéletlenszámok (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 az `iteratez`-t a `nextrandom`-ra és a `seed` valós számmá alakított kezdőértékére alkalmazzuk. A `mapz` gondoskodik arról, hogy a lusta listában minden értéket elosszunk `m`-mel, és így a `randomz` 1.0-nél kisebb nemnegatív értékeket adjon eredményül. Látható, hogy a lusta lista a megvalósítás részleteit szépen elrejti a felhasználó elől.

- Az előállított álvéletlen-számok tehát 1.0-nél kisebb nemnegatív valós számok; `mapz`-val alakíthatjuk át őket 0 és 9 közötti egészekké:

```
val rs = mapz (floor o (fn x => 10.0 * x)) (randomz 1.0);
val rs : Int.int list = _lazy

List.take(rs, 9);
val it : Int.int list = [0, 0, 1, 7, 4, 5, 2, 0, 6]
```

Prímszámok előállítása *eratosztenészi szitával*

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

```
fun siftz p = filterz (fn n => n mod p <> 0);
val siftz : int -> int list -> int list = _fn
```

- A `siftz` a `p` argumentum többszöröseit törli egy lusta listából.
- A `sievez`-nek már csak ismételten alkalmaznia kell `siftz`-t a megfelelő lusta listára.

```
fun lazy sievez [] = []
       | sievez (p::ps) = p :: sievez(siftz p ps);
val sievez : int list -> int list = _fn
```

Prímszámok előállítása *eratosztenészi szitával* (folyt.)

- Mivel most a lusta lista sohasem lehet üres, nem kellene az üres lusta listára illeszkedő változatot írnunk, de nélküle az Alice arra figyelmeztetne, hogy nem fedtünk le minden esetet.

```
fun lazy sievez (p::ps) = p :: sievez(siftz p ps);
1.9-1.49: warning: match is not exhaustive, because e.g.
    nil
is not covered
val sievez : int list -> int list = _fn
```

- Példa:

```
val primes = sievez(fromz 2);
val primes : int list = _lazy

List.take(primes, 11);
val it : int list = [2, 3, 5, 7, 11, 13, 17, 19, 23, 29, 31]
```

Négyzetgyökvonás Newton-Raphson módszerrel

- `nextapprox` x_k-ból a gyök egy x_{k+1} közelítését számítja ki az $x_{k+1} = \frac{\frac{a}{x_k}+x_k}{2}$ képlettel:

```
fun nextapprox a x = (a/x+x)/2.0;
val nextapprox : real -> real -> real = _fn
```

- A befejeződés megállapítására egyszerű tesztet írunk:

```
exception Impossible;
fun within (eps : real) (x::(yys as y::ys)) =
      if abs(x-y) <= eps
      then y
      else within eps yys
  | within _ _ = raise Impossible;
val within : real -> real list -> real = _fn
```

- A második klóz azért kell, hogy ne kapjunk figyelmeztetést a lefedetlen esetek miatt: mivel a `within` függvényt lusta listára fogjuk alkalmazni, ezt a klózt az Alice sohasem értékeli ki.

Négyzetgyökvonás Newton-Raphson módszerrel (folyt.)

- Ezzel a `qroot` függvény első változata:

```
fun qroot a = within 1e~6 (iteratez (nextapprox a) 1.0);
val qroot : real -> real = _fn

qroot 5.0;
val it : real = 2.23607
```

- A programban a *leállásvizsgálatot* világosan elválasztjuk *a közelítések előállításától.*
- A leállásvizsgálat (`within`) olyan függvény, amely egy valós számból és egy valós számokat tartalmazó listából egy valós számot állít elő.
- Itt az abszolút különbséget ($|\ \mathtt{x} - \mathtt{y}\ | < \varepsilon$) teszteljük, de vizsgálhatnánk pl. a relatív különbséget ($|\ \frac{\mathtt{x}}{\mathtt{y}} - 1\ | < \varepsilon$) vagy az $\frac{|\mathtt{x}-\mathtt{y}|}{\frac{|\mathtt{x}|+|\mathtt{y}|}{2}+1} < \varepsilon$ feltételt.
- A feladat többi része független attól, hogy milyen leállásvizsgálatot alkalmazunk, és általában is így célszerű megírni a programjainkat.

Négyzetgyökvonás Newton-Raphson módszerrel (folyt.)

- A közelítések előállítását célszerű még világosabban elhatárolni a program többi részétől. `approxz` a közelítések lusta listáját állítja elő:

```
fun approxz a =
      let
          fun nextapprox x = (a/x+x)/2.0
      in
          iteratez nextapprox 1.0
      end;
val approxz : real -> real list = _fn
```

- Ezzel a `qroot` függvény egy tisztább változata:

```
val qroot = within 1e~6 o approxz;
val qroot : real -> real = _fn

qroot 5.0;
val it : real = 2.23607
```

Keresztszorzatokból álló lista

- Legyen `xs` és `ys` egy-egy lusta lista. Képezzünk új lusta listát az $(\mathtt{x}_i, \mathtt{y}_j)$ párokból, ahol $\mathtt{x}_i \in \mathtt{xs}$ és $\mathtt{y}_j \in \mathtt{ys}$!
- A nem korlátos méretű listák kezelése különleges problémákat vet fel, ezért oldjuk meg először a feladatot korlátos méretű listákkal, `map` és `pair` alkalmazásával.
- `xs` és `ys` egy-egy lista. Képezzünk listát az $(\mathtt{x}_i, \mathtt{y}_j)$ párokból, ahol $\mathtt{x}_i \in \mathtt{xs}$ és $\mathtt{y}_j \in \mathtt{ys}$!
- `map`-et, `pair`-t és `List.concat`-ot alkalmazva juthatunk el a keresett függvényhez.

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

- A `pair`-t a `map`-pel az `ys` lista elemeire alkalmazva olyan párokból álló listát kapunk, 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.

```
fun pairss xs ys = map (fn x => map (pair x) ys) xs;
val pairss : 'a list -> 'b list -> ('a * 'b) list list = _fn
```

- `List.concat` elvégzi a szükséges simítást:

```
fun pairs xs ys = List.concat(pairss xs ys);
val pairs : 'a list -> 'b list -> ('a * 'b) list = _fn
```

Keresztszorzatokból álló lusta lista

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

```
fun pairssz xs ys = mapz (fn x => mapz (pair x) ys) xs;
val pairssz : 'a list -> 'b list -> ('a * 'b) list list = _fn
```

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

```
fun takeRect (xss, (m, n)) =
      map (fn y => List.take(y, n)) (List.take(xss, m));
val takeRect : 'a list list * (int * int) -> 'a list list = _fn
```

- 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:

```
val pss = pairssz (fromz 30) (sievez(fromz 2));
val pss : (int * int) list list = _lazy

takeRect(pss, (3, 5));
val it : (int * int) list list =
   [[(30, 2), (30, 3), (30, 5), (30, 7), (30, 11)],
    [(31, 2), (31, 3), (31, 5), (31, 7), (31, 11)],
    [(32, 2), (32, 3), (32, 5), (32, 7), (32, 11)]]
```

Keresztszorzatokból álló lusta lista (folyt.)

- Mostantól a `pss` már részben ki van értékelve:

```
pss;
val it : (int * int) list list =
   ((30, 2) :: (30, 3) :: (30, 5) :: (30, 7) :: (30, 11) :: _lazy)
   ((31, 2) :: (31, 3) :: (31, 5) :: (31, 7) :: (31, 11) :: _lazy)
   ((32, 2) :: (32, 3) :: (32, 5) :: (32, 7) :: (32, 11) :: _lazy)
   _lazy
```

- Ha ki akarunk símítani egy lusta listát, a `List.concat` függvénnyel nem megyünk semmire, ugyanis `appendz (xs, ys) = xs`. Ellenben két lusta lista elemei például *páronként egymásba ékelhetők* – `interleavez` a *rekurzív hívásban* váltogatja a két lusta listát:

```
fun lazy interleavez ([], ys) = ys
       | interleavez (x::xs, ys) = x :: interleavez(ys, xs);
val interleavez : 'a list * 'a list -> 'a list = _fn
```

- Példa `interleavez` alkalmazására:

```
List.take(interleavez(fromz 0, fromz 50), 10);
val it : int list = [0, 50, 1, 51, 2, 52, 3, 53, 4, 54]
```

Keresztszorzatokból álló lusta lista (folyt.)

- `enumeratez` lusta listák lusta listájából egyetlen lusta listát állít elő. Jelöljük a kétszeres mélységű lusta lista fejét `xs`-sel és a farkát `xss`-sel; alkalmazzuk `enumeratez`-t rekurzívan `xss`-re, majd az eredményt ékeljük `xs`-be:

```
fun lazy enumeratez [] = []
       | enumeratez (xs::xss) = interleavez(xs, enumeratez xss);
val enumeratez : 'a list list -> 'a list = _fn
```

- Állítsuk elő például a pozitív egészekből álló párok egy lusta listáját!

```
val pozIntss = pairssz (fromz 1) (fromz 1);
val pozIntss : (int * int) list list = _lazy

List.take(enumeratez pozIntss, 15);
val it : (int * int) list =
   [(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)]
```

HALMAZMŰVELETEK

Halmazműveletek: „halmaz hatványhalmaza” (`powerSet`)

A hatványhalmazt előállító algoritmus vázlata:

- Az S halmaz hatványhalmaza = *összes* részhalmazának a halmaza, magát az S halmazt és a $\{\}$ üres halmazt is beleértve.
- S hatványhalmaza úgy állítható elő, hogy kivesszük S-ből az x elemet, majd *rekurzív módon* előállítjuk az $S - \{x\}$ hatványhalmazát.
- Ha tetszőleges T halmazra $T \subseteq S - \{x\}$, akkor $T \subseteq S$ és $T \cup \{x\} \subseteq S$, így mind T, mind $T \cup \{x\}$ eleme S hatványhalmazának.
- Miközben a fenti elvet rekurzív módon alkalmazzuk, tehát fölsoroltatjuk az $S - \{x\}$ stb. részhalmazait, gyűjtjük a *már kiválasztott* elemeket. Egy-egy rekurzív lépésben a gyűjtő vagy változatlan (T), vagy kiegészül az x elemmel ($T \cup \{x\}$).
- A `pws` függvényben a `base` argumentumban gyűjtjük a halmaz *már kiválasztott* elemeit; kezdetben üres.
- `pws(xs, base)` = $\{S \cup$ `base` $\mid S \subseteq$ `xs`$\}$, azaz `xs` $\cup$ `base` azon részhalmazainak a listája, amelyek teljes egészében tartalmazzák a `base` halmazt.

Halmazműveletek: „halmaz hatványhalmaza” (folyt.)

- Ezzel a `pws` függvény:

```
(* pws : 'a list * 'a list -> 'a list list
   pws(xs, base) = mindazon halmazok listája, amelyek előállnak xs egy
                   részhalmazának és a base halmaznak az uniójaként    *)
fun pws ([], base) = [base]
  | pws (x::xs, base) = pws(xs, base) @ pws(xs, x::base)
```

- `pws(xs, base)` valósítja meg az $S - \{x\}$ rekurzív hívást (hiszen `x::xs` felel meg S-nek), azaz állítja elő az összes olyan halmazt, amelyekben `x` nincs benne.
- `pws(xs, x::base)` rekurzív módon `base`-ben gyűjti az `x` elemeket, vagyis előállítja az összes olyan halmazt, amelyben `x` benne van.
- `powerSet`-nek már csak megfelelő módon hívnia kell `pws`-t:

```
(* powerSet : 'a list -> 'a list list
   powerSet xs = az xs halmaz hatványhalmaza *)
fun powerSet xs = pws(xs, [])
```

- Példa:

```
powerSet [1,2,3,4,5,6,7,8,9,10,11,12,13,14,15,16,17,19,20];
```

Halmazműveletek: „halmaz hatványhalmaza”, hatékonyabban

- `pws` rossz hatékonyságú a kétfelé ágazó rekurzió miatt; 20 egész szám hatványhalmazának előállítása már a verem túlcsordulását okozza. Írjunk valamivel hatékonyabb változatot.
- Az `insAll` segédfüggvény egy elemet szúr be egy listákból álló lista minden eleme elé

```
(* insAll : 'a * 'a list list * 'a list list -> 'a list list
   insAll(x, yss, zss) = az yss lista ys elemeinek zss elé fűzött
                         listája, amelyben minden ys elem elé x van beszúrva *)
fun insAll (x, [], zss) = zss
  | insAll (x, ys::yss, zss) = insAll(x, yss, (x::ys)::zss)
```

- `powerSet` `insAll`-t használó lineáris-rekurzív processzt előállító változata

```
fun powerSet [] = [[]]
  | powerSet (x::xs) = let val pws = powerSet xs
                       in  pws @ insAll(x, pws, []) end
```

- `powerSet` `insAll`-t használó lineáris-iteratív processzt előállító változata

```
fun powerSet [] = [[]]
  | powerSet (x::xs) = let val pws = powerSet xs
                       in insAll(x, pws, pws) end

powerSet [1,2,3,4,5,6,7,8,9,10,11,12,13,14,15,16,17,19,20,21,22];
```