

ÖSSZETETT ADATTÍPUSOK



Rekord és ennes

- Két különböző típusú értékből rekordot vagy párt képezhetünk. Pl.

$\{x = 2, y = 1.0\} : \{x : \text{int}, y : \text{real}\}$ és $(2, 1.0) : (\text{int} * \text{real})$

- A pár is csak szintaktikus édesítőszer. Pl.

$(2, 1.0) \equiv \{1 = 2, 2 = 1.0\} \equiv \{2 = 1.0, 1 = 2\} \neq \{1 = 1.0, 2 = 2\}$.

Egy párban a tagok sorrendje meghatározó! Az 1 és a 2 is: *mezőnevek*.

- Rekordot kettőnél több értékből is összeállíthatunk. Pl.

$\{\text{nev} = \text{"Bea"}, \text{tel} = 3192144, \text{kor} = 19\} : \{\text{kor} : \text{int}, \text{nev} : \text{string}, \text{tel} : \text{int}\}$

Egy hasonló rekord egészszám-mezőnevekkel:

$\{1 = \text{"Bea"}, 3 = 3192144, 2 = 19\} : \{1 : \text{string}, 2 : \text{int}, 3 : \text{int}\}$

Az *utóbbi* azonos az alábbi *ennessel* (n-tuple):

$(\text{"Bea"}, 19, 3192144) : (\text{string} * \text{int} * \text{int})$

azaz

$(\text{string} * \text{int} * \text{int}) \equiv \{1 = \text{string}, 2 = \text{int}, 3 = \text{int}\}$

- Egy rekordban a tagok sorrendje közömbös, a tagokat a mezőnév azonosítja. Egy ennesben a tagok sorrendje nem közömbös, a tagokat a *pozícionális* mezőnév azonosítja.

GYENGE ÉS ERŐS ABSZTRAKCIÓ



Adattípusok: gyenge és erős absztrakció

- Gyenge absztrakció: a név szinonima, az adatszerkezet részei továbbra is hozzáférhetőek.
- Erős absztrakció: a név új dolgot (entitást, objektumot, izét) jelöl, az adatszerkezet részeihez csak korlátok között lehet hozzáférni.
- `type`: gyenge absztrakció; pl. `type rat = {num : int, den : int}`
 - Új nevet ad egy típuskifejezésnek (vö. értékdeklaráció).
 - Segíti a programszöveg megértését.
- `abstype`: erős absztrakció
 - Új típust hoz létre: név, műveletek, ábrázolás, jelölés.
 - Túlhaladott, van helyette jobb: `datatype` + modulok
- `datatype`: modulok nélkül gyenge, modulokkal erős absztrakció;
 - pl. `datatype 'a esetleg = Semmi | Valami of 'a`
 - Belső változata az SML-ben: `datatype 'a option = NONE | SOME of 'a`
 - Új entitást hoz létre.
 - Rekurzív és polimorf is lehet.

Adattípusok: felsorolásos és polimorf típusok `datatype` deklarációval

- `datatype logi = Igen | Nem` Felsorolásos típus.
`datatype logi3 = igen | nem | talan` Felsorolásos típus.
`datatype 'a esetleg = Semmi | 'a Valami` Polimorf típus.
- *Adatkonstruktor*nak nevezzük a létrehozott `Igen`, `Nem`, `igen`, `nem`, `talan`, `Semmi` és `Valami` értékeket. `Valami` ún. *adatkonstruktorfüggvény*, az összes többi ún. *adatkonstruktorállandó*. Az adatkonstruktorok a többi értéknévvel azonos névtérben vannak.
- *Típuskonstruktor*nak nevezzük a létrehozott `logi`, `logi3` és `esetleg` neveket; `esetleg` ún. postfix *típuskonstruktorfüggvény* (vagy típusoperátor), a másik kettő ún. *típuskonstruktorállandó* (röviden típusállandó). Típusnévként használható a típusállandó (pl. `logi`), valamint a típusállandóra vagy típusváltozóra alkalmazott típuskonstruktorfüggvény (pl. `int list` vagy `'a esetleg`). A típuskonstruktorok más névtérben vannak, mint az értéknevek.
- Természetesen az adatkonstruktoroknak is van típusuk, pl.

<code>Igen</code>	<code>:</code>	<code>logi</code>	<code>Semmi</code>	<code>:</code>	<code>'a esetleg</code>
<code>Nem</code>	<code>:</code>	<code>logi</code>	<code>Valami</code>	<code>:</code>	<code>'a -> 'a esetleg</code>
- Példa `datatype` deklarációval létrehozott adattípust kezelő függvényre


```
fun inverz Nem = Igen | Igen = Nem
```

LISTÁK

Listák összefűzése (append) és megfordítása (nrev)

- Két lista összefűzése (append, infix változatban @)

$$[x_1, \dots, x_m] @ [y_1, \dots, y_n] = [x_1, \dots, x_{m-1}] @ (x_m :: [y_1, \dots, y_n]) = \dots = [x_1, \dots, x_m, y_1, \dots, y_n]$$

Az xs -t először az elemeire bontjuk, majd hátulról visszafelé haladva fűzzük az elemeket az ys -hez, ugyanis a listákat csak előlről tudjuk építeni. A lépések száma $O(n)$.

```
(* append : 'a list * 'a list -> 'a list
   append xs, ys) = xs összes eleme ys elé fűzve *)
fun append ([], ys) = ys
  | append (x::xs, ys) = x::append(xs, ys)
```

- Lista naív megfordítása (nrev)

$$\text{nrev}[x_1, x_2, \dots, x_m] = \text{nrev}[x_2, \dots, x_m] @ [x_1] = \text{nrev}[\dots, x_m] @ [x_2] @ [x_1] = \dots = [x_m, \dots, x_1]$$

A lista elejéről levett elemet egyelemű listaként tudjuk a végéhez fűzni. A lépések száma $O(n^2)$.

```
(* nrev : 'a list -> 'a list
   nrev xs = xs megfordítva *)
fun nrev [] = []
  | nrev (x::xs) = (nrev xs) @ [x]
```

Listák megfordítása: példa nrev alkalmazására

● Egy példa nrev egyszerűsítésére

```

fun nrev [] = []
  | nrev (x::xs) = (nrev xs) @ [x]

fun [] @ ys = ys
  | (x::xs) @ ys = x :: xs @ ys (* = x :: (xs @ ys) *)

```

A :: és a @ jobbra kötnek, precedenciaszintjük 5.

```

nrev([1, 2, 3, 4]) → nrev([2, 3, 4])@[1] → nrev([3, 4])@[2]@[1]
→ nrev([4])@[3]@[2]@[1] → nrev([])@[4]@[3]@[2]@[1]
→ []@[4]@[3]@[2]@[1] → [4]@[3]@[2]@[1]
→ 4::[]@[3]@[2]@[1] → 4::[3]@[2]@[1])
→ [4, 3]@[2]@[1]) → 4::([3]@[2])@[1])
→ []@[4]@(3::[2, 1]) → []@[4]@[3, 2, 1] → ...

```

nrev rossz hatékonyságú: a lépések száma $O(n^2)$.

Listák összefűzése (revApp) és megfordítása (rev)

- Egy lista elemeinek egy másik lista elé fűzése fordított sorrendben (revApp)

```
(* revApp : 'a list * 'a list -> 'a list
   revApp(xs, ys) = xs elemei fordított sorrendben ys elé fűzve
*)
fun revApp ([], ys) = ys
  | revApp (x::xs, ys) = revApp(xs, x::ys)
```

revApp lépésszáma arányos a lista hosszával. Segítségével rev hatékonyan:

```
(* rev : 'a list -> 'a list
   rev xs = xs megfordítva
*)
fun rev xs = revApp (xs, [])
```

Egy 1000 elemű listát rev 1000 lépésben, nrev $\frac{1000 \cdot 1001}{2} = 500500$ lépésben fordít meg. Hatalmas a nyereség!

- append – @ néven, infix operátorként – és rev beépített függvények, List.revApp pedig List.revAppend néven könyvtári függvény az SML-ben.

LISTÁK HASZNÁLATA

Példák listák használatára: számtani sorozatok előállítása

Naív változat

```
(* upto1 : int -> int -> int list
   upto1 m n = 1 különbségű növekvő számtani sorozat
               az alulról zárt [m,n) tartományban
*)
fun upto1 m n = if m >= n then [] else m :: upto1 (m+1) n;
```

Naív változat, egy segédfüggvénnyel

```
(* upto2 : int -> int -> int list
   upto2 m n = 1 különbségű növekvő számtani sorozat
               az alulról zárt [m,n) tartományban
*)
fun upto2 m n =
  let fun up m = if m >= n then [] else m :: up (m+1)
      in
        up m
      end;
```

Példák listák használatára: számtani sorozatok előállítása (folyt.)

Naív változatok d különbséggel

```
(* fnto? : int -> int -> int -> int list
   fnto? d m n = d>0 különbségű, m<n esetén növekvő, m>n esetén csökkenő
   számtani sorozat az alulról, ill. felülről zárt [m,n) tartományban
*)
```

...két segédfüggvénnyel

```
fun fnto1 d m n =
  let fun up m = if n <= m then [] else m :: up (m+d)
        dn m = if m <= n then [] else m :: dn (m-d)
      in if d <= 0 then []
          else if n >= m then up m
                else (* if n < m then *) dn m
      end;
```

...egy segédfüggvénnyel

```
fun fnto2 d m n =
  let fun to cmp inc m = if cmp m then [] else m :: to cmp inc (inc m)
      in if d <= 0 then []
          else if n >= m then to (fn i => i >= n) (fn j => j + d) m
                else (* if n < m then *) to (fn i => i <= n) (fn j => j - d) m
      end;
```

Példák listák használatára: számtani sorozatok előállítása (folyt.)

Akkumulátoros változat d különbséggel, egy segédfüggvénnyel

(* fnto3 : int -> int -> int -> int list

fnto3 d n m = d>0 különbségű, m<n esetén növekvő, m>n esetén csökkenő számtani sorozat az alulról, ill. felülről zárt [m,n) tartományban

*)

```
fun fnto3 d m n =
```

```
  let fun to cmp inc m zs = if cmp m      Ha az eredménylista sorrendje
                             then rev zs közömbös, a rev elhagyható.
                             else to cmp inc (inc m) (m::zs)
```

```
  in
```

```
    if d <= 0
```

```
      then []
```

```
    else if n >= m
```

```
      then to (fn i => i >= n) (fn j => j + d) m []
```

```
    else (* if n < m then *)
```

```
      to (fn i => i <= n) (fn j => j - d) m []
```

```
  end;
```

Példák listák használatára: számtani sorozatok előállítása (folyt.)

Generikus akkumulátoros változat komparáló és különbségi függvénnyel

```
(* fnto4 : ('a * 'a -> bool) -> ('a -> 'a) -> 'a -> 'a -> 'a list
   fnto4 le df m n = az le kisebb-vagy-egyenlő reláció és a df
   különbségi függvény szerint növekvő sorozat az alulról,
   ill. felülről zárt [m,n) tartományban
```

*)

```
fun fnto4 le df m n =
  let fun to cmp m zs = if cmp m
                        then rev zs
                        else to cmp (df m) (m::zs);

      infix le
  in
    if      m le df m andalso m le n  (*df növel, [m,n)<>üres*)
    then to (fn i => n le i) m []
    else if df m le m andalso n le m  (*df csökkent, [m,n)<>üres*)
    then to (fn i => i le n) m []
    else []
  end;
```

Ha az eredménylista sorrendje
közömbös, a rev elhagyható.

Példák listák használatára: számtani sorozatok előállítása (folyt.)

Egyszerű akkumulátoros változat, egy segédfüggvénnyel

Az egyesével növekvő sorozat utolsó eleme ismert ($n-1$), ezért a listát hátulról visszafelé építjük fel.

```
(* upto3 : int -> int -> int list
   upto3 m n = 1 különbségű növekvő számtani sorozat
               az alulról zárt [m,n) tartományban
*)
fun upto3 m n =
  let
    (* hátulról visszafelé haladva építjük a listát! *)
    fun up n zs = if m >= n then zs else up (n-1) (n-1::zs)
  in
    up n []
  end;

infix --;
fun fm -- to = upto3 fm to;
```

Gyakorló feladat: a $d = +1$ vagy -1 különbségű számtani sorozatot előállító `upto4 d m n` függvény megírása.

Példák listák használatára: számtani sorozatok előállítása (folyt.)

```
use "upto.sml";
```

```
(* egyesével növekvő egészlisták *)
```

```
val ns50k   =      1 -- 50000;  
val ns150k  =  50001 -- 200000;  
val ns200k  = 200001 -- 400000;  
val ns400k  = 200001 -- 600000;  
val ns600k  = 600001 -- 1200000;  
val ns1M    = 100001 -- 1100000;  
val ns2M    = 100001 -- 2100000;
```

```
(* más sorozatok *)
```

```
  (* 1.5-del csökkenő, 666667 elemű valóslista *)  
val rs666667 = fmt4 op>= (fn i => i-1.5) 1000000.0 0.0;  
  (* #"A"-tól kezdve minden 2. betűt tartalmazó 13 elemű lista *)  
val cs13      = fmt4 op<= (fn i => chr(ord i + 2)) #"A", #"Z";
```


FUTÁSI IDŐ MÉRÉSE

Futási idő mérése PolyML alatt

- Akkumulátor nélküli, naív változatok (PolyML: "Increasing stack")

```
PolyML.timing(true);
```

```
val _ = upto1 1 1000000;      (* +1 különbségű növekvő sorozat *)
val _ = upto2 1 1000000;      (* +1 különbségű növekvő sorozat, 1 segédfv. *)
val _ = fto1 1 1 1000000;     (* +d különbségű növekvő/csökkenő, 1 segédfv. *)
val _ = fto2 1 1 1000000;     (* +d különbségű növekvő/csökkenő, 2 segédfv. *)
```

```
PolyML.timing(false);
```

Részletek a PolyML válaszából:

```
Warning - Increasing stack from 147456 to 294912 bytes
```

```
...
```

```
Timing - parse:0.0, semantics:0.0, translate:0.0, generate:0.0, run:1.9
```

```
Timing - parse:0.0, semantics:0.0, translate:0.0, generate:0.0, run:1.4
```

```
Warning - Increasing stack from 9437184 to 18874368 bytes
```

```
Timing - parse:0.0, semantics:0.0, translate:0.0, generate:0.0, run:2.2
```

```
Timing - parse:0.0, semantics:0.0, translate:0.0, generate:0.0, run:1.5
```

Futási idő mérése PolyML alatt (folyt.)

● Akkumulátoros változatok

```
PolyML.timing(true);
```

```
val _ = upto3 1 1000000;    (* +1 különbségű növekvő sorozat, végéről kezd *)
val _ = fmo3 1 1 1000000;  (* +d különbségű növekvő/csökkenő, 1 segéd fv. *)
val _ = fmo4 op<= (fn i => i+1) 1 1000000;    (* +d kül., gen. *)
val _ = fmo4 op<= (fn i => i+1.0) 1.0 1000000.0; (* +d kül., gen. *)
```

```
PolyML.timing(false);
```

Részletek a PolyML válaszából:

```
Timing - parse:0.0, semantics:0.0, translate:0.0, generate:0.0, run:0.8
Timing - parse:0.0, semantics:0.0, translate:0.0, generate:0.0, run:2.4
Timing - parse:0.0, semantics:0.0, translate:0.0, generate:0.0, run:2.2
Timing - parse:0.0, semantics:0.0, translate:0.0, generate:0.0, run:4.6
```

FONTOS APRÓSÁGOK



Fontos apróságok az SML-ről

- A nullas és a unit típus

A `()` vagy `{}` jelet *nullasnak* nevezzük, típusa: `unit`. A nullas a `unit` típus egyetlen eleme. A `unit` típusműveletek *egységeleme*.

- A `print` függvény

Ha a `string` \rightarrow `unit` típusú `print` függvényt egy füzérre alkalmazzuk, eredménye a *nullas*, *mellékhatásként* pedig kiírja a a füzér értékét.

- Az `(e1; e2; e3)` szekvenciális kifejezés *eredménye* azonos az `e3` kifejezés eredményével.

Ha az `e1` és `e2` kifejezéseknek van mellékhatásuk, az érvényesül. `(e1; e2; e3)` egyenértékű a következő `let`-kifejezéssel:

```
let val _ = e1 val _ = e2 in e3 end
```

- Az `e1 before e2 before e3` kifejezés *eredménye* azonos az `e1` kifejezés eredményével.

Ha az `e2` és `e3` kifejezésnek van mellékhatása, az érvényesül. `e1 before e2 before e3` egyenértékű a következő `let`-kifejezéssel:

```
let val e = e1 val _ = e2 val _ = e3 in e end
```

FUTÁSI IDŐ MÉRÉSE

Futási idő mérése: Timer – SML Basis Library

```
type cpu_timer
type real_timer
val startCPUtimer : unit -> cpu_timer
val checkCPUtimer : cpu_timer ->
    {usr: Time.time, sys: Time.time, gc: Time.time}
```

[cpu_timer] is the type of timers for measuring CPU time consumption (user time, garbage collection time, and system time).

[real_timer] is the type of timers for measuring the passing of real time (wall-clock time).

[startCPUtimer ()] returns a cpu_timer started at the moment of the call.

[checkCPUtimer tmr] returns {usr, sys, gc} where usr is the amount of user CPU time consumed since tmr was started, gc is the amount of user CPU time spent on garbage collection, and sys is the amount of system CPU time consumed since tmr was started. Note that gc time is included in the usr time. Under MS DOS, usr time and gc time are measured in real time.

Futási idő mérése: Timer és Time – SML Basis Library (folyt.)

```
val startRealTimer : unit -> real_timer  
val checkRealTimer : real_timer -> Time.time
```

[startRealTimer ()] returns a `real_timer` started at the moment of the call.

[checkRealTimer tmr] returns the amount of real time that has passed since `tmr` was started.

```
eqtype time  
exception Time  
val fmt : int -> time -> string
```

[time] is a type for representing durations as well as absolute points in time (which can be thought of as durations since some fixed time zero).

[zeroTime] represents the 0-second duration, and the origin of time, so $\text{zeroTime} + t = t + \text{zeroTime} = t$ for all t .

[fmt n t] returns as a string the number of seconds represented by t , rounded to n decimal digits. If $n \leq 0$, then no decimal digits are reported.

Futási idő mérése: idomero

```
(* app load ["Timer", "Time"]; *)

(* idomero = fn : ('a -> 'b) -> 'a -> 'b * time * time
   idomero f a = az a argumentumra alkalmazott f függvény
   eredményéből, továbbá a kiértékeléséhez felhasznált CPU-időt
   és valós időt másodpercben tartalmazó füzérekből álló hármas
*)
fun idomero f a =
  let
    val RealStart   = Timer.startRealTimer()
    val CPUstart    = Timer.startCPUTimer()
    val res = f a
    val {usr, ...} = Timer.checkCPUTimer CPUstart
    val rtm       = Timer.checkRealTimer RealStart
  in
    (res, usr, rtm)
  end;
```

Futási idő mérése: `idoiro`

```
(* idoiro = fn : string -> ('a -> 'b) -> 'a -> 'b
   idoiro t f a = f a
       mellékhatóként kiírja a t füzért, az a argumentumra
       alkalmazott f függvény kiértékeléséhez felhasznált CPU-időt
       és valós időt (mp-ben)
*)
fun idoiro t f a =
  let val (res, usr, rtm) = idomero f a
  in
    (print (t ^ " CPU time: " ^ Time.fmt 2 usr ^
            ", real time: " ^ Time.fmt 2 rtm ^ " (sec)\n");
     res
    )
  end;
```

LISTÁK HASZNÁLATA

Példák listák használatára: max1 még több változatban

```
(* app load ["Int"]; *)
use "idomero.sml";
use "szamlistak.sml";

(* max1? : int list -> int
   max1? ns = az ns egészlista legnagyobb eleme
*)

fun max11 [n]          = n
  | max11 (n::ns)     =
      Int.max(n, max11 ns)
  | max11 []          = raise Empty;

fun max12 [n]          = n
  | max12 (n::m::ns)  =
      max12(Int.max(n, m)::ns)
  | max12 []          = raise Empty;

fun max13 []          = raise Empty
  | max13 [n]         = n
  | max13 (n::m::ns) =
      max13(Int.max(n, m)::ns);

fun max14 [] = raise Empty
  | max14 ns =
      let (* [] esete lefedetlen! *)
        fun mxl [n]          = n
          | mxl (n::m::ns) =
              mxl(Int.max(n, m)::ns)
        in mxl ns
      end;

fun max15 []          = raise Empty
  | max15 (n::ns)     =
      let fun mxl (m, [])      = m
          | mxl (m, n::ns) =
              mxl(Int.max(m, n), ns)
        in mxl(n, ns)
      end;
```

Példák listák használatára: max1 még több változatban (folyt.)

```
(* max1? : ('a*'a->'a) * 'a list -> 'a
   max1? (max, ns) = az ns lista max
             szerinti legnagyobb eleme
*)
fun max16 (max, [])      = raise Empty
  | max16 (max, [n])    = n
  | max16 (max, n::m::ns) =
      max16(max, max(n, m)::ns);

fun max17 (max, []) = raise Empty
  | max17 (max, ns) =
      let (* [] esete lefedetlen! *)
          fun mxl [n]      = n
            | mxl (n::m::ns) =
                mxl(max(n, m)::ns)
          in mxl ns end;

fun max18 (max, [])      = raise Empty
  | max18 (max, n::ns) =
      let fun mxl (m, [])      = m
            | mxl (m, n::ns) =
                mxl(max(m, n), ns)
          in mxl(n, ns) end;
```

```
datatype 'a esetleg = Semmi
                  | Valami of 'a;
(* max19 : ('a*'a->'a) * 'a list
           -> 'a esetleg
   max19 (max, ns) = az ns lista max
             szerinti legnagyobb eleme
*)
fun max19 (max, [])      = Semmi
  | max19 (max, n::ns) =
      let fun mxl (m, [])      = Valami m
            | mxl (m, n::ns) =
                mxl(max(m, n), ns)
          in mxl(n, ns)
          end;
```

● Példák max1 alkalmazására:

```
max17(Int.max, [1,3,9,7,4,6,2]) = 9;
max18(op+, [1,2,3,4,5,6,7,8,9]) = 45;
max19(op*, [1,2,3,4,5,6,7,8,9]) =
                                     Valami 36288
```

● Furcsa „maximum” max18(...) és max19(...) eredménye!

max1 futási idejének mérése idoiro-val

```
idoiro "max11, ns2M: " max11 ns2M;
idoiro "max12, ns2M: " max12 ns2M;
idoiro "max13, ns2M: " max13 ns2M;
idoiro "max14, ns2M: " max14 ns2M;
idoiro "max15, ns2M: " max15 ns2M;
idoiro "max16, ns2M: " max16(Int.max, ns2M);
idoiro "max17, ns2M: " max17(Int.max, ns2M);
idoiro "max18, ns2M: " max18(Int.max, ns2M);
idoiro "max19, ns2M: " max19(Int.max, ns2M);
```

Részletek a PolyML válaszából:

```
Warning - Increasing stack from 9437184 to 18874368 bytes
max11, ns2M:      CPU time: 4.57, real time: 7.99 (sec)
max12, ns2M:      CPU time: 0.73, real time: 1.00 (sec)
max13, ns2M:      CPU time: 0.80, real time: 0.80 (sec)
max14, ns2M:      CPU time: 0.83, real time: 0.83 (sec)
max15, ns2M:      CPU time: 0.14, real time: 0.16 (sec)   !!!
max16, ns2M:      CPU time: 1.41, real time: 1.41 (sec)
max17, ns2M:      CPU time: 1.54, real time: 1.55 (sec)
max18, ns2M:      CPU time: 0.85, real time: 0.85 (sec)
    val it = 2099999 : Int.int
max19, ns2M:      CPU time: 0.86, real time: 0.85 (sec)
    val it = Valami 2099999 : Int.int esetleg
```

LISTÁK HASZNÁLATA

Példák listák használatára: futamok előállítása

A futam olyan elemekből álló lista, amelynek szomszédos elemei kielégítenek egy predikátumot. Írjon olyan SML függvényt `futamok` néven, amelynek egy lista futamaiból álló lista az eredménye.

• Első változat: futam és maradék előállítása két függvénnyel

```
(* futam1 : ('a * 'a -> bool) -> ('a * 'a list) -> 'a list
   futam1 p (x, ys) = az x::ys p-t kielégítő első futama (prefixuma *)
fun futam1 p (x, [])      = [x]
  | futam1 p (x, y::ys) = if p(x, y) then x :: futam1 p (y, ys) else [x]

(* maradék : ('a * 'a -> bool) -> ('a * 'a list) -> 'a list
   maradék p (x, ys) = az x::ys p-t kielégítő futama utáni maradéka *)
fun maradék p (x, [])      = []
  | maradék p (x, yys as y::ys) = if p(x, y) then maradék p (y, ys) else yys

(* futamok1 : ('a * 'a -> bool) -> 'a list -> 'a list list
   futamok1 p xs = az xs p-t kielégítő futamaiból álló lista *)
fun futamok1 p []          = []
  | futamok1 p (x::xs) =
    let val fs = futam1 p (x, xs)
        val ms = maradék p (x, xs)
    in
      if null ms then [fs] else fs :: futamok1 p ms
    end;
```


Példák listák használatára: futamok előállítása (folyt.)

● Példák:

```
futam1   op<= (1, [9, 19, 3, 4, 24, 34, 4, 11, 45, 66, 13, 45, 66, 99]);
maradek  op<= (1, [9, 19, 3, 4, 24, 34, 4, 11, 45, 66, 13, 45, 66, 99]);
futamok1 op<= [1, 9, 19, 3, 4, 24, 34, 4, 11, 45, 66, 13, 45, 66, 99];
futamok1 op<= [99, 1];
futamok1 op<= [99];
futamok1 op<= [];
```

● Hatékonyságot rontó tényezők

1. `futamok1` kétszer megy végig a listán: először `futam1`, azután `maradek`,
2. `p-t` paraméterként adjuk át `futam1`-nek és `maradek`-nak,
3. egyik függvény sem használ akkumulátort.

● Javítási lehetőség

1. `futam2` egy párt adjon eredményül, ennek első tagja legyen a futam, második tagja pedig a maradék; a futam elemeinek gyűjtésére használjunk akkumulátort,
2. `futam2` legyen lokális `futamok2`-n belül,

Példák listák használatára: futamok előállítása (folyt.)

• Második változat: futam és maradék előállítása egy függvénnyel

```
(* futam2 : ('a * 'a -> bool) -> ('a * 'a list) -> 'a list * 'a list
```

futam2 p (x, ys) zs = olyan pár, amelynek első tagja az x::ys p-t kielégítő első futama (prefixuma) a zs elé fűzve, második tagja pedig az x::ys maradéka

*)

```
fun futam2 p (x, []) zs = (rev(x::zs), [])
  | futam2 p (x, yys as y::ys) zs = if p(x, y)
                                   then futam2 p (y, ys) (x::zs)
                                   else (rev(x::zs), yys);
```

```
(* futamok2 : ('a * 'a -> bool) -> 'a list -> 'a list list
   futamok2 p xs = az xs p-t kielégítő futamaiból álló lista
```

*)

```
fun futamok2 p [] = []
  | futamok2 p (x::xs) =
    let val (fs, ms) = futam2 p (x, xs) []
    in
      if null ms then [fs] else fs :: futamok2 p ms
    end;
```

Példák listák használatára: futamok előállítása (folyt.)

• Harmadik változat: futam és maradék előállítása egy lokális függvénnyel

```
(* futamok3 : ('a * 'a -> bool) -> 'a list -> 'a list list
   futamok3 p xs = az xs p-t kielégítő futamaiból álló lista
*)
fun futamok3 p []          = []
  | futamok3 p (x::xs) =
    let (* futam : ('a * 'a list) -> 'a list * 'a list
        futam (x, ys) zs = olyan pár, amelynek első tagja az x::ys p-t
                           kielégítő első futama (prefixuma) a zs elé
                           fűzve, második tagja pedig az x::ys maradéka
        *)
        fun futam (x, []) zs          = (rev(x::zs), [])
          | futam (x, yys as y::ys) zs = if p(x, y)
                                         then futam (y, ys) (x::zs)
                                         else (rev(x::zs), yys);
        val (fs, ms) = futam (x, xs) []
    in
        if null ms then [fs] else fs :: futamok3 p ms
    end;
```

Példák listák használatára: futamok előállítása (folyt.)

- Negyedik változat: az egyes futamokat és a futamok listáját is gyűjtjük

```
(* futamok4 : ('a * 'a -> bool) -> 'a list -> 'a list list
   futamok4 p xs = az xs p-t kielégítő futamaiból álló lista
*)
fun futamok4 p []          = []
  | futamok4 p (x::xs) =
    let (* futamok : ('a * 'a list) -> 'a list -> 'a list * 'a list
        futamok (x, ys) zs zss = az x::ys p-t kielégítő futamaiból álló
                                lista zss elé fűzve
        *)
        fun futamok (x, []) zs zss          = rev(rev(x::zs)::zss)
          | futamok (x, yys as y::ys) zs zss =
              if p(x, y)
                then futamok (y, ys) (x::zs) zss
                else futamok (y, ys) [] (rev(x::zs)::zss)
    in
        futamok (x, xs) [] []
    end;
```