LISTÁK RENDEZÉSE

Listák rendezése (folyt.)

- `inssort` (beszúró rendezés),
- **`selsort` (kiválasztó rendezés),**
- `quicksort` (gyorsrendezés),
- `tmsort` (fölülről lefelé haladó összefésülő rendezés),
- `bmsort` (alulról fölfelé haladó összefésülő rendezés),
- `smsort` (simarendezés).

Kiválasztó rendezés

```
(* selsort : ('a * 'a -> order) -> 'a list -> 'a list
   selsort cmp xs = az xs elemei cmp szerint növekvő sorrendben
*)
fun selsort cmp xs =
      let
          (* max : 'a * 'a -> 'a
             max (x, y) = x és y közül cmp szerint a nagyobb
          *)
          fun max (x, y) = if cmp(x, y) = GREATER then x else y

          (* min : 'a * 'a -> 'a
             min (x, y) = x és y közül cmp szerint a kisebb
          *)
          fun min (x, y) = if cmp(x, y) = LESS then x else y

          (* maxSelect : 'a * 'a list * 'a list -> 'a * 'a list
             maxSelect (x, ys, zs) = (m, ms), ahol m az (x::ys) lista
                cmp szerinti legnagyobb eleme, ms pedig az x::ys többi
                eleméből és a zs elemeiből álló lista
          *)
          fun maxSelect (x, [], zs) = (x, zs)
            | maxSelect (x, y::ys, zs) =
                            maxSelect(max(x,y), ys, min(x,y)::zs)
```


Kiválasztó rendezés (folyt.)

```
          (* sSort : 'a list * 'a list -> 'a list
             sSort (xs, ws) = az xs elemei cmp szerint növekvő
                              sorrendben a ws elé fűzve
          *)
          fun sSort ([], ws) = ws
            | sSort (x::xs, ws) =
                let
                    val (m, ms) = maxSelect(x, xs, [])
                in
                    sSort (ms, m::ws)
                end
      in
          sSort (xs, [])
      end;

app load ["Int","Char","Real"];

selsort Int.compare [1,2,3,4,5,6,7,8,9];
selsort Int.compare [9,8,7,6,5,4,3,2,1];
selsort Real.compare [4.5,6.7,3.6,4.3,1.2,0.9,8.9,9.8,2.0];
selsort Char.compare (explode "Ej mi a ko tyukanyo");
```

Listák rendezése (folyt.)

- `inssort` (beszúró rendezés),
- `selsort` (kiválasztó rendezés),
- **`quicksort` (gyorsrendezés),**
- `tmsort` (fölülről lefelé haladó összefésülő rendezés),
- `bmsort` (alulról fölfelé haladó összefésülő rendezés),
- `smsort` (simarendezés).

Quicksort algoritmusa:

qs [] = []

qs(m::xs) = qs([...,x_i,...] | x_i <m) @ [m] @ qs([...,x_j,...] | x_j ≥m), ahol x_i, x_j ∈ xs

Ismeretes, hogy Quicksort egy n elemű sorozatot $O(n \cdot log\, n)$ lépésben rendez.

Gyorsrendezés akkumulátor nélkül

```
(* quicksort1 cmp xs = az xs elemeinek cmp szerint rendezett listája
   quicksort1 : ('a * 'a -> order) -> 'a list -> 'a list
     qs ys = ys elemeinek cmp szerint rendezett listája
     qs : 'a list -> 'a list
       partition (xs, ls, rs) = olyan pár, amelynek
          első tagja az xs m-nél kisebb elemeinek a listája ls elé fűzve,
          második tagja pedig az xs többi eleme rs elé fűzve
       partition : 'a list * 'a list * 'a list -> 'a list                *)
fun quicksort1 cmp xs =
      let
          fun qs (m::ys) =
                let
                    fun partition (x::xs, ls, rs) =
                          if cmp(x, m) = LESS then partition(xs, x::ls, rs)
                                              else partition(xs, ls, x::rs)
                      | partition ([], ls, rs) = qs ls @ (m::qs rs)
                in
                    partition (ys, [], [])
                end
            | qs [] = []
      in
          qs xs
      end;
```


Gyorsrendezés akkumulátorral

```
(* quicksort2 cmp xs = az xs elemeinek cmp szerint rendezett listája
   quicksort2 : ('a * 'a -> order) -> 'a list -> 'a list
     qs ys zs = ys elemeinek cmp szerint rendezett listája zs elé fűzve
     qs : 'a list -> 'a list -> 'a list
       partition (xs, ls, rs) = olyan pár, amelynek
          első tagja az xs m-nél kisebb elemeinek a listája ls elé fűzve,
          második tagja pedig az xs többi eleme rs elé fűzve
       partition : 'a list * 'a list * 'a list -> 'a list                *)
fun quicksort2 cmp xs =
      let
          fun qs (m::ys) zs =
                let
                   fun partition (x::xs, ls, rs) =
                          if cmp(x, m) = LESS then partition(xs, x::ls, rs)
                                              else partition(xs, ls, x::rs)
                      | partition ([], ls, rs) = qs ls (m :: qs rs zs)
                in
                    partition (ys, [], [])
                end
            | qs [] zs = zs
      in
          qs xs []
      end;
```


Gyorsrendezés akkumulátorral, `List.partition`-nal

- Egy lista két részre bontására a `List.partition` is függvény használható.
- `List.partition : ('a -> bool ) -> 'a list -> ('a list * 'a list)`

```
(* quicksort3 cmp xs = az xs elemeinek cmp szerint rendezett listája
   quicksort3 : ('a * 'a -> order) -> 'a list -> 'a list
     qs ys zs = ys elemeinek cmp szerint rendezett listája zs elé fűzve
     qs : 'a list -> 'a list -> 'a list
*)
fun quicksort3 cmp xs =
      let
          fun qs (m::ys) zs =
                let
                    val (ls,rs) = List.partition (fn x => cmp(x,m)=LESS) ys
                in
                    qs ls (m :: qs rs zs)
                end
            | qs [] zs = zs
      in
          qs xs []
      end;
```

A futási idők mérése és összehasonlítása

Ha mosml használunk, be kell tölteni a szükséges programkönyvtárakat:

```
app load ["Listsort","Int","Random","Timer","Time"];
```

A futási idő mérésére és kiíratására `futIdo` alábbi változatát használjuk:

```
(* futIdo (sort, sortFn, cmpFn) (xs, kind) = megméri és kiírja a képernyőre
             sort xs futási idejét a függvény, a reláció, és az xs lista
             jellemzőivel együtt
   futIdo : ('a list -> 'b) * string * string -> 'a list * string -> unit
*)
fun futIdo (sort, sortFn, cmpFn) (xs, kind) =
      let val starttime = Timer.startCPUTimer()
          val zs = sort xs
          val {usr=tim,...} = Timer.checkCPUTimer starttime
      in
          print ("Int sort with " ^ sortFn ^ ", " ^ cmpFn ^
                 ", length = " ^ Int.toString(length xs) ^ " (" ^
                 kind ^ "), time = " ^ Time.fmt 2 tim ^ "s\n")
      end;
```

Megjegyzés: `sort` néven egy n elemű listát legfeljebb $O(n \cdot log\, n)$ lépésben rendező függvényt találunk az mosml `Listsort`, ill. az smlnj `ListMergeSort` könyvtárában.

A futási idők mérése és összehasonlítása (folyt.)

10000 elemet tartalmazó rendezetlen egészlistával mérjük a futási időket:

```
val xs10000R = Random.rangelist (1, 100000) (10000, Random.newgen());

  (* kb. 25 M összehasonlítás! *)
futIdo (inssort2 op>=,"inssort2","op>=") (xs10000R,"rand");

  (* kb. 130 E összehasonlítás *)
futIdo (quicksort1 Int.compare,"quicksort1","Int.compare") (xs10000R,"rand");
futIdo (quicksort2 Int.compare,"quicksort2","Int.compare") (xs10000R,"rand");
futIdo (quicksort3 Int.compare,"quicksort3","Int.compare") (xs10000R,"rand");

  (* kb. 130 E összehasonlítás *)
futIdo (Listsort.sort Int.compare,"Listsort.sort","Int.compare")
                                                          (xs10000R,"rand");
```

Az mosml válaszai:

```
Int sort with inssort2, op>=, length = 10000 (random), time = 9.49s

Int sort with quicksort1, Int.compare, length = 10000 (rand), time = 0.10s
Int sort with quicksort2, Int.compare, length = 10000 (rand), time = 0.08s
Int sort with quicksort3, Int.compare, length = 10000 (rand), time = 0.09s

Int sort with Listsort.sort, Int.compare, length = 10000 (rand), time = 0.08s
```


A futási idők mérése és összehasonlítása (folyt.)

Vizsgáljuk meg mosml alatt – `inssort2` kivételével – a többi rendező algoritmus viselkedését 200000 elemet tartalmazó rendezetlen egészlistára is:

```
val xs200000R = Random.rangelist (1, 100000) (200000, Random.newgen());

futIdo (quicksort1 Int.compare,"quicksort1","Int.compare") (xs200000R,"rand");
futIdo (quicksort2 Int.compare,"quicksort2","Int.compare") (xs200000R,"rand");
futIdo (quicksort3 Int.compare,"quicksort3","Int.compare") (xs200000R,"rand");
```

Az mosml válaszai:

```
Int sort with quicksort1, Int.compare, length = 200000 (rand), time = 3.44s
Int sort with quicksort2, Int.compare, length = 200000 (rand), time = 2.37s
Int sort with quicksort3, Int.compare, length = 200000 (rand), time = 3.29s
```

Vajon mit kezd `Listsort.sort` egy 200000 elemből álló listával?

```
futIdo (Listsort.sort Int.compare,"Listsort.sort","Int.compare")
                                                          (xs200000R,"rand");
```

Az mosml válasza:

```
! Uncaught exception:
! Out_of_memory
```

Ejnye!

Listák rendezése (folyt.)

- `inssort` (beszúró rendezés),
- `selsort` (kiválasztó rendezés),
- `quicksort` (gyorsrendezés),
- **`tmsort` (fölülről lefelé haladó összefésülő rendezés),**
- **`bmsort` (alulról fölfelé haladó összefésülő rendezés),**
- `smsort` (simarendezés).

Összefésülő rendezések

- Ebben a részben csak egészeket tartalmazó listák rendezésével foglalkozunk.
- A függvényeket generikussá tehetjük, ha az összehasonlító műveletet paraméterként adjuk át.
- Az összefésülő rendezéshez kell egy olyan függvény, amely két listát növekvő sorrendben egyesít.

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

- Korlátot jelent, hogy a részeredményeket a veremben tároljuk.
- Akkumulátor használata esetén meg kell fordítani az eredménylistát vagy az összefésülendő listákat.

Fölülről lefelé haladó összefésülő rendezés

- A fölülről lefelé haladó összefésülő rendezés *(top-down merge sort)* akkor hatékony, ha közel azonos hosszúságú az a két lista, amelyekre a rendezendő listát szétszedjük.

```
(* 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(List.take(xs, k)),
                              tmsort(List.drop(xs, k)))
                   else xs
                end;
```

- A legrosszabb esetben $O(n \cdot log\, n)$ lépésre van szükség.

Alulról fölfelé haladó összefésülő rendezés

- Az alulról fölfelé haladó összefésülő rendezés *(bottom-up merge sort)* legegyszerűbb változata az eredeti k hosszúságú listát k darab egyelemű listára bontja, majd a szomszédos listákat összefuttatja, így 2, 4, 8, 16 stb. elemű listákat állít elő.
- 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 *egymás mellé írással* jelöljük:

```
A B C D E F G H I J K
AB  C D E F G H I J K
AB  CD  E F G H I J K
ABCD    E F G H I J K
ABCD    EF  G H I J K
ABCD    EF  GH  I J K
ABCD    EFGH    I J K
ABCDEFGH        I J K
ABCDEFGH        IJ  K
...
```


Alulról fölfelé haladó összefésülő rendezés (folyt.)

- `bmsort` a `sorting` segédfüggvényt használja. A `sorting` függvény
  - első argumentuma a rendezendő lista,
  - második argumentuma a már rendezett részlistákból álló lista akkumulátora,
  - harmadik argumentuma az adott lépésben feldolgozandó elem sorszáma.

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

Alulról fölfelé haladó összefésülő rendezés (folyt.)

- Ha a rendezendő lista (`xs`) 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.
- 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 rendezett részlisták összesen
                          már k elemet tartalmazó lss listájába
     PRE: k >= 0
   sorting : int list * int list list * int -> int list
*)
and sorting (x::xs, lss, k) =
             sorting(xs, mergepairs([x]::lss, k+1), k+1)
  | sorting ([], lss, k) = hd(mergepairs(lss, 0))
```

- `mergepairs` az argumentumként átadott lista két azonos hosszúságú bal oldali részlistáját fűzi egybe, feltéve persze, hogy vannak ilyenek. `k` az átadott elem sorszáma.

Alulról fölfelé haladó összefésülő rendezés (folyt.)

- `mergepairs` egyetlen listában gyűjti a már összefuttatott részlistákat. Az éppen átadott elem `k` sorszámából dönti el, hogy mit kell csinálnia a következő részlistával.

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

- Ha `k` páratlan, `mergepairs` a listát változtatás nélkül adja vissza, ha páros, akkor az `llss` lista elején álló két, egyforma hosszú listát egyetlen rendezett listává futtatja össze.
- `k=0`-ra `mergepairs` az összes listák listáját olyan listává futtatja össze, amelynek egyetlen eleme maga is lista.

Alulról fölfelé haladó összefésülő rendezés (folyt.)

- A legrosszabb esetben $O(n \cdot log\, n)$ lépésre van szükség.
- A függvények működését egy példán is bemutatjuk. A kezdőhívás 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,
  - második argumentuma a `mergepairs([x]::lss, k+1)` függvényalkalmazás eredménye, ahol kezdetben `lss = []`,
  - harmadik argumentuma (`k+1`) a már feldolgozott listaelemek száma.

```
fun sorting (x::xs, lss, k) =
            sorting(xs, mergepairs([x]::lss, k+1), k+1)
  | sorting ([], lss, k) = hd(mergepairs(lss, 0))
```


Alulról fölfelé haladó összefésülő rendezés (folyt.)

- A következő táblázatos elrendezés
  - `mergepairs` mindkét argumentumát,
  - a rekurzív `sorting` hívás itt `j`-vel jelölt 3. argumentumát, `k+1`-et, és
  - bináris számként `k`-t mutatja lépésről lépésre.
- A `sorting` függvény hívja `mergepairs`-t azokban a sorokban, amelyekben a `j` új értéket vesz föl, a többi helyen `mergepairs` hívása rekurzív.
- Ne feledjük, hogy `mergepairs`-nek listák listája az első argumentuma!
- 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.

```
fun sorting (x::xs, lss, k) =
            sorting(xs, mergepairs([x]::lss, k+1), k+1)
  | sorting ([], lss, k) = hd(mergepairs(lss, 0))
```

Alulról fölfelé haladó összefésülő rendezés (folyt.)

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]]				

```
fun sorting (x::xs, lss, k) =
      sorting(
              xs,
              mergepairs([x]::lss, k+1),
              k+1
              )
  | sorting ([], lss, k) =
      hd(mergepairs(lss, 0))
```

m1: Az argumentumként átadott listának egyetlen eleme van (maga is 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.

Listák rendezése (folyt.)

- `inssort` (beszúró rendezés),
- `selsort` (kiválasztó rendezés),
- `quicksort` (gyorsrendezés),
- `tmsort` (fölülről lefelé haladó összefésülő rendezés),
- `bmsort` (alulról fölfelé haladó összefésülő rendezés),
- **`smsort` (simarendezés).**

Simarendezés

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

```
(* nextrun (run, xs) = (rs, ms), ahol rs az xs egy növekvő
                       futama, ms pedig pedig az xs maradéka
   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, amelynek első tagja a futam (egy növekvő számsorozat), a második tagja pedig a rendezendő lista maradéka.
- A futam csökkenő sorrendben bővül, kilépéskor a futamot meg kell fordítani.

Simarendezés (folyt.)

- `smsorting` a futamokat ismételten előállítja és összefuttatja:

```
(* smsorting (xs, lss, k) = a még rendezetlen xs lista elemeit
                            berakja a rendezett részlisták összesen
                            már k elemet tartalmazó lss listájába
     PRE: k >= 0
   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 <= szerint rendezett listája
   smsort : int list -> int list
*)
fun smsort xs = smsorting(xs, [], 0)
```

- A simarendezés egy változata `sort` néven található meg az mosml `Listsort` könyvtárában.

Futási idők összehasonlítása smlnj alatt

- Az eddig használt `Random` struktúra az mosml alatt működik.
- Most az smlnj értelmezővel hasonlítjuk össze a gyorsrendezés, az alulról fölfelé haladó összefésülő rendezés és a simarendezés bemutatott változatainak, valamint az smlnj-ben megvalósított `ListMergeSort.sort` függvénynek a futási idejét.
- Az smlnj `Random` struktúrája többek között a `Random.rand (i,j)` és `Random.randInt rand` függvényeket definiálja (ld. `<http://www.smlnj.org/doc/smlnj-lib/Manual/random.html>`).
- smlnj alatt 800000 véletlenszámból álló egészlistát állítunk elő a mérésekhez:

```
(*
local
      val rand = Random.rand(57,87641)
      fun rg 0 zs = zs
        | rg i zs = rg (i-1) (Random.randInt rand :: zs)
      fun rgen n = rg n []
in
   val xs800000R = rgen 800000
end
*)
```


Futási idők összehasonlítása smlnj alatt (folyt.)

A mérések:

```
(*
futIdo (quicksort1 Int.compare,"quicksort1","Int.compare") (xs800000R,"rand");
futIdo (quicksort2 Int.compare,"quicksort2","Int.compare") (xs800000R,"rand");
futIdo (quicksort3 Int.compare,"quicksort3","Int.compare") (xs800000R,"rand");
futIdo (bmsort,"bmsort","built-in op<=") (xs800000R,"rand");
futIdo (smsort,"smsort","built-in op<=") (xs800000R,"rand");
futIdo (ListMergeSort.sort op>=,"ListMergeSort.sort","op>=") (xs800000R,"rand");
*)
```

Az smlnj válaszai:

```
Int sort with quicksort1, Int.compare, length = 800000 (rand), time = 5.61s
Int sort with quicksort2, Int.compare, length = 800000 (rand), time = 2.80s
Int sort with quicksort3, Int.compare, length = 800000 (rand), time = 5.26s
Int sort with bmsort, built-in op<=, length = 800000 (rand), time = 5.19s
Int sort with smsort, built-in op<=, length = 800000 (rand), time = 5.45s
Int sort with ListMergeSort.sort, op>=, length = 800000 (rand), time = 5.70s
```

Feltűnő, hogy `quicksort2` kétszer gyorsabb `quicksort1`-nél: ennek oka feltehetően az, hogy `quicksort1`-ben a `@`-ot, `quicksort2`-ben pedig a `::`-ot alkalmazzuk. `quicksort3` lassúságának oka minden bizonnyal a `List.partition` alkalmazásában keresendő.

Listák rendezése (folyt.)

- `inssort` (beszúró rendezés),
- `selsort` (kiválasztó rendezés),
- `quicksort` (gyorsrendezés),
- `tmsort` (fölülről lefelé haladó összefésülő rendezés),
- `bmsort` (alulról fölfelé haladó összefésülő rendezés),
- `smsort` (simarendezés),
- **`bmsort` (alulról fölfelé haladó összefésülő rendezés), generikus változat**
- **`smsort` (simarendezés), generikus változat.**

`bmsort` generikus változata: `genbmsort`

A következőkben megírjuk `bmsort`, majd `smsort` generikus változatát, azután pedig összehasonlítjuk a futási idejüket a nemgenerikus változatokkal.

```
(* genbmsort cmp xs = az xs elemeinek cmp szerint rendezett listája
   genbmsort : ('a * 'a -> order) -> 'a list -> 'a list

   merge (xs, ys) = xs és ys elemeinek cmp szerint egyesített listája
     PRE: xs, ys cmp szerint rendezve vannak
   merge : 'a list * 'a list -> 'a list

   mergepairs (llss, n) = az n elemet tartalmazó, már rendezett llss lista
              első két részlistáját, ha egyforma a hosszuk, összefuttatja
     PRE: n >= 0
   mergepairs : 'a list list * int -> 'a list list

   sorting (xs, lss, k) = a még rendezetlen xs lista elemeit berakja a
       rendezett részlisták összesen már k elemet tartalmazó lss listájába
     PRE: k >= 0
   sorting : 'a list * 'a list list * int -> 'a list
*)
```

bmsort generikus változata: genbmsort (folyt.)

```
fun genbmsort cmp xs =
      let
          fun merge (xxs as x::xs, yys as y::ys) =
                if cmp(y, x) = GREATER then x :: merge (xs, yys)
                                       else y :: merge (xxs, ys)
            | merge ([], ys) = ys
            | merge (xs, []) = xs

          fun mergepairs (llss as ls1::ls2::lss, n) = (* llss min. kételemű *)
                if n mod 2 = 1 then llss
                else mergepairs (merge (ls1, ls2) :: lss, n div 2)
            | mergepairs (lss, _) = lss (* lss egyelemű *)

          fun sorting (x::xs, lss, k) =
                      sorting (xs, mergepairs ([x]::lss, k+1), k+1)
            | sorting ([], lss, k) = hd(mergepairs (lss, 0))
      in
          sorting (xs, [], 0)
      end;
```


smsort generikus változata: gensmsort

```
(* gensmsort cmp xs = az xs elemeinek cmp szerint rendezett listája
   gensmsort : ('a * 'a -> order) -> 'a list -> 'a list

   merge (xs, ys) = xs és ys elemeinek cmp szerint egyesített listája
     PRE: xs, ys cmp szerint rendezve vannak
   merge : 'a list * 'a list -> 'a list

   mergepairs (llss, n) = az n elemet tartalmazó, már rendezett llss lista
                 első két részlistáját, ha egyforma a hosszuk, összefuttatja
     PRE: n >= 0
   mergepairs : 'a list list * int -> 'a list list

   nextrun (run, xs) = olyan pár, amelynek első tagja xs egy cmp szerint
                   növekvő sorrendű futama, második tagja pedig xs maradéka
   nextrun : 'a list * 'a list -> 'a list * 'a list

   smsorting (xs, lss, k) = a még rendezetlen xs lista elemeit berakja a
       rendezett részlisták összesen már k elemet tartalmazó lss listájába
     PRE: k >= 0
   smsorting : 'a list * 'a list list * int -> 'a list
*)
```


smsort generikus változata: gensmsort (folyt.)

```
fun gensmsort cmp xs =
      let fun merge (xxs as x::xs, yys as y::ys)=
                if cmp(y, x) = GREATER then x :: merge (xs, yys)
                                       else y :: merge (xxs, ys)
            | merge ([], ys) = ys
            | merge (xs, []) = xs

          fun mergepairs (llss as ls1::ls2::lss, n) = (* llss min. kételemű *)
                if n mod 2 = 1 then llss
                else mergepairs (merge (ls1, ls2) :: lss, n div 2)
            | mergepairs (lss, _) = lss (* lss egyelemű *)

          fun nextrun (run, x::xs) =
                if cmp(x, hd run) = LESS then (rev run, x::xs)
                else nextrun (x::run, xs)
            | nextrun (run, []) = (rev run, [])

          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))
      in smsorting (xs, [], 0) end;
```


Futási idők összehasonlítása smlnj alatt (folyt.)

Az smlnj alatt összehasonlítjuk az alulról fölfelé haladó összefésülő rendezés és a simarendezés nemgenerikus és generikus változatainak futási idejét:

```
(*
futIdo (bmsort,"bmsort","built-in op<=") (xs800000R,"rand");
futIdo (genbmsort Int.compare,"genbmsort","Int.compare") (xs800000R,"rand");

futIdo (smsort,"smsort","built-in op<=") (xs800000R,"rand");
futIdo (gensmsort Int.compare,"gensmsort","Int.compare") (xs800000R,"rand");
*)
```

Az smlnj válaszai:

```
Int sort with bmsort, built-in op<=, length = 800000 (rand), time = 5.31s
Int sort with genbmsort, Int.compare, length = 800000 (rand), time = 5.81s

Int sort with smsort, built-in op<=, length = 800000 (rand), time = 4.88s
Int sort with gensmsort, Int.compare, length = 800000 (rand), time = 5.65s
```

A generikus változatok azért lassúbbak, mert az értelmezők az `Int.compare` relációt lassabban értékelik ki a `op<=` relációnál.
Az alulról fölfelé haladó összefésülő rendezés és különösen a simarendezés akkor hatékonyabb a gyorsrendezésnél, ha a rendezendő lista – több része – már kezdetben rendezve van. A vizsgálatokat meghagyjuk gyakorló feladatnak.

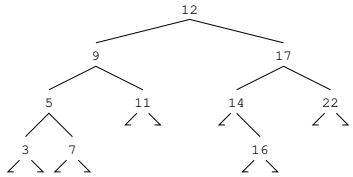
BINÁRIS FÁK

Bináris fák `datatype` deklarációval

- A listához hasonlóan rekurzív adatszerkezet a *fa*.
- Először olyan bináris fát deklarálunk, amelynek a levelei üresek, a csomópontjaiban pedig előbb a bal részfát, majd az `'a` típusú értéket, és végül a jobb részfát adjuk meg:

```
datatype 'a tree = L | B of 'a tree * 'a * 'a tree;
```

- Tekintsük például az alábbi fát:

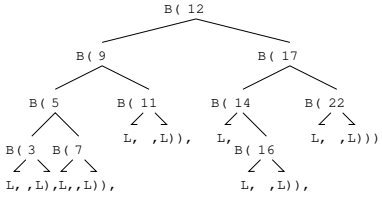


- Az `'a tree` adattípus `L` és `B` adatkonstruktoraival ez a fa pl. a következő lapon látható módon írható le.

Bináris fák `datatype` deklarációval (folyt.)

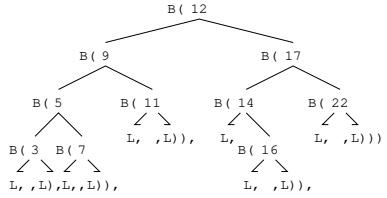
```
B(B(B(B(L,3,L),
        5,
        B(L,7,L)
      ),
     9,
     B(L,11,L)
   ),
  12,
  B(B(L,
        14,
        B(L,16,L)
      ),
     17,
     B(L,22,L)
   )
);
```

A bal oldali kifejezést elég nehéz átlátni. A fastruktúra szöveges leírását megkönnyíti, ha az ábrába beírjuk a megfelelő adatkonstruktorokat.



Bináris fák `datatype` deklarációval (folyt.)

- A fastruktúra szöveges leírása átláthatóbb, ha az egyes részfáknak nevet adunk, és a részfákból építjük fel a teljes fát:




```
val tr3  = B(L,3,L);          val tr7  = B(L,7,L);
val tr5  = B(tr3,5,tr7);      val tr11 = B(L,11,L);
val tr9  = B(tr5,9,tr11);     val tr16 = B(L,16,L);
val tr14 = B(L,14,tr16);      val tr22 = B(L,22,L);
val tr17 = B(tr14,17,tr22);   val tr12 = B(tr9,12,tr17);
```

Bináris fák `datatype` deklarációval (folyt.)

- Másféle fastruktúrákat is deklarálhatunk, pl.
  - kezdhetjük az `'a` típusú értékkel, majd folytathatjuk előbb a bal, azután a jobb részfa megadásával:

    ```
    datatype 'a tree = L | B of 'a * 'a tree * 'a tree
    ```

  - felhasználhatjuk a levelet is értékek tárolására:

    ```
    datatype 'a tree = L of 'a | B of 'a * 'a tree * 'a tree
    ```

  - az értéket nem tároló üres csonkokat pedig `E`-vel jelölhetjük:

    ```
    datatype 'a tree = E | L of 'a | B of 'a * 'a tree * 'a tree
    ```

- A rekurzív függvényekhez hasonlóan a rekurzív adattípusok deklarációjában is kell lennie nemrekurzív ágnak (a rekurzió leállításához).
- A nemrekurzív ág hiánya miatt az alábbi, szintaktikailag helyes deklarációk használhatatlanok:

```
datatype 'a badtree = B of 'a badtree * 'a * 'a badtree
datatype 'a badtree = L of 'a badtree
                    | B of 'a badtree * 'a * 'a badtree
```


Újra az egyidejű deklarációról

- Típusok, ill. értékek *egyidejűleg* is deklarálhatók az `and` kulcsszó alkalmazásával (az utóbbiakra már láttunk példákat).
- Vegyük a következő deklarációsorozatokat:

```
type sor = int; type osz = int;
datatype fa = L | B of fa * fa;
    datatype 'a verem = >| | >> of 'a * 'a verem;
val v1 = "a"; val v2 = "z";
fun f1 i = i +1; fun f2 i = i - 1;
```

A fenti deklarációkat az SML-értelmező a *megadott sorrendben* értékeli ki.

```
type sor = int and osz = int;
datatype fa = L | B of fa * fa and
    'a verem = >| | >> of 'a * 'a verem;
val v1 = "a" and v2 = "z";
fun f1 i = i +1 and f2 i = i - 1;
```

Az `and` szócskával elválasztott deklarációkat az SML-értelmező *egyidejűleg* értékeli ki.

Egyszerű műveletek bináris fákon

Legyen

```
datatype 'a tree = L | N of 'a * 'a tree * 'a tree
```

- `nodes` egy fa csomópontjait számlálja meg.

```
(* nodes : 'a tree -> int
   nodes f = az f fa csomópontjainak a száma *)
fun nodes (N(_, t1, t2)) = 1 + nodes t2 + nodes t1
  | nodes L = 0
```

- Akkumulátort használó változata: `nodesa` – a kétfelé ágazó rekurzió miatt nem nyerünk vele szinte semmit, de sikerült kevésbé érthetővé tennünk. :-(

```
fun nodesa f =
      let (* nodes0(f, n) = n + a csomópontok száma f-ben
             nodes0 : 'a tree * int -> int *)
          fun nodes0 (N(_, t1, t2), n) = nodes0(t1, nodes0(t2, n+1))
            | nodes0 (L, n) = n
      in
         nodes0(f, 0)
      end
```


Egyszerű műveletek bináris fákon (folyt.)

- A fa gyökeréből a leveléhez vezető úton az élek számát (az út hosszát) az adott levél szintjének is nevezzük. A szintek közül a legnagyobbat a fa *mélységének* hívjuk.
- `depth` egy fa mélységét határozza meg.

```
(* depth : 'a tree -> int
   depth f = az f fa mélysége *)
fun depth (N(_, t1, t2)) = 1 + Int.max(depth t2, depth t1)
  | depth L = 0
```

- `depth` akkumulátort használó változata: `deptha` – a kétfelé ágazó rekurzió miatt most sem nyerünk vele szinte semmit.

```
fun deptha f =
      let fun depth0 (N(_, t1, t2), d) =
                              Int.max(depth0(t1, d+1), depth0(t2, d+1))
            | depth0 (L, d) = d
      in
         depth0(f, 0)
      end
```

Egyszerű műveletek bináris fákon (folyt.)

- `fulltree` n mélységű *teljes bináris fát* épít, és a fa csomópontjait 1-től $2^n - 1$-ig beszámozza.

```
(* fulltree : int -> int tree
   fulltree n = n mélységű teljes fa *)
fun fulltree n = let fun ftree (_, 0) = L
                       | ftree (k, n) = N(k, ftree(2*k, n-1),
                                             ftree(2*k+1, n-1))
                 in
                     ftree(1, n)
                 end
```

  Egy teljes bináris fában minden csomópontból pontosan két él indul ki, és minden levelének ugyanaz a szintje.

- `reflect` a fát a függőleges tengelye mentén tükrözi.

```
(* reflect : 'a tree -> 'a tree
   reflect t = a függőleges tengelye mentén tükrözött t fa *)
fun reflect L = L
  | reflect (N(v,t1,t2)) = N(v, reflect t2, reflect t1)
```


Lista előállítása bináris fa elemeiből

- Mindhárom függvény *bináris fából listát* állít elő. Abban különböznek, hogy mikor veszik ki a csomópontokban tárolt értékeket, és milyen sorrendben járják be a részfákat:
  - `preorder` először az értéket veszi ki, majd bejárja a bal, és azután a jobb részfát;
  - `inorder` bejárja a bal részfát, majd kiveszi az értéket, és végül bejárja a jobb részfát;
  - `postorder` először bejárja a bal, majd a jobb részfát, és utoljára veszi ki az értéket.
- Az akkumulátort nem használó változatok egyszerűek, érthetőek, de nem elég hatékonyak a `@` operátor használata miatt.

```
(* preorder, inorder, postorder : 'a tree -> 'a list *)
(* preorder f = az f fa elemeinek preorder sorrendű listája *)
fun preorder L = []
  | preorder (N(v,t1,t2)) = v :: preorder t1 @ preorder t2
(* inorder f = az f fa elemeinek inorder sorrendű listája *)
fun inorder L = []
  | inorder (N(v,t1,t2)) = inorder t1 @ (v :: inorder t2)
(* postorder f = az f fa elemeinek postorder sorrendű listája *)
fun postorder L = []
  | postorder (N(v,t1,t2)) = postorder t1 @ (postorder t2 @ [v])
```


Lista előállítása bináris fa elemeiből (folyt.)

- Ha `inorder` előző változatában az `inorder t1 @ (v :: inorder t2)` kifejezésben a `v :: inorder t2` részkifejezést nem tesszük zárójelbe, a fordító hibát jelez, mivel `::` és `@` azonos precedenciájú, és ezért zárójelek nélkül a nyilvánvalóan hibás `inorder t1 @ v` részkifejezést akarná kiértékelni.
- `inorder` előző megvalósításával kb. egyenértékű a következő változata, amelyben a `v` elem helyett az egyelemű `[v]` listát fűzzük `inorder t2` elé:

```
fun inorder L = []
  | inorder (N(v,t1,t2)) = inorder t1 @ ([v] @ inorder t2)
```

  Ez a változat azonban *roppant sérülékeny,* ugyanis a hatékonysága függ a zárójelek kirakásától. Ha a `[v] @ inorder t2` részkifejezést nem tesszük zárójelbe, akkor a fordító először a `inorder t1 @ [v]` részkifejezést fogja kiértékelni, azaz egy egyelemű listához fűz egy (általában) jóval hosszabbat!
- Az elmondottakhoz hasonló okból `postorder` bemutatott változata is *rendkívül sérülékeny!* Ha ugyanis a `postorder t1 @ (postorder t2 @ [v])` kifejezésben az amúgyis rossz hatékonyságú `postorder t2 @ [v]` részkifejezést nem tesszük zárójelbe, akkor a fordító először a `postorder t1 @ postorder t2` részkifejezést értékeli ki, azaz a két, feltehetően hosszú listát fűzi egybe, majd a létrehozott eredménylistát fűzi az egyelemű listához!

Lista előállítása bináris fa elemeiből (folyt.)

Az akkumulátort használó változatok megint nehezebben érthetők, de talán valamivel *hatékonyabbak*, főleg a veremhasználat szempontjából (`@` helyett `::`-ot használunk).

```
(* preord : 'a tree * 'a list -> 'a list
   preord(f, vs) = az f fa elemeinek a vs lista elé fűzött,
                   preorder sorrendű listája *)
fun preord (L, vs) = vs
  | preord (N(v,t1,t2), vs) = v::preord(t1, preord(t2,vs))

(* inord : 'a tree * 'a list -> 'a list
   inord(f, vs) = az f fa elemeinek a vs lista elé fűzött,
                  inorder sorrendű listája *)
fun inord (N(v,t1,t2), vs) = inord(t1, v::inord(t2,vs))
  | inord (L, vs) = vs

(* postord : 'a tree * 'a list -> 'a list
   postord(f, vs) = az f fa elemeinek a vs lista elé fűzött,
                    postorder sorrendű listája *)
fun postord (N(v,t1,t2), vs) = postord(t1, postord(t2, v::vs))
  | postord (L, vs) = vs
```

Bináris fa előállítása lista elemeiből: balPreorder

- Listát *kiegyensúlyozott* (**bal**anced) *bináris fává* alakítanak a következő függvények: balPreorder, balInorder és balPostorder; a különbség közöttük most is a bejárási sorrendben van.
-
```
(* balPreorder: 'a list -> 'a tree
   balPreorder xs = az xs lista elemeiből álló, preorder
                    bejárású, kiegyensúlyozott fa
*)
fun balPreorder [] = L
  | balPreorder (x::xs) =
      let val k = length xs div 2
      in
         N(x, balPreorder(List.take(xs, k)),
              balPreorder(List.drop(xs, k)))
      end
```
- A hatékonyságot kisebb mértékben rontja, hogy List.take és List.drop egymástól függetlenül *kétszer* mennek végig a lista első felén.

take és drop egyetlen függvénnyel: take'ndrop

- Írjunk take'ndrop néven olyan függvényt, amelynek egy xs listából és egy k egészből álló pár az argumentuma, és egy olyan pár az eredménye, amelynek első tagja a lista első k db eleme, második tagja pedig a lista többi eleme.

```
(* take'ndrop : 'a list * int -> 'a list * 'a list
   take'ndrop(xs, k) = olyan pár, amelynek
                       első tagja xs első k db eleme,
                       második tagja pedig xs maradéka
*)
fun take'ndrop (xs, k) =
      let fun td (xs, 0, ts) = (rev ts, xs)
            | td ([], _, ts) = (rev ts, [])
            | td (x::xs, k, ts) = td(xs, k-1, x::ts)
      in
          td(xs, k, [])
      end
```
- take'ndrop felhasználása, nevezetesen az eredményül átadott pár miatt módosítani kell balpreorder felépítésén.

Bináris fa előállítása lista elemeiből: balPreorder, újra

- Ez volt:

```
fun balPreorder [] = L
  | balPreorder (x::xs) =
      let val k = length xs div 2
      in  N(x, balPreorder(List.take(xs, k)),
               balPreorder(List.drop(xs, k)))
      end
```
- Ez lett:

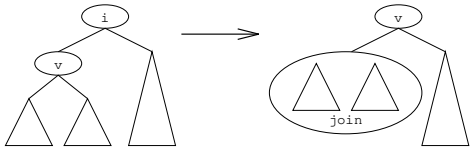
```
(* balPreorder: 'a list -> 'a tree
    balPreorder xs = az xs lista elemeiből álló, preorder ... *)
fun balPreorder [] = L
  | balPreorder (x::xs) =
      let val k = length xs div 2
          val (ts, ds) = take'ndrop(xs, k)
      in  N(x, balPreorder ts, balPreorder ds)
      end
```


Bináris fa előállítása lista elemeiből

-
```
(* balInorder: 'a list -> 'a tree
   balInorder xs = az xs lista elemeiből álló, inorder bejárású,
                   kiegyensúlyozott fa
*)
fun balInorder [] = L
  | balInorder (xxs as x::xs) =
      let val k = length xxs div 2
          val ys = List.drop(xxs, k)
      in
          N(hd ys, balInorder(List.take(xxs, k)),
                   balInorder(tl ys))
      end
```
-
```
(* balPostorder: 'a list -> 'a tree
   balPostorder xs = az xs lista elemeiből álló, postorder
                     bejárású, kiegyensúlyozott fa
*)
fun balPostorder xs = balPreorder(rev xs)
```
- balInorder take'ndrop-pal való definiálását meghagyjuk gyakorló feladatnak.

Elem rekurzív törlése bináris fából

- Adott értékű *elemet* rekurzív módszerrel *megkeresni* egyszerű feladat.
- *Új elemet beszúrni* sem nehéz: rekurzív módszerrel keresünk egy levelet, és ennek a helyére berakjuk az új értéket. Ha a fa rendezve van, ügyelnünk kell arra, hogy a rendezettség megmaradjon.
- Adott értékű *elemet* vagy *elemeket* rekurzív módszerrel *kitörölni* valamivel nehezebb: ha a törlendő érték az éppen vizsgált részfa gyökerében van, a két részre széteső fa részfáit *egyesíteni* kell, miután a törlést a két részfán már végrehajtottuk.



- Megtehetjük, hogy előbb egyesítjük a két részfát, majd az eredményül kapott fából töröljük az adott értékű elemet.

Elem rekurzív törlése bináris fából (folyt.)

- A `join`-nal egyesítjük a törlés hatására létrejövő két részfát: a bal részfát lebontja, és közben az elemeit egyesével berakja a jobb részfába.

```
(* join : 'a tree * 'a tree -> 'a tree
   join(b, j) = a b és a j fák egyesítésével létrehozott fa *)
fun join (L, tr) = tr
  | join (N(v, lt, rt), tr) = N(v, join(lt, rt), tr)
```

- A `remove` rendezetlen bináris fából törli az `i` értékű elem *összes* előfordulását.

```
(* remove : 'a * 'a tree -> 'a tree
   remove(i, f) = i összes előfordulását törli f-ből *)
fun remove (i, L) = L
  | remove (i, N(v,lt,rt)) =
      if i<>v
      then N(v, remove(i,lt), remove(i,rt))
      else join(remove(i,lt), remove(i,rt))
```


Bináris keresőfák: `blookup`, `binsert`

- Rendszerint adott kulcsú elemet keresünk egy rendezett bináris fában, ehhez értékeket kell összehasonlítanunk egymással, ehhez a keresett kulcsnak *egyenlőségi típusúnak* kell lennie (a példában a `string` típust használjuk).
- A függvények *kivételt* jeleznek, ha a keresett kulcsú elem nincs a keresőfában:

```
exception Bsearch of string
```

- A `blookup` függvény adott kulcshoz tartozó értéket ad vissza:

```
(* blookup : (string * 'a) tree * string -> 'a
   blookup(f, b) = az f fában a b kulcshoz tartozó érték
*)
fun blookup (L, b) = raise Bsearch("LOOKUP: " ^ b)
  | blookup (N((a,x), t1, t2), b) =
      if b < a      then blookup(t1,b)
      else if a < b then blookup(t2, b)
      else x;
```


Bináris keresőfák: `bupdate`

- A `binsert` függvény egy új kulcsú elemet rak be egy rendezett bináris fába, ha még nincs benne:

```
(* binsert : (string * 'a) tree * (string * 'a) -> (string * 'a) tree
   binsert(f, (b,y)) = az új (b,y) kulcs-érték párral bővített f fa *)
fun binsert (L, (b,y)) = N((b,y), L, L)
  | binsert (N((a, x), t1, t2), (b,y)) =
      if b < a      then N((a, x), binsert(t1, (b,y)), t2)
      else if a < b then N((a, x), t1, binsert(t2, (b,y)))
      else (* a=b *) raise Bsearch("INSERT: " ^ b);
```

- A `bupdate` függvény meglévő kulcsú elembe új értéket ír be egy rendezett bináris fában:

```
(* bupdate : (string * 'a) tree * (string * 'a) -> (string * 'a) tree
   bupdate(f, (b,y)) = az f fa, a b kulcshoz tartozó érték helyén
                       az y értékkel *)
fun bupdate (L, (b,y)) = raise Bsearch("UPDATE: " ^ b)
  | bupdate (N((a,x), t1, t2), (b,y)) =
      if b < a      then N((a,x), bupdate(t1, (b,y)), t2)
      else if a < b then N((a,x), t1, bupdate(t2, (b,y)))
      else (* a=b *) N((b,y), t1, t2);
```

- A függvények *generikussá* tételét meghagyjuk gyakorló feladatnak.

ABSZTRAKCIÓ FÜGGVÉNYEKKEL (ELJÁRÁSOKKAL)

Legnagyobb közös osztó

- Fontos, hogy algoritmusaink futási idejét és tárigényét ismerjük – különösen korlátos erősforrású (pl. beágyazott) rendszerekben fontos ez.
- Következő példánk a és b legnagyobb közös osztóját számolja ki az euklideszi algoritmussal.
- Az alapgondolat az, hogy ha a-t b-vel osztva r a maradék, akkor a és b közös osztói azonosak b és r közös osztóival.
- A matematikai definíciót most is pontosan követi az SML-függvény.

$$\gcd(a,0) = a$$
$$\gcd(a,b) = \gcd(b, a \bmod b)$$

```
fun gcd (a, 0) = a
  | gcd (a, b) = gcd(b, a mod b)
```

- A függvény *rekurzív,* a *folyamat* iteratív. A lépések száma logaritmikusan nő – miért is?

  Az ún. *Lamé-tétel* szerint – ld. SICP, 1.2.5. szakasz – ha az euklideszi algoritmus egy számpár legnagyobb közös osztóját k lépésben számítja ki, akkor a számpár kisebbik tagja nem lehet kisebb a k-adik Fibonacci-számnál.

  Legyen n az algoritmus kisebbik paramétere. Ha a legnagyobb közös osztó kiszámításához k lépésre van szükség, akkor $n \geq F(k) \approx \Phi^k/\sqrt{5}$. A k lépésszám tehát valóban az n – Φ alapú – logaritmusával arányos.

  Idézzük föl, hogy $\Phi = (1+\sqrt{5})/2 \approx 1.61803$ az *aranymetszés* arányszáma.

Prímteszt

- A `prime` predikátum egy n szám prím voltát teszteli. A `findDivisor` függvény 2-től kezdve megkeresi az n szám legkisebb osztóját. Az n szám prím, ha legkisebb osztója maga az n.
- Az n osztóit 2-től $\sqrt{n}$-ig kell keresni, így a lépések száma $O(\sqrt{n})$.

```
(* prime n = true if n is prime
   prime : int -> bool
*)
fun prime n =
  let
      infix divides
      fun smallestDivisor n = findDivisor(n, 2)
      and findDivisor (n, testDivisor) =
            if square testDivisor > n
            then n
            else if testDivisor divides n
            then testDivisor
            else findDivisor(n, testDivisor+1)
      and square x = x * x
      and a divides b = b mod a = 0
in
    n = smallestDivisor n
end;
```


Prímteszt (folyt.)

- A következő SML-predikátum egy szám prím voltát *valószínűségi módszerrel* teszteli. A lépések száma $O(\lg n)$.
- Az algoritmus a kis Fermat-tételen alapul, amely azt mondja ki, hogy:

  ha n prím és $0 < a < n$, akkor a^n modulo n szerint *kongruens* a-val, azaz $a^n \bmod n = a$ (ld. pl. SICP, 1.2.6. szakasz).

  - Két szám akkor *kongruens* modulo n szerint, ha n-nel osztva mindkettőnek ugyanaz a maradéka. Egy a szám n-nel való osztásának maradékát a modulo n szerinti maradékának, vagy röviden a modulo n-nek is nevezik.
- Ha n nem prím, akkor az $a < n$ számok nagy hányadára nem teljesül a fenti reláció.
- A prímteszt algoritmusa ezek után a következő :
  - Adott n-re véletlenszerűen válasszunk egy $a < n$ számot: ha $a^n \bmod n \neq a$, akkor n nem prím. Ellenkező esetben nagy a valószínűsége, hogy n prím.
  - Válasszunk véletlenszerűen egy másik $a < n$ számot: ha $a^n \bmod n = a$, akkor növekedett annak a valószínűsége, hogy az n prím. Újabb és újabb a értékeket választva egyre biztosabbak lehetünk abban, hogy az n prím.

Prímteszt (folyt.)

- Az `expmod` segédfüggvény a `base` szám `exp`-edik hatványának modulo `m` szerinti maradékát adja eredményül.

```
(* expmod (base, exp, m) = base exp-edik hatványa modulo m
   expmod : int * int * int -> int
*)
fun expmod (_, 0, _) = 1
  | expmod (b, e, m) =
      if even e
      then square(expmod(b, e div 2, m)) mod m
      else b * expmod(b, e-1, m) mod m
and even n = n mod 2 = 0
and square x = x * x;
```

- Nagyon hasonló felépítésű `exptFast`-hoz. A lépések száma a kitevő logaritmusával arányos.

Prímteszt (folyt.)

- Ha mosml-t használunk, betöltjük a `Random` könyvtárat:

```
load "Random";
```

- `fermatTest` generál egy álvéletlen-számot, és egyszer elvégzi a vizsgálatot:

```
(* fermatTest n = false if n is not prime
   fermatTest : int -> bool
*)
fun fermatTest n =
      let fun tryIt a = expmod(a, n, n) = a
      in  tryIt(Random.range (1, n) (Random.newgen()) )
      end;
```

- `fastPrime` `times`-szor megismétli a vizsgálatot:

```
(* fastPrime (n, times) = true if n passes the prime test times times
   fastPrime : int * int -> bool
*)
fun fastPrime (n, 0) = true
  | fastPrime (n, t) = fermatTest n andalso fastPrime(n, t-1);
```

- Ez a megoldás csak nagy valószínűséggel, de nem teljes bizonyossággal ad választ a kérdésre. Például az 561 átmegy a Fermat-teszten, bár nem prím.

Függvények mint általános számítási módszerek

- Láttuk, hogy a függvény (ill. általában az eljárás) olyan *absztrakció*, amely – a paraméterként átadott adatok konkrét értékétől függetlenül – összetett műveleteket ír le.
- Az olyan magasabbrendű függvény, amelynek függvény a paramétere, még *magasabb szintű* absztrakció, hiszen az általa megvalósított összetett műveletet nemcsak egyes konkrét adatoktól, hanem egyes konkrét műveletektől is függetlenné tesszük.
- A magasabbrendű függvény (eljárás) tehát valamilyen *általános számítási módszert* fejez ki.
- A következő lapokon két nagyobb példát ismertetünk a SICP alapján: általános számítási módszert függvények *zérushelyeinek* és *fixpontjának* a megtalálására.

Egyenlet gyökeinek meghatározása intervallumfelezéssel

- Az intervallumfelezés módszere hatékony eljárás az $f(x) = 0$ egyenlet gyökeinek megtalálására, ahol f folytonos függvény.
- A közismert alapötlet a következő:
  - Megfelelően megválasztott a-ra és b-re, amelyekre $f(a) < 0 < f(b)$, f-nek legalább egy zérushelye van a és b között.
  - A zérushely megtalálásához legyen $x = (a + b)/2$. Ha $f(x) > 0$, akkor f zérushelyét a és x között, ha $f(x) < 0$, akkor x és b között kell keresnünk.
  - A keresést – a rekurziót – akkor hagyjuk abba, amikor két egymás utáni közelítő érték *eltérése* egy előre meghatározott értéknél kisebb lesz.
- Mivel az eltérés minden lépésben a felére csökken, az f zérushelyének megtalálásához szükséges lépések száma $O(L/T)$, ahol L az intervallum hossza kezdetben, és T a megengedett eltérés.
- A leírt algoritmust valósítja meg a `search` függvény (ld. a következő lapon):

Egyenlet gyökeinek meghatározása intervallumfelezéssel (folyt.)

```
(* search (f, negPoint, posPoint) = root of f x in the
                                     negPoint <= x <= posPoint interval
     PRE: f negPoint <= 0 and f posPoint >= 0
   search : (real -> real) * real * real -> real
*)
fun search (f, negPoint, posPoint) =
      let val midPoint = average(negPoint, posPoint)
      in
          if closeEnough(negPoint, posPoint)
          then midPoint
          else let val testValue = f midPoint
               in
                   if positive(testValue)
                   then search(f, negPoint, midPoint)
                   else if negative(testValue)
                   then search(f, midPoint, posPoint)
                   else midPoint
                end
      end
and average (x, y) = (x+y)/2.0
and closeEnough (x, y) = abs(x-y) < 0.001
and positive x = x > 0.0
and negative x = x < 0.0;
```


Egyenlet gyökeinek meghatározása intervallumfelezéssel (folyt.)

- Az előfeltételek betartását célszerű `search` alkalmazása előtt ellenőrizni, nehogy rossz választ kapjunk az SML értelmezőtől.

  `load "Math";`

  - Helyes az eredménye:

    `search(Math.sin, 4.0, 2.0);`

    `> val it = 3.14111328125 : real`

  - **Hibás** az eredménye:

    `search(Math.sin, 2.0, 4.0);`

    `> val it = 2.00048828125 : real`

- A `halfIntervalMethod` függvény (ld. a következő lapon) elvégzi az ellenőrzést, és jelzi, ha `negPoint` vagy `posPoint` kezdeti értéke nem jó.
- Figyeljük meg az *ügyek szétválasztása* elv alkalmazását: `search` a gyökkeresési stratégiát valósítja meg, `halfIntervalMethod` pedig az előfeltételek meglétét ellenőrzi.

Egyenlet gyökeinek meghatározása intervallumfelezéssel (folyt.)

- ```
  (* halfIntervalMethod (f, a, b) = root of f x in the a <= x <= b interval
     halfIntervalMethod : (real -> real) * real * real -> real
  *)
  fun halfIntervalMethod(f, a, b) =
        let val (aValue, bValue) = (f a, f b)
        in
            if negativeOrZero aValue andalso positiveOrZero bValue
            then search(f, a, b)
            else if negativeOrZero bValue andalso positiveOrZero aValue
            then search(f, b, a)
            else print ("Values " ^ makestring a ^ " and " ^
                   makestring b ^ " are not of opposite sign.\n")
        end
  and positiveOrZero x = x >= 0.0
  and negativeOrZero x = x <= 0.0;
  ```
- A függvénynek ez a változata hibás, mert az `if-then-else` feltételes kifejezés összes ágának *ugyanolyan típusú* eredményt *kell* adnia, márpedig `print` eredménye nem `int` típusú.
- (Csúnya) megoldás az `(e; f)` alakú *szekvenciális kifejezés* használata: az értelmező kiértékeli `e`-t és `f`-et a felírt sorrendben, eredményül pedig `f` értékét adja (ld. a következő lapon).
- Szebb, ha *kivételt* jelzünk: ezt meghagyjuk gyakorló feladatnak.

Egyenlet gyökeinek meghatározása intervallumfelezéssel (folyt.)

```
(* halfIntervalMethod (f, a, b) = root of f x in the a <= x <= b interval
   halfIntervalMethod : (real -> real) * real * real -> real              *)
fun halfIntervalMethod(f, a, b) =
      let val (aValue, bValue) = (f a, f b)
      in if negativeOrZero aValue andalso positiveOrZero bValue
         then search(f, a, b)
         else if negativeOrZero bValue andalso positiveOrZero aValue
         then search(f, b, a)
         else (print ("Values " ^ makestring a ^ " and " ^ makestring b ^
                      " are not of opposite sign.\n"); 0.0)
      end
and positiveOrZero x = x >= 0.0
and negativeOrZero x = x <= 0.0;
```

```
halfIntervalMethod(Math.sin, 2.0, 4.0);
> val it = 3.14111328125 : real
halfIntervalMethod(fn x => x*x*x-2.0*x-3.0, 1.0, 2.0);
> val it = 1.89306640625 : real
halfIntervalMethod(Math.sin, 2.0, 2.5);
Values 2.0 and 2.5 are not of opposite signs
> val it = 0.0 : real
```
```

Függvény fixpontjának meghatározása

- Az $f(x) = x$ egyenletet kielégítő x az f függvény *fixpontja.*
- Egy f függvény valamely fixpontját megfelelő kezdőértékből kiindulva f rekurzív alkalmazásával határozhatjuk meg:
  $fx, f(fx), f(f(fx)), f(f(f(fx))), \ldots$
  A rekurzió akkor fejezhető be, amikor már elhanyagolható mértékű a változás.
- A `fixedPoint` függvény paramétere egy pár; ennek első tagja egy függvény, amelynek a fixpontját keressük, a második tagja pedig a fixpont egy első közelítése:
  `fixedPoint (f, firstGuess)` = fixpoint of `f` in the proximity of `firstGuess` with tolerance `tolerance`.
- Szükségünk van tehát még a közelítés megkívánt pontosságára:

```
val tolerance = 0.00001;
```


Függvény fixpontjának meghatározása (folyt.)

```
(* fixedPoint (f, firstGuess) = fixpoint of f in the proximity of
                                firstGuess with tolerance tolerance
   fixedPoint : (real -> real) * real -> real
*)
fun fixedPoint (f, firstGuess) =
      let
          fun closeEnough (v1, v2) = abs(v1-v2) < tolerance
          fun try guess =
                let
                    val next = f guess
                in
                    if closeEnough(guess, next)
                    then next
                    else try next
                end
      in
          try firstGuess
      end;

fixedPoint(Math.cos, 1.0);
fixedPoint(fn y => Math.sin y + Math.cos y, 1.0);
```


Függvény fixpontjának meghatározása (folyt.)

- A fixpontszámítás hasonlít a négyzetgyökvonás korábban megbeszélt folyamatára: mindkettő azon alapul, hogy addig finomítjuk a közelítést, amíg valamilyen feltétel nem teljesül.
- A négyzetgyökvonás könnyedén megfogalmazható fixpontszámításként: ha x négyzetgyöke y, akkor $y * y = x$, azaz $y = x/y$. Az $fy = x/y$ függvény fixpontja tehát az x négyzetgyöke.

```
fun sqrt x = fixedPoint (fn y => x/y, 1.0);
```

- A megoldásunk rossz, ugyanis nem konvergál! Könnyen belátható:
  Legyen x négyzetgyökének első közelítése $y1$, a második $y2 = x/y1$, a harmadik $y3 = x/y2 = x/(x/y1) = y1$. Látható, hogy a folyamat sohasem ér véget.
- Az oszcillációt pl. úgy gátolhatjuk meg, hogy *korlátozzuk* két közelítő érték között a változás mértékét.
- Mivel a helyes válasz mindig az y közelítő érték és x/y között van, y-hoz x/y-nál *közelebb eső* új közelítő értékként y és x/y átlagát választhatjuk: $y \leftarrow (y + x/y)/2$.

```
fun sqrt x = fixedPoint (fn y => (y+x/y)/2.0, 1.0);
```

- Ezt a gyakran használható módszert *átlagcsillapításnak* (angolul *average damping)* nevezik.

Az átlagcsillapítás módszere

- A függvényekről mint absztrakciós eszközökről szólva eddig olyan magasabbrendű függvényeket használtunk, amelyeknek más függvények voltak a paraméterei.
- Most olyan magasabbrendű függvényeket mutatunk be, amelyek *függvényt* (pontosabban *függvényértéket)* adnak eredményül.
- A korábban bemutatott *átlagcsillapítás* sokszor használható módszer, ezért érdemes önálló függvényként megírni: ha adott az f függvény, elő kell állítani x és fx átlagát.

```
(* averageDamp f = f valamely x-re alkalmazva előállítja x és f x átlagát
   averageDamp : (real -> real) -> real -> real
*)
fun averageDamp f = fn x => (x + f x) / 2.0;
```

- Jól látható, hogy `averageDamp`, ha csak egyetlen paraméterre alkalmazzuk, függvényértéket ad eredményül. `averageDamp` részlegesen alkalmazható függvény.
- Példa `averageDamp` alkalmazására:

```
(averageDamp (fn x => x*x)) 10.0; (* 10.0 és 100.0 átlaga *)
```

- A kiértékelés sorrendje miatt a külső zárójelpár el is hagyható:

```
averageDamp (fn x => x*x) 10.0;
```

Az átlagcsillapítás módszere (folyt.)

- `averageDamp` definíciója felírható a `fun` *szintaktikai édesítőszerrel* is:

```
fun averageDamp f x = (x + f x) / 2.0;
```

- `sqrt averageDamp`-pel felírt változata explicitté teszi a *fixpontmeghatározás* és az *átlagcsillapítás* módszerét, továbbá az $y = x/y$ *egyenlet használatát.*

```
fun sqrt x = fixedPoint(averageDamp (fn y => x/y), 1.0);
sqrt 4.0;
```

- Tanulság: egy folyamatot sokféle eljárással leírhatunk, de a *lényeget* sokkal könnyebb megérteni, ha *megfelelően megválasztott absztrakciókat* vezetünk be.
- Még egy példa a bemutatottak alkalmazására: az x köbgyöke az $y \mapsto x/y^2$ – SML-jelöléssel az `fn y => x/(y*y)` – függvény fixpontja. A megoldás már kész is van!

```
fun cubeRoot x  = fixedPoint(averageDamp (fn y => x/y/y), 1.0);
cubeRoot 8.0;
```


Az általános Newton-módszer

- Legyen $x \mapsto g(x)$ egy differenciálható függvény és $f(x) = x - g(x)/g'(x)$, ahol $g'(x)$ a g függvény x szerinti deriváltja. Ekkor a $g(x) = 0$ egyenlet x megoldása az $x \mapsto f(x)$ függvény egy fixpontja.
- Az *általános Newton-módszer* tehát a fixpontmódszer egy alkalmazása az f függvény egy fixpontjának megtalálására. Számos g függvényre és megfelelően megválasztott x értékre az általános Newton-módszer gyorsan konvergál.
- Először is azt a `deriv` függvényt kell definiálnunk, amelynek (az `averageDamp` függvényhez hasonlóan) függvény a paramétere, és függvényt ad eredényül.
- Ha g függvény és dx egy kis szám, akkor a g függvény g' deriváltja az a függvény, amelynek értéke bármely x számra a következő: $g'(x) = (g(x + dx) - g(x))/dx$.

```
(* deriv g = g deriváltja
   deriv : (real -> real) -> real -> real
*)
val dx = 0.00001;
fun deriv g = fn x => (g(x+dx) - g x) / dx;
```

- Például az $x \mapsto x^3$ függvény deriváltja $x = 5$-re (pontos értéke 75):

```
let fun cube x = x*x*x in deriv cube 5.0 end;
```


A Newton-módszer fixpont-folyamatként

- `deriv` felhasználásával az általános Newton-módszer *fixpont-folyamatként* definiálható:

```
(* newtonTransform g x = g transzformáltja az általános
                         Newton-módszer alkalmazásához
   newtonTransform : (real -> real) -> real -> real
   newtonsMethod g guess = g gyökhelye, ahol guess a
                           gyökkeresés kezdőértéke
   newtonsMethod : (real -> real) -> real -> real
*)
fun newtonTransform g x = x - (g x / deriv g x)
and newtonsMethod g guess = fixedPoint(newtonTransform g, guess);
```

- Példa `newtonsMethod` használatára:

```
fun sqrt x = newtonsMethod (fn y => y*y-x) 1.0;
sqrt 16.0;
```


A fixpontmódszer kétféle alkalmazása

- Két általános módszer egy-egy alkalmazását láttuk egy szám négyzetgyökének kiszámítására: az egyik a fixpont-, a másik a Newton-módszer.
- Mivel az utóbbi is a fixpontmódszeren alapul, valójában a fixpontmódszer kétféle alkalmazását láttuk.
- Mindkét esetben egy függvényből indulunk ki, és kiszámítjuk valamely transzformáltjának egy fixpontját.
- Ezt az általános módszert is definiálhatjuk eljárásként (függvényként):

```
(* fixedPointOfTransform (g, transform, guess) = a fixed point
                          of transform g with the initial guess guess
   fixedPointOfTransform :
                          'a * ('a -> real -> real) * real -> real
*)
fun fixedPointOfTransform (g, transform, guess) =
                                  fixedPoint(transform g, guess);
```

A fixpontmódszer kétféle alkalmazása (folyt.)

- Ez volt például `sqrt` fixpontkeresésen alapuló első változata:

```
fun sqrt x = fixedPoint(averageDamp (fn y => x/y), 1.0)
```

- Átírva az általános módszert megvalósító függvénnyel:

```
fun sqrt x = fixedPointOfTransform (fn y => x/y,
                                    averageDamp, 1.0)
```

- Ez volt például `sqrt` Newton általános módszerét használó második változata:

```
fun sqrt x = newtonsMethod (fn y => y*y-x) 1.0;
```

- Átírva az általános módszert megvalósító függvénnyel:

```
fun sqrt x = fixedPointOfTransform (fn y => y*y-x,
                                    newtonTransform, 1.0)
```


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 előnyös 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 ($|\ x - y\ | < \varepsilon$) 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 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 (x_i, y_j) párokból, ahol $x_i \in$ `xs` és $y_j \in$ `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 (x_i, y_j) párokból, ahol $x_i \in$ `xs` és $y_j \in$ `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)` = $\{S \cup \texttt{base} \mid S \subseteq \texttt{xs}\}$, azaz $\texttt{xs} \cup \texttt{base}$ 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];
```