1. Listák használata: rendezés

Néhány rendezőalgoritmus SML-ben: beszúró rendezés (`inssort`), gyorsrendezés (`quicksort`), összefésülő rendezés (`tmsort` és `bmsort`), simarendezés (`smsort`).

1.1. Beszúró rendezés

Az `ins` segédfüggvény az `x` elemet a megfelelő helyre rakja be az `ys` listában:

```
(* ins (x, ys) = az x értékkel a <= reláció szerint
                 bővített ys
   ins :  real * real list -> real list
   PRE: ys a <= reláció szerint rendezett
*)
fun ins (x, y::ys) =
            if x <= y then x::y::ys else y::ins(x, ys)
  | ins (x :  real, []) = [x]
```

`inssort`-tal rekurzívan rendezzük a lista maradékát; végrehajtási ideje $O(n^2)$:

```
(* inssort xs = az xs elemeinek a <= reláció szerint
                rendezett listája
   inssort :  real list -> real list
*)
fun inssort (x::xs) = ins(x, inssort xs)
  | inssort [] = []
```

1.1.1. Generikus megoldások

Ha a következő elemet a helyére rakó `f` függvényt paraméterként adjuk át, az `inssort` tetszőleges típusú adatok rendezésére használható:

```
(* inssort f xs = az xs elemeinek az f függvény
                  segítségével rendezett listája
   inssort :  ('a * 'b list -> 'b list) ->
                                      'a list -> 'b list
*)
fun inssort f (x::xs) = f(x, inssort f xs)
  | inssort _ [] = []
```

Még jobb, ha magát az `ins` függvényt tesszük generikussá:

```
(* ins cmp (x, ys) = az x értékkel a cmp reláció
szerint
                     bővített ys
   ins :  ('a * 'a -> bool) -> 'a * 'a list -> 'a list
   PRE: ys a cmp reláció szerint rendezett
*)
fun ins cmp (x, ys) =
      let infix cmp
          fun ins0 (y::ys) =
```



```
                  if x cmp y then x::y::ys else y::ins0
ys
                | ins0 [] = [x]
      in
         ins0 ys
      end
```

Ezzel `inssort` egy újabb változata:

```
(* inssort xs = az xs elemeinek a cmp reláció szerint
                rendezett listája
   inssort :  ('a * 'a -> bool) -> 'a list -> 'a list
*)
fun inssort cmp (x::xs) = ins cmp (x, inssort cmp xs)
  | inssort _ [] = []
```

`inssort` eddig bemutatott változatai előbb elemeire szedik szét a rendezendő listát, majd hátulról visszafelé haladva, rendezés közben építik fel az újat. Jobbrekurziót és gyűjtőargumentumot használó változatának kisebb veremre van szüksége, mivel a listáról leválasztott elemeket balról jobbra haladva azonnal berakja a helyükre az eredménylistában. (A két megoldás futási idejét az 1.1.3 szakaszban hasonlítjuk össze).

```
fun inssort2 cmp xs =
             (* sort xs zs = az xs már feldolgozott
                elemeinek a cmp reláció szerint
                rendezett listája zs
                sort :  'a list -> 'a list -> 'a list
             *)
      let fun sort (x::xs) zs = sort xs (ins cmp (x,
zs))
            | sort [] zs = zs
      in sort xs []
      end
```

1.1.2. Beszúró rendezés **`foldr`**-rel és **`foldl`**-lel

A második argumentumát gyűjtőargumentumként használó `foldl` sokkal kisebb vermet használ, mint `foldr`, ezért `inssort2` hosszabb listák rendezésére alkalmas. A futási időkről az 1.1.3 szakaszban lesz szó.

```
fun inssort cmp  = foldr (ins cmp) []
fun inssort2 cmp = foldl (ins cmp) []
```

1.1.3. A futási idők összehasonlítása

Véletlenszerűen előállított listák, illetve eredetileg éppen fordított sorrendű listák rendezéséhez szükséges futási időt mérünk. Véletlen eloszlású listát állít elő a `Random` könyvtárbeli `rangelist` függvény. Növekvő sorrendű egészlistát állít elő a `--` operátor:

```
infix --;
```

```
fun fm -- to =
      let fun upto to zs = if to < fm
                           then zs
                           else upto (to-1) (to::zs)
      in upto to []
      end;
```

A futási idők méréséhez 2000 elemet tartalmazó listákat állítunk elő:

```
val xs = Random.rangelist (1, 100000)
                          (2000, Random.newgen());
```

illetve

```
val xs = 1 -- 2000;
```

A futási időket az alábbi programmal mérjük meg és íratjuk ki:

```
app load ["Int", "Random", "Time", "Timer"];
let val starttime = Timer.startCPUTimer()
    val zs = inssort op>= xs
    val {usr=tim,...} = Timer.checkCPUTimer starttime
    val t1 = "Integer sort with inssort:  length = " ^
             Int.toString(length xs) ^ ", time = " ^
             Time.fmt 2 tim ^ " sec\n";
    val starttime = Timer.startCPUTimer()
    val zs = inssort2 op>= xs
    val {usr=tim,...} = Timer.checkCPUTimer starttime
    val t2 = "Integer sort with inssort2:  length = " ^

             Int.toString(length xs) ^ ", time = " ^
             Time.fmt 2 tim ^ " sec\n"
in
    print(t1 ^ t2)
end;
```

A 2000 elemet tartalmazó, eredetileg fordított sorrendű listák rendezése `inssort` fenti változataival több mint 5 s-ig, a gyűjtőargumentumos `inssort2` bemutatott változataival viszont csak 0.01-0.02 s-ig tart (linux alatt, 233 MHz-es Pentium processzorral). Hatalmas a nyereség, nem beszélve arról, hogy gyűjtőargumentum nélkül a verem hamar megtelik! De eltűnik különbség a kétféle változat között, ha ugyanolyan hosszú, de véletlenszerűen előállított listákat rendezünk: a futási idő bármelyik változat esetén kb. 2 s.

1.2. Gyorsrendezés

A gyorsrendezés a növekvő sorrendben rendezendő sorozatot három részre osztja: az m mediánra, a mediánnál nem nagyobb, ill. nagyobb elemekre:

$\leq$	m	$<$

Mediánnak az adott lépésben rendezendő lista fejét választjuk.

```
(* quicksort cmp xs = az xs elemeinek cmp szerint
rendezett
                        listája
   quicksort :  ('a * 'a -> bool) -> 'a list -> 'a list
*)
fun quicksort cmp xs =
      let (* qs :  'a list -> 'a list
          *)
          fun qs (m::ys) = partition ([], [], m, ys)
            | qs ys = ys
          (* partition :  'a list * ' list * 'a * 'a
list
                                                  -> 'a
list
          *)
          and partition (ls, rs, m, x::xs) =
                if cmp(x, m)
                then partition(x::ls, rs, m, xs)
                else partition(ls, x::rs, m, xs)
            | partition (ls, rs, m, []) =
                                          qs ls @ (m::qs
rs)
      in
          qs xs
      end
```

`partition` iteratív függvény, amely két eredménylistát épít fel, `ls`-t és `rs`-t. `quicksort` a 2000 elemet tartalmazó, eredetileg fordított sorrendű listát majdnem 14 s alatt rendezi. Az ugyanilyen hosszú, de véletlenszerűen előállított lista rendezéséhez ugyanakkor csak 0.1 s-ra van szüksége.
A `@` művelet kiküszöbölhető újabb segédargumentum bevezetésével. Az algoritmus hatékonysága tovább javítható pl. úgy, hogy az m értékű elemeket egy harmadik listában gyűjtjük, amikor a listát két részre bontjuk.

1.3. Összefésülő rendezés

Az összefésülő rendezéshez szükségünk van egy olyan függvényre, amely két listát növekvő sorrendben egyesít:

```
(* merge(xs, ys) = xs és ys elemeinek <= szerint
                   egyesített listája
   merge :  int list * int list -> int list
*)
fun merge (x::xs, y::ys) =
      if x <= y
      then x::merge(xs, y::ys)
      else y::merge(x::xs, ys)
  | merge ([], ys) = ys
  | merge (xs, []) = xs
```

vagy ún. *réteges minta* (...`as`...) alkalmazásával:

```
fun merge (xxs as x::xs, yys as y::ys)=
      if x <= y
```

```
        then x::merge(xs, yys)
        else y::merge(xxs, ys)
  ...
```

Hatékonyságromlást okoz, hogy a részeredményeket itt is a veremben tároljuk. Sajnos, ezen nem lehet segíteni, mert iteratív megoldás esetén vissza kellene fordítani az eredményül kapott listát.

1.3.1. Fölülről lefelé haladó összefésülő rendezés

A fölülről lefelé haladó (*top-down*) összefésülő rendezés akkor hatékony, ha közel azonos hosszúságú az a két lista, amelyekre a rendezendő listát szétszedjük. (A `tmsort` név elején a `t` betű utal a *top-down* rendezésre.)

```
(* tmsort xs = az xs elemeinek a <= reláció szerint
               rendezett listája
   tmsort :  int list -> int list
*)
fun tmsort xs =
       let val h = length xs
           val k = h div 2
       in
          if h > 1
          then merge(tmsort(take(xs, k)),
                     tmsort(drop(xs, k)))
          else xs
       end
```

A legrosszabb esetben $O(n \cdot log\, n)$ lépésre van szükség. A módszer egyszerű és elég gyors.

1.3.2. Alulról fölfelé haladó összefésülő rendezés

Az alulról fölfelé haladó (*bottom-up*) összefésülő rendezés legegyszerűbb változata az eredeti l hosszúságú listát l darab egyelemű listára bontja, majd a szomszédos listákat összefuttatja, így 2, 4, 8, 16 stb. elemű listákat állít elő. A megoldás egyszerű, de pazarló.
R. O'Keefe algoritmusa (1982) lépésről lépésre futtatja össze az egyforma hosszú részlistákat, de csak az utolsó lépésben rendezi az összeset. Az alábbi példában az összefuttatott részlistákat aláhúzással jelöljük:

```
A B C D E F G H I J K
A B C D E F G H I J K
A B C D E F G H I J K
A B C D E F G H I J K
A B C D E F G H I J K
A B C D E F G H I J K
A B C D E F G H I J K
A B C D E F G H I J K
```

Vagyis az algoritmus először az `A`-t futtatja össze a `B`-vel, majd a `C`-t a `D`-vel, ezt követően pedig az `AB`-t a `CD`-vel, hiszen most már ezek hossza is egyforma. Ezután `E` és `F`, `G` és `H`, `EF` és `GH`, majd `ABCD` és `EFGH` összefuttatása következik s.í.t.

`bmsort` néven definiáljuk az összefésülő rendezés alulról fölfelé haladó változatát, amely `quicksort`-tal kb. azonos idő alatt rendez (a `bmsort` név első betűje, `b` utal a *bottom-up* rendezésre):

```
(* bmsort xs = az xs elemeinek a <= reláció szerint
               rendezett listája
   bmsort :  int list -> int list
*)
fun bmsort xs = sorting(xs, [], 0)
```

`bmsort` a `sorting` segédfüggvényt használja. Ennek első argumentuma a rendezendő lista, második argumentumában a már rendezett részlistákat gyűjtjük (kezdőértéke `[]`) , harmadik argumentuma pedig az adott lépésben összefuttatandó elem sorszáma (kezdetben 0).
Ha a rendezendő lista még nem fogyott el, soron következő eleméből `sorting` egyelemű listát (`[x]`) képez, és ezt a már rendezett részlisták listája (`lss`) elé fűzve meghívja a `mergepairs` segédfüggvényt. `mergepairs`, amint látni fogjuk, az argumentumként átadott lista két azonos hosszúságú bal oldali részlistáját fűzi egybe, feltéve persze, hogy vannak ilyenek. `k` az éppen átadott elem sorszáma. Ha a rendezendő lista kiürült, `sorting` a kétszintű lista egyetlen elemét, a rendezett listát adja eredményül.

```
(* sorting(xs, lss, k) = a még rendezetlen xs lista
              elemeit berakja a k elemet tartalmazó,
              már rendezett lss listába
   sorting :  int list * int list list * int -> int
list
   PRE: k >= 0
*)
fun sorting (x::xs, lss, k) =
      sorting(xs, mergepairs([x]::lss, k+1), k+1)
  | sorting ([], lss, k) = hd(mergepairs(lss, 0))
```

`mergepairs` egyetlen listában gyűjti a már összefuttatott részlistákat. Nem a listák hosszát hasonlítja össze, hanem az éppen átadott elem `k` sorszámából dönti el, hogy mit kell csinálni a következő részlistával.

```
(* mergepairs(llss, n)= az n elemet tartalmazó, már
              rendezett lss lista első két
részlistáját,
              ha egyforma a hosszuk, összefuttatja
   mergepairs :  int list list -> int list list
   PRE: n >= 0
*)
fun mergepairs (llss as ls1::ls2::lss, n) =
      (* legalább kételemű a lista *)
      if n mod 2 = 1
      then llss
      else mergepairs(merge(ls1, ls2)::lss, n div 2)
  | mergepairs (ls, _) = ls (* egyelemű a lista *)
```

Ha `n` páratlan, `mergepairs` a listát változtatás nélkül adja vissza, ha pedig páros, akkor az `llss` lista elején álló két, egyforma hosszú listát egyetlen rendezett listává

futtatja össze. `n=0`-ra `mergepairs` az összes listák listáját olyan listává futtatja össze, amelynek egyetlen eleme maga is lista.
A függvények működését az alábbi példán követhetjük. A kezdő hívás legyen

```
bmsort [1,2,3,4,5,6,7,8,9] =
          sorting ([1,2,3,4,5,6,7,8,9], [], 0)
```

Amíg `sorting` első argumentuma a nem üres `x::xs` lista, `sorting` saját magát hívja meg. A rekurzív hívás első argumentuma a lépésenként egyre rövidülő `xs` lista, harmadik argumentuma (`k+1`) a már feldolgozott listaelemek száma, második argumentuma pedig a `mergepairs([x]::lss, k+1)` függvényalkalmazás eredménye, ahol kezdetben `lss = []`.
Az alábbi táblázatos elrendezés `mergepairs` mindkét argumentumát, valamint a rekurzív `sorting` hívás itt `j`-vel jelölt harmadik argumentumát, `k+1`-et és bináris számként `k`-t mutatja lépésről lépésre. (Ne feledjük, hogy `mergepairs`-nek listák listája az első argumentuma.) A `sorting` függvény hívja `mergepairs`-t azokban a sorokban, amelyekben a `j` új értéket vesz föl, a többi helyen `mergepairs` hívása rekurzív. A táblázat utolsó oszlopa a vonatkozó magyarázatra hivatkozik.
Vegyük észre, hogy kapcsolat van az `llss` első eleme utáni listaelemek hossza és a `k` bitjei között! Ha `k` valamelyik bitje 1, akkor (balról jobbra haladva) az `llss` megfelelő listaelemének a hossza az adott bit helyiértékével egyenlő. A 0 értékű biteknek megfelelő listaelemek „hiányoznak” `llss`-ből.

llss	n	j	k	
[[1]]	1	1	0	m1
[[2],[1]]	2	2	1	m2
[[1,2]]	1			m3
[[3],[1,2]]	3	3	10	m3
[[4],[3],[1,2]]	4	4	11	m2
[[3,4],[1,2]]	2			m2
[[1,2,3,4]]	1			m3
[[5],[1,2,3,4]]	5	5	100	m3
[[6],[5],[1,2,3,4]]	6	6	101	m2
[[5,6],[1,2,3,4]]	3			m3
[[7],[5,6],[1,2,3,4]]	7	7	110	m3
[[8],[7],[5,6],[1,2,3,4]]	8	8	111	m2
[[7,8],[5,6],[1,2,3,4]]	4			m2
[[5,6,7,8],[1,2,3,4]]	2			m2
[[1,2,3,4,5,6,7,8]]	1			m3
[[9],[1,2,3,4,5,6,7,8]]	9	9	1000	m3
[[9],[1,2,3,4,5,6,7,8]]	0	0		m4
[[1,2,3,4,5,6,7,8,9]]				

Megjegyzések.

m1: Az argumentumként átadott listának egyetlen eleme van (ez maga is egy lista), ezért az argumentumot `mergepairs` második klóza változtatás nélkül visszaadja az őt hívó `sorting`-nak.

m2: `n` páros, ez azt jelzi, hogy az argumentumként átadott lista első két eleme egyforma hosszú lista, amelyeket `merge` egyetlen rendezett listává futtat össze, majd az eredménnyel `mergepairs` első klóza meghívja saját magát.

m3: `n` páratlan, ez azt jelzi, hogy az argumentumként átadott lista első két eleme nem egyforma hosszú lista, ezért az argumentumot `mergepairs` első klóza változtatás nélkül visszaadja az őt hívó `sorting`-nak.

m4: n=0, az összes listák listáját olyan listává kell összefuttatni, amelynek egyetlen lista az eleme.

A 2000 elemből álló listákat `bmsort` kevesebb mint 0.1 s alatt rendezi mind eredetileg fordított sorrendű, mint véletlenszerűen előállított listák esetén.

1.4. Simarendezés

A simarendezés (*smooth sort*) végrehajtási ideje $O(n)$, azaz arányos az elemek számával, ha a bemeneti lista csaknem rendezve van, és a legrosszabb esetben is legfeljebb $O(n \cdot log\, n)$.
Az applikatív simarendezés algoritmusa O'Keefe alulról fölfelé haladó rendezéséhez hasonló, de nem egyelemű listákat, hanem növekvő futamokat (*run*) állít elő.
Ha a futamok száma n-től független, azaz a lista majdnem rendezve van, akkor az algoritmus végrehajtási ideje $O(n)$.

```
  (* nextrun(run, xs) = ???
     nextrun :  int list * int list -> int list * int
list
  *)
  fun nextrun (run, x::xs) =
        if x < hd run
        then (rev run, x::xs)
        else nextrun(x::run, xs)
    | nextrun (run, []) = (rev run, [])
```

`nextrun` eredménye egy pár. A pár első tagja a futam (egy növekvő számsorozat), a második tagja pedig a rendezendő lista maradéka. Mivel a futam csökkenő sorrendben bővül, kilépéskor a futamot meg kell fordítani.
`smsorting` a futamokat ismételten előállítja és összefuttatja:

```
  (* smsorting (xs, lss, k) = ???
     smsorting :  int list * int list list * int
                                        -> int list
  *)
  fun smsorting (x::xs, lss, k) =
           let val (run, tail) = nextrun([x], xs)
           in
              smsorting(tail,
                        mergepairs(run::lss, k+1), k+1)
           end
    | smsorting ([], lss, k) = hd(mergepairs(lss, 0))

  (* smsort xs = az xs elemeinek a <= reláció szerint
                 rendezett listája
```

```
    smsort :  int list -> int list
  *)
  fun smsort xs = smsorting(xs, [], 0);
```

A simarendezés egy változata `sort` néven megtalálható a `Listsort` könyvtárban. A függvény specifikációja a következő:

```
  sort ordr xs = xs elemei ordr szerint nem csökkenő
                 sorrendben (Richard O'Keefe applikatív
                 simarendezése)
  val sort :  ('a * 'a -> order) -> 'a list -> 'a list
```

Az `ordr` olyan függvény, amelynek egy pár az argumentuma, és e pár két elemének az összehasonlításával kapott `order` típusú érték az eredménye:

```
  datatype order = LESS | EQUAL | GREATER
```

Az `order` típust a `General` könyvtár deklarálja. `compare` néven többek között a `Char`, az `Int`, a `Real`, a `String`, a `Word` és a `Word8` könyvtárban található olyan függvény, amely `sort` első argumentumaként használható.
A 2000 elemből álló listákat `Listsort.sort Int.compare` kevesebb mint 0.1 s alatt rendezi mind eredetileg fordított sorrendű, mint véletlenszerűen előállított listák esetén.

2. Listák használata: polimorf halmazműveletek

A `newMem` függvény egy új elemet rak be egy listába, ha az elem még nincs benne:

```
  (* newMem(x, xs) = [x] és xs listaként ábrázolt uniója
     newMem :  ''a * ''a list -> ''a list
  *)
  fun newMem (x, xs) = if x isMem xs then xs else x::xs;
```

`newMem`, ha a sorrendtől eltekintünk, halmazt hoz létre. A `setof` függvény halmazt készít egy listából úgy, hogy kiszedi belőle az ismétlődő elemeket:

```
  (* setof xs = xs elemeinek listaként ábrázolt halmaza
     setof :  ''a list -> ''a list
  *)
  fun setof (x::xs) = newMem (x, setof xs)
    | setof [] = [];
```

A `setof` függvénynek elég rossz a hatékonysága. Szerencsésebb, ha a halmazokat a megszokott halmazműveletekkel kezeljük. Most továbbra is egyszerű listaként ábrázoljuk őket, de később valamilyen hatékonyabb tárolást választhatunk, pl. rendezett listát vagy bináris fát.
Öt halmazműveletet definiálunk: unió (`union`, $S \bigcup T$), metszet (`inter`, $S \bigcap T$), részhalmaza-e (`isSubset`, $T \subseteq S$), egyenlők-e (`isSetEq`, $S = T$), hatványhalmaz (`powerset`, pS).

```
  (* union(xs, ys) = az xs és ys elemeiből álló halmazok
                     uniója
     union :  ''a list * ''a list -> ''a list
  *)
```



```
  fun union (x::xs, ys) = newMem(x, union(xs, ys))
    | union ([], ys) = ys;

  (* inter(xs, ys) = az xs és ys elemeiből álló halmazok
                     metszete
     inter :  ''a list * ''a list -> ''a list
  *)
  fun inter (x::xs, ys) = if x isMem ys
                          then x::inter(xs, ys)
                          else inter(xs, ys)
    | inter ([], _) = [];

  (* isSubset (xs, ys) = az xs elemeiből álló halmaz
                         részhalmaza-e
                         az ys elemeiből álló halmaznak
     isSubset :  ''a list * ''a list -> bool
  *)
  fun isSubset (x::xs, ys) =
        (x isMem ys) andalso isSubset(xs, ys)
    | isSubset ([], _) = true;
  infix isSubset;
```

A listák egyenlőségvizsgálata belső művelet az SML-ben. Halmazokra mégsem használható, mert pl. a `[3, 4]` és a `[4, 3, 4]` listák ugyan különböznek, de mint halmazok egyenlőek. Halmazként egyenlő pl. `[3, 4]` és `[4, 3]` is.

```
  (* isSetEq(xs, ys) = az xs és ys elemeiből álló
halmazok
                       egyenlők-e
     isSetEq :  ''a list * ''a list -> bool
  *)
  fun isSetEq (xs, ys) =
        (xs isSubset ys) andalso (ys isSubset xs);
```

A hatványhalmaz egy halmaz *összes* részhalmazának a halmaza, az eredeti halmazt és az üres halmazt is beleértve. Jelöljük S-sel az eredeti halmazt. S hatványhalmazát úgy állíthatjuk elő, hogy S-ből kiveszünk egy x elemet, és azután *rekurzív módon* előállítjuk az $S - \{x\}$ hatványhalmazát. Ha tetszőleges T halmazra $T \subseteq S - \{x\}$, akkor $T \subseteq S$ és $T \bigcup \{x\} \subseteq S$, így mind T, mind $T \bigcup \{x\}$ eleme S hatványhalmazának. A `pws` függvényben a `base` argumentum gyűjti a hatványhalmaz elemeit; kezdetben üresnek kell lennie.

```
  (* pws(xs, base) = az xs halmaz hatványhalmazának és
                     a base halmaznak az uniója
     pws :  'a list * 'a list -> 'a list list
  *)
  fun pws (x::xs, base) = pws(xs, base) @ pws(xs,
x::base)
    | pws ([], base) = [base];
```

A `pws(xs, base) @ pws(xs, x::base)` kifejezésben `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)` pedig ugyancsak rekurzív módon `base`-ben gyűjti az `x` elemeket, vagyis előállítja az összes olyan halmazt, amelyben `x` benne van. Halmazegyenlettel `pws` eredménye így adható meg:
$\mathtt{pws}(S, B) = \{T \bigcup B | T \subseteq S\}$

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

Az algoritmus helyességét nem könnyű belátni. A ún. *magasabb rendű* függvények tárgyalásakor majd könnyebben igazolható megoldást mutatunk be.

3. További rekurzív függvények

Egy függvény n-edik hatványát így specifikálhatjuk: $f^n = f(\cdots f(f(x))\cdots)$, ha $n \geq 0$ (f n-szer ismétlődik). Definiáljunk SML-függvényt ilyen ismétlések felírására!

```
(* repeat f n x = f n-edik hatványa az x helyen
   repeat :  ('a -> 'a) -> int -> 'a-> 'a
*)
fun repeat f n x =
        if n > 0 then repeat f (n-1) (f x) else x
```

Sok függvény felírható `repeat` segítségével. Példák:

```
(* drop(xs, k) = xs első k elemének elhagyásával
                 előálló lista
   drop :  'a list * int -> 'a list
*)
fun drop (xs, k) = repeat tl k xs

(* replist k = füzér, amelyben "Ha!" k-szor ismétlődik
   replist :  int -> string list
*)
fun replist k = repeat (secl "Ha!" op::)  k []
```

4. A case-kifejezés

Szintaxisa a következő:

```
case E of P1 => E1 | P2 => E2 | ··· | Pn => En
```

Az SML-értelmező – balról jobbra és fölülről lefelé haladva – megpróbálja `E`-t `P1`-re illeszteni, ha nem sikerül, `P2`-re s.í.t. A `case`-kifejezés eredménye az `E` kifejezésre illeszkedő első `Pi` mintához tartozó `Ei` kifejezés lesz . Például a `lady` függvényt így is definiálhattuk volna:

```
(* lady p = p főnemes hitvesének rangja
   lady :  degree -> string
*)
```



```
fun lady p =
  case p of Duke     => "Duchess "
          | Marquis  => "Marchioness"
          | Earl     => "Countess"
          | Viscount => "Viscountess"
          | Baron    => "Baroness"
```

Az a lehetőség tehát, hogy a `fun` függvénydefinícióban változatokat definiálhatunk, nem egyéb, mint rövidítés, *szintaktikai édesítőszer*.