

## KIÍRÁS AZ SML-BEN

Kiírás az SML-ben FP-67

### Kiírás (folyt.)

- Példák:

```
print("alma^Korte\n");
almaKorte
> val it = () : unit

makestring ~5.8e~3;
> val it = "~0.0058" : string

printVal("alma^Korte\n");
"almaKorte\n"> val it = "almaKorte\n" : string

makestring("alma^Korte\n");
> val it = "\"almaKorte\\n\"" : string
```

- `printVal`-lal tetszőleges típusú érték írható ki, például ennes, rekord vagy lista:

```
printVal (3, 5.0);
(3, 5.0)> val it = (3, 5.0) : int * real

printVal {m2 = 3, m1 = 5.0};
{m1 = 5.0, m2 = 3}> val it = {m1 = 5.0, m2 = 3} : {m1 : real, m2 : int}

printVal [#"A",#"Z",#":"];
[#"A", #"Z", #":"]> val it = [#"A", #"Z", #":"] : char list
```

## Kiírás

- `{TextIO.}print` : `string -> unit`  
`print s` = kiírja az `s` értékét a standard kimenetre, és azonnal kiüríti a puffert.
- `{General.}makestring` : `numtxt-> string` (Csak mosml!)  
`makestring v` = eredménye a `v` érték *ábrázolása*.
- `{Meta.}printVal` : `'a -> 'a` (Csak mosml! Az Alice-ben: `'a -> unit`)  
`printVal e` = kiírja az `e` kifejezés értékét a standard kimenetre pontosan úgy, ahogy az értelmező írja ki a „legfelső szinten”, és azonnal kiüríti a puffert. Eredményül visszaadja az `e` kifejezés értékét. *Csak interaktív módban használható.*
  1. megjegyzés. A `{ és }` kapcsos zárójelek között opcionális modulnév áll. Pl. `{TextIO.}print` azt jelenti, hogy a függvény a `TextIO` modulban van definiálva, de az SML-értelmező a `print` nevet rövid alakban is felismeri.
  2. megjegyzés. `numtxt = int | real | word | word8 | char | string`
- Különböző típusú egyszerű értékeket alakítanak át füzérré a `toString` függvények:

<code>Bool.toString</code> : <code>bool -&gt; string</code>	<code>String.toString</code> : <code>string -&gt; string</code>
<code>Char.toString</code> : <code>char -&gt; string</code>	<code>Time.toString</code> : <code>time -&gt; string</code>
<code>Date.toString</code> : <code>date -&gt; string</code>	<code>Word.toString</code> : <code>word -&gt; string</code>
<code>Int.toString</code> : <code>int -&gt; string</code>	<code>Word8.toString</code> : <code>word8 -&gt; string</code>
<code>Real.toString</code> : <code>real -&gt; string</code>	

Deklaratív programozás. BME VIK, 2006. őszi félév

(Függvényes programozás)

### Kiírás (folyt.)

- Hosszú lista, ill. egymásba skatulyázott adatszerkezetek esetén `printVal` (és maga az mosml-értelmező is) alapesetben csak az első 200 listaelemet, ill. legfeljebb 20 szintet ír ki. A hosszát a `printLength`, a szintek számát a `printDepth` *frissíthető változó* szabályozza. Mindkét érték felülírható.

```
printLength : int ref      printLength := 7; !printLength;
printDepth  : int ref      printDepth  := 3; !printDepth;
```

- Példák:

```
printLength := 6; printVal [1,2,3,4,5,6,7,8] before print "\n";
[1, 2, 3, 4, 5, 6, ...]
> val it = [1, 2, 3, 4, 5, 6, ...] : int list
```

```
printDepth := 4; printVal fa before print "\n";
B(B(#, #), B(#, #))
> val it = B(B(#, #), B(#, #)) : t
```

- Figyelem: a `printLength` és a `!printLength` kifejezések különböznek!

```
printLength;          | !printLength;
> val it = ref 7 : int ref | > val it = 7 : int
```

## Elágazó rekurzió

- Korábban lineáris-rekurzív, ill. lineáris-iteratív folyamatokra láttunk példákat (faktoriális kiszámítása kétféleképpen).
- Most *elágazó rekurzióra* nézzünk példát: állítsuk elő a Fibonacci-számok sorozatát.
- Egy Fibonacci-számot az előző két Fibonacci-szám összege adja:
 
$$0, 1, 1, 2, 3, 5, 8, 13, 21, \dots$$
- A Fibonacci-számok matematikai definíciója könnyen átírható SML-függvénnyé:

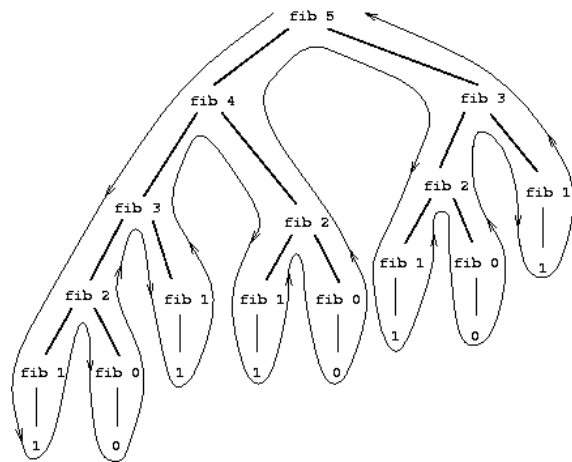
$$\begin{array}{l|l}
 F(0) = 0 & \text{fun fib 0} = 0 \\
 F(1) = 1 & \quad | \text{ fib 1} = 1 \\
 F(n) = F(n-1) + F(n-2), \text{ ha } n > 1 & \quad | \text{ fib } n = \text{fib}(n-1) + \text{fib}(n-2)
 \end{array}$$

Emlékeztetőül: a `fib` függvény definíciójában a 3. klónnak az utolsónak kell lennie, mert az `n` minta minden argumentumra illeszkedik.

- A következő lapon látható ábra illusztrálja az elágazóan rekurzív folyamatot `fib 5` kiszámítása esetén.

## ABSZTRAKCIÓ FÜGGVÉNYEKSEL (ELJÁRÁSOKKAL)

### Elágazó rekurzió (folyt.)



- `fib 5`-öt `fib 4` és `fib 3`, `fib 4`-et `fib 3` és `fib 2` kiszámításával stb. kapjuk.

### Elágazó rekurzió (folyt.)

- Az előző program alkalmas az elágazó rekurzió lényegének bemutatására, de szinte alkalmatlan a Fibonacci-számok előállítására!
- Vegyük észre, hogy pl. `fib 3`-at kétszer is kiszámítjuk, azaz a munkának ezt a részét kb. a harmadát feleslegesen végezzük el.
- Belátható, hogy  $F(n)$  meghatározásához pontosan  $F(n+1)$  levélből álló fát kell bejárni, azaz ennyiszor kell meghatározni  $F(0)$ -t vagy  $F(1)$ -et.
- $F(n)$  exponenciálisan nő  $n$ -nel.  
Pontosabban,  $F(n)$  a  $\Phi^n / \sqrt{5}$ -höz közel eső egész, ahol  $\Phi = (1 + \sqrt{5})/2 \approx 1.61803$ , az ún. *aranymetszés* arányszáma.  $\Phi$  kielégíti a  $\Phi^2 - \Phi - 1 = 0$  egyenletet.
- A megteendő lépések száma tehát  $F(n)$ -nel együtt exponenciálisan nő  $n$ -nel. Ugyanakkor a tárigény csak lineárisan nő  $n$ -nel, mert csak azt kell nyilvántartani, hogy hányadik szinten járunk a fában.
- Általában is igaz, hogy elágazó rekurzió esetén a lépések száma a fa csomópontjainak a számával, a tárigény viszont a fa maximális mélységével arányos.

## Elágazó rekurzió (folyt.)

- A Fibonacci-számok lineáris-iteratív folyamattal is előállíthatók.

Ha az  $a$  és  $b$  változók kezdőértéke rendre  $F(1) = 1$  és  $F(0) = 0$ , és ismétlődően alkalmazzuk az  $a \leftarrow a + b$ ,  $b \leftarrow a$  transzformációkat, akkor  $n$  lépés után  $a = F(n + 1)$  és  $b = F(n)$  lesz. Az iteratív folyamatot létrehozó SML-függvény egy változata:

```
fun fib n = let fun fibIter (i, b, a) =
                if i = n then b
                else fibIter(i+1, a, a+b)
            in
                fibIter(0, 0, 1)
            end
```

- *Mintaillesztést* használhatunk, ha  $i$ -t nem növeljük, hanem  $n$ -től 0-ig csökkentjük.

Figyelem: a klózok sorrendje, mivel nem egymást kizáróak a minták, lényeges!

```
fun fib n = let fun fibIter (0, b, a) = b
                | fibIter (i, b, a) = fibIter(i-1, a, a+b)
            in
                fibIter(n, 0, 1)
            end
```

## Elágazó rekurzió (folyt.): pénzváltás

Tegyük föl, hogy  $n$  darab érme áll a rendelkezésünkre valamilyen (pl. nagyság szerint csökkenő) sorrendben. Ekkor az  $a$  összeg lehetséges felváltásainak számát úgy kapjuk meg, hogy

- kiszámoljuk, hogy az  $a$  összeg hányféleképpen váltható fel az első ( $d$  értékű) érmét kivéve a többi érmével, és ehhez
- hozzáadjuk, hogy az  $a - d$  összeg hányféleképpen váltható fel az összes érmével, az elsőt is beleértve – más szóval azt, hogy az  $a$  összeget hányféleképpen tudjuk úgy felváltani, hogy a  $d$  érmét legalább egyszer felhasználjuk.

A feladat tehát rekurzióval megoldható, hiszen redukálható úgy, hogy kisebb összegeket kevesebb érmével kell felváltanunk. A következő alapeseteket különböztessük meg:

- Ha  $a = 0$ , a felváltások száma 1.  
(Ha az összeg 0, csak egyféleképpen, 0 db érmével lehet „felváltani”.)
- Ha  $a < 0$ , a felváltások száma 0.
- Ha  $n = 0$ , a felváltások száma 0.

A példában a `firstDenomination` (magyarul *első címlet*) függvényt felsorolással valósítottuk meg. Tömörebb és rugalmasabb lenne a megvalósítása lista alkalmazásával.

## Elágazó rekurzió (folyt.)

- A Fibonacci-példában a lépések száma elágazó rekurzióval tehát  $n$ -nel exponenciálisan, lineáris rekurzióval  $n$ -nel arányosan nőtt, kis  $n$ -ekre is hatalmas a nyereség!
- Téves lenne azonban azt a következtetést levonni, hogy az elágazó rekurzió használhatatlan. Amikor hierarchikusan strukturált adatokon kell műveleteket végezni, pl. egy fát kell bejárni, akkor az elágazó rekurzió (angolul: *tree recursion*) nagyon is természetes és hasznos eszköz.
- Az elágazó rekurzió numerikus számításoknál az algoritmus első megfogalmazásakor is hasznos lehet: gondoljunk csak arra, hogy milyen könnyű volt átírni a Fibonacci-számok matematikai definícióját programmá.
- Ha már értjük a feladatot, az első, rossz hatékonyságú változatot könnyebb átírni jó, hatékony programmá. Az elágazó rekurzió segíthet a feladat megértésében.

Az iteratív Fibonacci-algortimushoz csak egy aprócska ötlet kellett. A következő feladatra azonban nem lenne könnyű iteratív algoritmust írni.

- Hányféleképpen lehet felváltani egy dollárt 50, 25, 10, 5 és 1 centesekre?
- Általánosabban: adott összeget adott érmékkel hányféleképpen lehet felváltani?

## Elágazó rekurzió (folyt.): pénzváltás

```
fun countChange amount =
  let (* cC amount kindsOfCoins = az amount összes felváltásainak száma
      kindsOfCoins db érmével *)
      fun cC (amount, kindsOfCoins) =
          if amount < 0 orelse kindsOfCoins = 0 then 0
          else if amount = 0 then 1
          else cC (amount, kindsOfCoins - 1) +
              cC (amount - firstDenomination kindsOfCoins, kindsOfCoins)
      and firstDenomination 1 = 1
        | firstDenomination 2 = 5
        | firstDenomination 3 = 10
        | firstDenomination 4 = 25
        | firstDenomination 5 = 50
    in
        cC(amount, 5)
    end;
```

```
countChange 10 = 4; countChange 100 = 292;
```

*Gyakorló feladatok.*

- Írja át a `firstDenomination` függvényt úgy, hogy a címleteket egy lista tartalmazza.
- Írja meg a `cC` függvény mintaillesztést használó változatát!

## Hatványozás

- Az eddig látott folyamatokban a kiértékelési (végrehajtási) lépések száma az adatok  $n$  számával lineárisan, ill. exponenciálisan nőtt. Most olyan példa következik, amelyben a lépések száma az  $n$  logaritmusával arányos.

- A  $b$  szám  $n$ -edik hatványának definícióját ugyancsak könnyű átrakni SML-be.

$$b^0 = 1 \quad \left| \begin{array}{l} \text{fun expt } (b, 0) = 1 \\ b^n = b \cdot b^{n-1} \quad \left| \text{expt } (b, n) = b * \text{expt}(b, n-1) \end{array} \right.$$

- A létrejövő folyamat lineáris-rekurzív.  $O(n)$  lépés és  $O(n)$  méretű tár kell a végrehajtásához.
- A faktoriálisszámításhoz hasonlóan könnyű felírni lineáris-iteratív változatát.

```
fun expt (b, n) =
  let fun exptIter (0, product) = product
      | exptIter (counter, product) =
          exptIter (counter-1, b * product)
  in
    exptIter(n, 1)
  end
```

- $O(n)$  lépés és  $O(1)$  – azaz konstans – méretű tár kell a végrehajtásához.

## LISTÁK

## Hatványozás (folyt.)

- Kevesebb lépés is elég, ha kihasználjuk az alábbi egyenlőségeket:

$$b^0 = 1$$

$$b^n = (b^{n/2})^2, \text{ ha } n \text{ páros}$$

$$b^n = b \cdot b^{n-1}, \text{ ha } n \text{ páratlan}$$

```
fun expt (b, n) =
  let fun exptFast 0 = 1
      | exptFast n =
          if even n
          then square(exptFast(n div 2))
          else b * exptFast(n-1)
  and even i = i mod 2 = 0
  and square x = x * x
  in exptFast n end
```

- A lépések száma és a tár mérete  $O(\lg n)$ -nel arányos. Konstans tárigenyű iteratív változata:

```
fun expt (b, 0) = 1 (* Nem hagyható el! Miért nem? *)
  | expt (b, n) = let fun exptFast (1, r) = r
                    | exptFast (n, r) =
                        if even n then exptFast(n div 2, r*r)
                        else exptFast(n-1, r*b)
                    and even i = i mod 2 = 0
                    in exptFast(n, b) end
```

## Adott számú elem egy lista elejéről és végéről (take, drop)

- Legyen  $xs = [x_0, x_1, \dots, x_{i-1}, x_i, x_{i+1}, \dots, x_{n-1}]$ , akkor  $\text{take}(xs, i) = [x_0, x_1, \dots, x_{i-1}]$  és  $\text{drop}(xs, i) = [x_i, x_{i+1}, \dots, x_{n-1}]$ .
- $\text{take}$  egy megvalósítása (jobbrekurzív-e? jobbrekurzívvá tehető-e? robusztus-e?)

```
(* take : 'a list * int -> 'a list
   take (xs, i) = ha i < 0, xs; ha i >= 0,
                   az xs első i db eleméből álló lista *)
fun take (_, 0) = []
  | take ([], _) = []
  | take (x::xs, i) = x :: take(xs, i-1)
```

- $\text{drop}$  egy megvalósítása (jobbrekurzív-e? jobbrekurzívvá tehető-e? robusztus-e?)

```
(* drop : 'a list * int -> 'a list
   drop (xs, i) = ha i < 0, xs; ha i >= 0,
                   az xs első i db elemének eldobásával előálló lista *)
fun drop ([], _) = []
  | drop (x::xs, i) = if i > 0 then drop (xs, i-1) else x::xs
```

- Könyvtári változatuk – `List.take`, ill. `List.drop` – ha az  $xs$  listára alkalmazzuk,  $i < 0$  vagy  $i > \text{length } xs$  esetén `Subscript` néven kivételt jelez.

## Lista redukciója kétoperandusú művelettel

Idézzük föl az egészlista maximális értékét megkereső `maxl` függvény két változatát:

- `maxl` jobbról balra egyszerűsítő (nem jobbrekurzív) változata

```
(* maxl : int list -> int
   maxl ns = az ns egészlista legnagyobb eleme
*)
fun maxl [] = raise Empty
  | maxl [n] = n
  | maxl (n::ns) = Int.max(n, maxl ns)
```

- `maxl` balról jobbra egyszerűsítő (jobbrekurzív) változata:

```
(* maxl' : int list -> int
   maxl' ns = az ns egészlista legnagyobb eleme
*)
fun maxl' [] = raise Empty
  | maxl' [n] = n
  | maxl' (n::m::ns) = maxl' (Int.max(n,m)::ns)
```

- Amint ez a példa is mutatja, vissza-visszatérő feladat egy lista redukciója kétoperandusú művelettel.
- Közös bennük, hogy  $n$  db értékből egyetlen értéket kell előállítani, ezért is beszélünk *redukcióról*.

## Példák `foldr` és `foldl` alkalmazására

- `isum` egy egészlista elemeinek összegét, `rprod` egy valóslista elemeinek szorzatát adja eredményül.

```
val isum = foldr op+ 0;           val rprod = foldr op* 1.0;
val isum = foldl op+ 0;           val rprod = foldl op* 1.0;
```

- A `length` függvény is definiálható `foldl`-lel vagy `foldr`-rel. Kétoperandusú műveletként olyan segédfüggvényt (`inc`) alkalmazunk, amelyik *nem használja* az első paraméterét.

```
(* inc : 'a * int -> int
   inc (_, n) = n + 1 *)
fun inc (_, n) = n + 1;

(* lengthr, lengthl : 'a list -> int *)
val lengthr = fn ls => foldr inc 0 ls;
fun lengthr ls = foldr inc 0 ls;
val lengthl = foldr inc 0;
val lengthl = foldr (fn (_,n) => n+1) 0;

val lengthr = fn ls => foldl inc 0 ls;
fun lengthl ls = foldl inc 0 ls;
val lengthr = foldl inc 0;
val lengthr = foldl (fn (_,n) => n+1) 0;

lengthr (explode "hajdu sogor");
lengthl (explode "tengertanc");
```

## Lista redukciója kétoperandusú művelettel (`foldr`, `foldl`)

- `foldr` jobbról balra, `foldl` balról jobbra haladva egy kétoperandusú műveletet (pontosabban egy *párra alkalmazható, prefix* pozíciójú függvényt) alkalmaz egy listára. Példák szorzat és összeg kiszámítására:

```
foldr op* 1.0 [] = 1.0;           foldl op+ 0 [] = 0;
foldr op* 1.0 [4.0] = 4.0;       foldl op+ 0 [4] = 4;
foldr op* 1.0 [1.0, 2.0, 3.0, 4.0] = 24.0;
foldl op+ 0 [1, 2, 3, 4] = 10;
```

- Jelöljön  $\oplus$  tetszőleges kétoperandusú infix operátort. Akkor

```
foldr op⊕ e [x1, x2, ..., xn] = (x1 ⊕ (x2 ⊕ ... ⊕ (xn ⊕ e) ...))
foldr op⊕ e [] = e
foldl op⊕ e [x1, x2, ..., xn] = (xn ⊕ ... ⊕ (x2 ⊕ (x1 ⊕ e)) ...)
foldl op⊕ e [] = e
```

- A  $\oplus$  művelet  $e$  operandusa néhány gyakori műveletben – összeadás, szorzás, konjunkció (logikai „és”), alternáció (logikai „vagy”) – a (jobb oldali) *egységelem* szerepét tölti be.
- Ha a művelet asszociatív és kommutatív, `foldr` és `foldl` eredménye azonos.

## Lista: `foldr` és `foldl` definíciója

- $\text{foldr } op \oplus e [x_1, x_2, \dots, x_n] = (x_1 \oplus (x_2 \oplus \dots \oplus (x_n \oplus e) \dots))$   
 $\text{foldr } op \oplus e [] = e$

(\* `foldr f e xs` = az `xs` elemeire jobbról balra haladva alkalmazott, kétoperandusú,  $e$  egységelemű  $f$  művelet eredménye

```
foldr : ('a * 'b -> 'b) -> 'b -> 'a list -> 'b *)
fun foldr f e (x::xs) = f(x, foldr f e xs)
  | foldr f e [] = e;
```

- $\text{foldl } op \oplus e [x_1, x_2, \dots, x_n] = (x_n \oplus \dots \oplus (x_2 \oplus (x_1 \oplus e) \dots))$   
 $\text{foldl } op \oplus e [] = e$

(\* `foldl f e xs` = az `xs` elemeire balról jobbra haladva alkalmazott, kétoperandusú,  $e$  egységelemű  $f$  művelet eredménye

```
foldl : ('a * 'b -> 'b) -> 'b -> 'a list -> 'b *)
fun foldl f e (x::xs) = foldl f (f(x, e)) xs
  | foldl f e [] = e;
```

További példák `foldr` és `foldl` alkalmazására

- Egy lista elemeit egy másik lista elé fűzi `foldr` és `foldl`, ha kétoperandusú műveletként a `cons` konstruktorfüggvényt – azaz az `op::-`-ot – alkalmazzuk.

```
foldr op:: ys [x1, x2, x3] = (x1 :: (x2 :: (x3 :: ys)))
```

```
foldl op:: ys [x1, x2, x3] = (x3 :: (x2 :: (x1 :: ys)))
```

- A `::` nem asszociatív és kommutatív, ezért `foldl` és `foldr` eredménye különböző!

```
(* append : 'a list -> 'a list -> 'a list
   append xs ys = az xs ys elé fűzésével előálló lista *)
fun append xs ys = foldr op:: ys xs;
```

```
(* revApp : 'a list -> 'a list -> 'a list
   revApp xs ys = a megfordított xs ys elé fűzésével
                 előálló lista *)
fun revApp xs ys = foldl op:: ys xs;
```

```
append [1, 2, 3] [4, 5, 6] = [1, 2, 3, 4, 5, 6]; (vö. Prolog: append)
```

```
revApp [1, 2, 3] [4, 5, 6] = [3, 2, 1, 4, 5, 6]; (vö. Prolog: revapp)
```

## Példa listák használatára: futamok előállítása

- A futam egy olyan lista, amelynek szomszédos elemei egy adott feltételnek megfelelnek.
- Az adott feltételt az előző és az aktuális elemre alkalmazandó *predikátumként* adjuk át a futamot előállító függvénynek.
- A feladat: írjunk olyan SML függvényt, amely (az elemek eredeti sorrendjének megőrzésével) egy lista egymás utáni elemeiből képzett futamok listáját adja eredményül.
- Az első változatban egy-egy segédfüggvényt írunk egy lista első (prefix) futamának, valamint a maradéklistának az előállítására.
- A futam segédfüggvénynek két argumentuma van: az első egy predikátum, amely a kívánt feltételt megvalósítja, a második pedig egy pár. A pár első tagja az előző elem, a második tagja pedig az a lista, amelynek az előző elemmel induló futamát kell futam-nak előállítania.
- A maradék segédfüggvény két argumentuma azonos futam argumentumaival. Eredményül azt a listát kell visszaadnia, amelyet az első futam leválasztásával állít elő a pár második tagjaként átadott listából.
- A következő diákon a futam és a maradék segédfüggvények, valamint a futamok függvény különböző változatai láthatók.

További példák `foldr` és `foldl` alkalmazására

- `maxl` két megvalósítása

```
(* maxl : ('a * 'a -> 'a) -> 'a list -> 'a
   maxl max ns = az ns lista max szerinti legnagyobb eleme *)
```

```
(* nem jobbrekurzív *)
```

```
fun maxl max [] = raise Empty
  | maxl max (n::ns) = foldr max n ns
```

```
(* jobbrekurzív *)
```

```
fun maxl' max [] = raise Empty
  | maxl' max (n::ns) = foldl max n ns
```

## Példa listák használatára: futamok előállítása (folyt.)

- Első változat: futam és maradék előállítása két függvénnyel

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

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

## Példa listák használatára: futamok előállítása (folyt.)

## • Hatékonyságot rontó tényezők

1. futamok1 kétszer megy végig a listán: először futam, azután maradék,
2. p-t, bár sohasem változik, paraméterként adjuk át futam-nak és maradék-nak,
3. egyik függvény sem használ akkumulátort.

## • Javítási lehetőségek

1. futam egy párt adjon eredményül, ennek első tagja legyen a futam, második tagja pedig a maradék; a futam elemeinek gyűjtésére használjunk akkumulátort,
2. futam legyen lokális futamok2-n belül,
3. az if null ms then [fs] else szövegrész törölhető: a rekurzió egy hívással később mindenképpen leáll.

## Példa listák használatára: futamok előállítása (folyt.)

## • Harmadik változat: az egyes futamokat és a futamok listáját is gyűjtjük

```
(* 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 (* 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;
```

## Példa listák használatára: futamok előállítása (folyt.)

## • Második változat: futam és maradék előállítása egy lokális függvénnyel

```
(* 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 (* futam : ('a * 'a list) -> 'a list * 'a list
        futam (x, ys) zs = olyan pár, amelynek első tagja az x::ys p-t
                            kielégítő első (prefix) futama 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);
    in
      val (fs, ms) = futam (x, xs) []
    in
      fs :: futamok2 p ms
    end
```

## Példa listák használatára: futamok előállítása (folyt.)

## • Példák a függvények alkalmazására:

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

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

futamok1 op<= [99,1] = [[99], [1]];

futamok1 op<= [99] = [[99]];

futamok1 op<= [] = [];
```

## ABSZTRAKCIÓ ADATOKKAL

### Adatabsztrakció: racionális számok (folyt.)

- Tegyük föl, hogy
  - van olyan *konstruktorműveletünk*, amely egy  $n$  számlálóból és egy  $d$  nevezőből létrehozza a racionális számot: `makeRat (n, d)`, továbbá
  - van egy-egy olyan *szelektorműveletünk*, amelyek egy  $q$  racionális szám számlálóját, ill. nevezőjét előállítják: `num q, den q`.

- A jól ismert műveleteket írjuk át SML-programmá:

$$n_1/d_1 + n_2/d_2 = (n_1d_2 + n_2d_1)/(d_1d_2), \quad n_1/d_1 - n_2/d_2 = (n_1d_2 - n_2d_1)/(d_1d_2),$$

$$(n_1/d_1)(n_2/d_2) = (n_1n_2)/(d_1d_2), \quad (n_1/d_1)/(n_2/d_2) = (n_1d_2)/(d_1n_2),$$

$$n_1/d_1 = n_2/d_2 \text{ akkor és csak akkor, ha } n_1d_2 = n_2d_1.$$

```

fun addRat(x, y) =
  makeRat(num x * den y + num y * den x, den x * den y)
fun subRat(x, y) =
  makeRat(num x * den y - num y * den x, den x * den y)
fun mulRat(x, y) = makeRat(num x * num y, den x * den y)
fun divRat(x, y) = makeRat(num x * den y, den x * num y)
fun equRat(x, y) = num x * den y = den x * num y

```

### Adatabsztrakció: racionális számok

- A következő előadásokon összetett adatokkal és adatabsztrakcióval foglalkozunk.
- Az adatabsztrakció lényege: összetett adatokkal dolgozó programjainkat úgy építjük föl, hogy
  - az adatokat felhasználó programrészek az adatok szerkezetéről ne tétélezzenek fel semmit, csak az előre definiált műveleteket használják,
  - az adatokat definiáló programrészek az adatokat felhasználó programrészektől függetlenek legyenek.
  - A program e két része közötti interfész *konstruktorokból* és *szelektorokból* áll.
- Az összetett adatok közül eddig ennesekkel, rekordokkal és listákkal találkoztunk.
- Első nagyobb példánkban a racionális számok és a rajtuk végezhető műveletek megvalósítását mutatjuk be.
- A racionális számot ábrázolhatjuk egy olyan párral, amelynek az első tagja a számláló (*numerator*) és a második a nevező (*denominator*).
- Megvalósítjuk a négy aritmetikai alpműveletet: `addRat`, `subRat`, `mulRat`, `divRat`, továbbá az egyenlőségvizsgálatot: `equRat`.

### Adatabsztrakció: racionális számok (folyt.)

- Az SML-ben az *ennes* létrehozására van *konstruktorműveletünk*: a tagokat kerek zárójelek között, vesszővel elválasztva felsoroljuk, és
- van az *ennes* egy-egy tagját kiválasztó *szelektorműveletünk*: `# i`, ahol  $i$  az  $i$ -edik tag *pozicionális címkeje*, 1-től kezdve.
- Példák: `(3, 4)`; `#1(3, 4)`; `#2(3, 4)`;
- Az *ennes* tagjai *mintaillesztéssel* is köthetők névhez, pl. `val (n, d) = (3, 4)`.
- *Gyenge* absztrakcióval valósítjuk meg a típust, a konstruktort és szelektorokat:
 

```

type rat = int * int;
fun makeRat (n, d) = (n, d) : rat;
fun num (q : rat) = #1 q;
fun den q = #2 (q : rat);

```
- A *gyenge absztrakció* nevet ad egy objektumnak, de *nem rejti el* a megvalósítás részleteit.
- Szükségünk lesz kiíróműveletre is az  $n/d$  alakú racionális szám kiírásához.
 

```

fun printRat q =
  print(makestring(num q) ^ "/" ^ makestring(den q) ^ "\n");

```



## Adatabsztrakció: racionális számok (folyt.)

- Ezzel racionális számokat megvalósító programunk első változata kész is van. A teljes program:

```

type rat = int * int;
fun makeRat (n, d) = (n, d) : rat;
fun num (q : rat) = #1 q;
fun den q = #2 (q : rat);

fun addRat(x, y) =
  makeRat(num x * den y + num y * den x, den x * den y)
fun subRat(x, y) =
  makeRat(num x * den y - num y * den x, den x * den y)
fun mulRat(x, y) = makeRat(num x * num y, den x * den y)
fun divRat(x, y) = makeRat(num x * den y, den x * num y)
fun equRat(x, y) = num x * den y = den x * num y

fun printRat q =
  print(makestring(num q) ^ "/" ^ makestring(den q) ^ "\n");

```

## Adatabsztrakció: racionális számok (folyt.)

- Az utolsó példából, ha kipróbáljuk, láthatjuk, hogy programunk nem *normalizálja*, azaz nem a lehető legegyszerűbb alakban tárolja, ill. írja ki a racionális számokat.
- Segíthetünk a dolgon, ha a konstruktorműveletben a számlálót és a nevezőt a legnagyobb közös osztóval elosztjuk; a szelektorműveleteken nem változtatunk:

```

(* gcd : int * int -> int
   gcd (a, b) = a és b legnagyobb közös osztója
*)
fun gcd (a, 0) = a
  | gcd (a, b) = gcd(b, a mod b)

fun makeRat (n, d) =
  let val g = gcd(n, d) in (n div g, d div g) : rat end;

```

- A racionális számokat normalizált alakjukban tároljuk, ezért nemcsak a kiírás, hanem az egyenlőségvizsgálat is helyes eredményt ad:

```

printRat(addRat(oneThird, oneThird));
addRat(oneThird, oneThird) = twoThird;

```

- A normalizáláshoz csak egyetlen helyen kellett változtatni a programon!

## Adatabsztrakció: racionális számok (folyt.)

- Néhány példa a program használatára:

```

val oneHalf = makeRat(1,2);
val oneThird = makeRat(1,3);
val twoThird = makeRat(2,3);

printRat oneHalf;
printRat(addRat(oneHalf, oneThird));
printRat(mulRat(oneHalf, oneThird));
printRat(addRat(oneThird, oneThird));

equRat(addRat(oneThird, oneThird), twoThird);

oneThird = oneThird;
addRat(oneThird, oneThird) = twoThird;

```

## Adatabsztrakció: racionális számok (folyt.)

*Adatabsztrakciós korlátok* a racionális számok csomagban

```

-----
Racionális számot használó programok
-----
Racionális szám a feladattérben
-----
addRat subRat mulRat divRat equRat
-----
Racionális szám mint számláló és nevező
-----
konstruktor: makeRat; szelektorok: num, den
-----
Racionális szám mint pár
-----
konstruktor: ( , ) ; szelektorok: #1, #2
-----
A pár megvalósítása SML-ben

```

## Adatabsztrakció: racionális számok (folyt.)

- Az absztrakciós korlátok elszigetelik egymástól a program egyes részeit.
- Előnye, hogy a programokat egyszerűbb karbantartani és módosítani, pl. az adatok ábrázolását megváltoztatni.
- Pl. a racionális szám normalizálható a létrehozása helyett akkor, amikor a számlálójára vagy a nevezőjére van szükségünk. Ha gyakran hozunk létre racionális számokat, de csak ritkán használjuk a számlálóját vagy a nevezőjét, akkor az utóbbi megoldás a hatékonyabb.

```
fun num (q : rat) =
  let val (n, d) = q; val g = gcd(n, d) in n div g end;
fun den (q : rat) =
  let val (n, d) = q; val g = gcd(n, d) in d div g end;
```

- A makeRat függvény nem normalizáló változatát használjuk; a program többi része nem változik.

```
printRat(addRat(oneThird, oneThird));
addRat(oneThird, oneThird) = twoThird;
equRat(addRat(oneThird, oneThird), twoThird) = true;
```

## Adatabsztrakció: racionális számok (folyt.)

- Bármely megvalósítás, amely ezeket a feltételeket kielégíti, megfelel, például a következő is:

```
exception Cons of string;
fun cons (x, y) =
  let fun dispatch 0 = x
      | dispatch 1 = y
      | dispatch _ = raise Cons "argument not 0 or 1"
  in dispatch
  end;
fun fst z = z 0;
fun snd z = z 1;
```

- A tulajdonságleíró egyenletek

```
q = cons(n, d)
n = fst q
d = snd q
```

- Vegyük észre, hogy a racionális számot megvalósító cons objektum: *függvény!* fst és snd *üzenetet küld* az objektumnak. Ennek a programozási stílusnak ezért *üzenetküldés* a neve.

## Adatabsztrakció: racionális számok (folyt.)

- *Adatokról* szólva nem elég annyit mondanunk, hogy „adat az, amit az adott konstruktorok és szelektorok megvalósítanak”.
- Nyilvánvaló, hogy konstruktorok és szelektorok csak bizonyos halmaza alkalmas pl. a racionális számok megvalósítására.
- Racionális számok esetén a konstruktornak és a szelektoroknak garantálniuk kell az alábbi feltételek (axiómák) teljesülését:

```
(* PRE : d > 0 *)
x = makeRat(n, d);
n = num x
d = den x
```

- Egygel alacsonyabb absztrakciós szinten a pár-ábrázolásnak is ki kell elégítenie a következő feltételeket:

```
q = (x, y)
x = #1 q
y = #2 q
```

## Adatabsztrakció: racionális számok (folyt.)

- Példa:

```
val q = cons(1, 2);
fst q = 1; snd q = 2;
```

- A racionális számok konstruktorának és a szelektorainak megvalósítása üzenetküldéssel:

```
fun makeRat (n, d) =
  let val g = gcd(n, d) in cons(n div g, d div g) end;
fun num q = fst q;
fun den q = snd q;
```

- Racionális számokat megvalósító csomagunk nagy hibája, hogy *gyenge absztrakciót* valósít meg, azaz nem rejti el a megvalósítás részleteit; a programozóra bízta, hogy az absztrakciós korlátokat milyen mértékben tartja be. Ez hibák forrása.
- A megvalósítás részleteit *erős absztrakcióval*, modulok segítségével rejthetjük el a külvilág elől. Az „implementációs” modul neve az SML-ben: *structure*, az (opcionális) „interfészmodul” neve pedig: *signature*.

```
structure name = struct ... end
signature name = sig ... end
```

## ABSZTRAKCIÓ ADATOKKAL

Absztrakció adatokkal FP-107

### A datatype deklaráció (folyt.)

```
King :    person
Peer :    string * string * int -> person
Knight :  string -> person
Peasant : string -> person
```

- King (király) csak egy van, ezért definiálhattuk konstruktorállandóként.
- A Peer-t (főnemes) nemesi címe (string), birtokának neve (string) és sorszáma (int) azonosítja.
- A Knight-ot (lovagot) és a Peasant-ot (parasztot) csupán a neve (string) azonosítja.

- Példa a person adattípus alkalmazására:

```
val persons = [King, Peasant "Jack Cade", Knight "Gawain",
               Peer("Duke", "Norfolk", 9)];

> val persons = [King, Peasant "Jack Cade", ...] : person list
```

- Az egyes esetek mintaillesztéssel választhatók szét.
- Minden esetet le kell fedni mintával; ha nem, figyelmeztetést kapunk.
- A minták tetszőlegesen összetettek lehetnek.

### Felhasználói adattípusok: ismét a datatype deklarációról

- person néven új összetett típust hozunk létre:

```
datatype person = King
                | Peer of string * string * int
                | Knight of string
                | Peasant of string
```

- Az új típusnak négy *adatkonstruktora* (röviden: *konstruktor*) van: King, Peer, Knight és Peasant.
- King ún. *adatkonstruktorállandó*, a többi ún. *adatkonstruktorfüggvény*.
- Az adatkonstruktoroknak is van típusuk:

```
King :    person
Peer :    string * string * int -> person
Knight :  string -> person
Peasant : string -> person
```

Deklaratív programozás. BME VIK, 2006. őszi félév

(Funkcionális programozás)

Absztrakció adatokkal FP-108

### A datatype deklaráció (folyt.)

- Az alábbi példában a négy közül az egyik a Peasant *name minta*, és benne *name* a *mintaazonosító*.

```
(* title : person -> string
   title p = p megszólítása *)
fun title King = "His Majesty the King "
  | title (Peer (deg, ter, _)) = "The " ^ deg ^ " of " ^ ter
  | title (Knight name) = "Sir " ^ name
  | title (Peasant name) = name
```

- A sirs függvény az összes Knight nevet összegyűjti a person típusú személyek egy listájából (a változatok sorrendje *fontos* az *\_* miatt!):

```
(* sirs : person list -> string list
   sirs ps = az összes Knight nevének listája *)
fun sirs [] = []
  | sirs ((Knight s)::ps) = s::sirs ps
  | sirs (_::ps) = sirs ps
```

## A datatype deklaráció (folyt.)

- Ha más lenne a változatok sorrendje, a `_ : ps` minta nemcsak a `King`-re, a `Peer`-re és a `Peasant`-ra illeszkedne (ti. ezek helyett áll a példában), hanem a `Knight`-ra is.
- Az összes diszjunkt eset felsorolása segíti az algoritmus helyességének belátását, bizonyítását.
- Azért vontunk össze három esetet egyetlen változatban, mert a részletezésük hosszabbá tenné a program szövegét is, végrehajtását is.
- A bizonyítás nem okoz gondot, ha a függvény harmadik sorát (`sirs (_:ps) = sirs ps`) *feltételes egyetlennek* tekintjük:

```
sirs(p::ps) = sirs ps if  $\forall s.p \neq \text{Knight } s$ .
```

## Felsorolós típus datatype deklarációval

- Gyakori, hogy egy név csak néhány különböző értéket vehet fel (azaz a név által felvehető értékek halmaza kis számosságú), ilyen esetben érdemes *felsorolós típust* létrehozni a datatype deklarációval. Pl.

```
datatype degree = Duke | Marquis | Earl | Viscount | Baron
```

- A felsorolós típusnak csak *konstruktorállandói* vannak. Az új típus alkalmazásához a `person` típust újra deklarálnunk kell:

```
datatype person = King
  | Peer of degree * string * int
  | Knight of string
  | Peasant of string
```

## A datatype deklaráció (folyt.)

- A sorrend még fontosabb a következő példában, amelyben személyek hierarchiáját vizsgáljuk. Itt 16 helyett csak 7 esetet kell megkülönböztetnünk: azokat, amelyek *igaz* eredményt adnak.

```
(* superior : person * person -> bool
   superior (p, r) = igaz, ha p magasabb rangú r-nél *)
fun superior (King, Peer _) = true
  | superior (King, Knight _) = true
  | superior (King, Peasant _) = true
  | superior (Peer _, Knight _) = true
  | superior (Peer _, Peasant _) = true
  | superior (Knight _, Peasant _) = true
  | superior _ = false
```

## Felsorolós típus datatype deklarációval (folyt.)

- A `degree` típusú adatok feldolgozásakor külön-külön elemezzük az előforduló eseteket, pl.

```
(* lady : degree -> string
   lady p = p főnemes hitvesének rangja *)
fun lady Duke = "Duchess "
  | lady Marquis = "Marchioness"
  | lady Earl = "Countess"
  | lady Viscount = "Viscountess"
  | lady Baron = "Baroness"
```

- A belső `bool` típushoz hasonló `Bool` típust és hozzá a `Not` függvényt például így is deklarálhatnánk, ill. definiálhatnánk:

```
datatype Bool = True | False
(* Not : Bool -> Bool
   Not b = b negáltja *)
fun Not True = False | Not False = True
```

## Polimorf adattípusok

- Láttuk, hogy a `list postfix` pozíciójú *típusoperátor*, nem típus: a `datatype` deklaráció az adatkonstruktorok mellett *típuskonstruktor* is létrehoz.

- A belső `'a list` típushoz hasonló `'a List` listát és vele együtt a `Nil` és a `Cons` *adatkonstruktorokat* például így definiálhatjuk:

```
datatype 'a List = Nil | Cons of 'a * 'a List;
```

- A `Cons` *adatkonstruktorfüggvény* alkalmazásával elég körülményes a listák létrehozása. Az 1, 2, 3, 4 sorozatot például így kell megadni:

```
Cons(1, Cons(2, Cons(3, Cons(4, Nil))));
```

- Bevezethetjük az *infix* pozíciójú `:::` *adatkonstruktoroperátort*:

```
infixr 5 ::: ; val op ::: = Cons
```

- A *hatospont* *infix* helyzetét magában a típusdeklarációban is definiálhatjuk:

```
infixr 5 ::: ; datatype 'a List = Nil | ::: of 'a * 'a List
```

## Megkülönböztetett egyesítés (folyt.)

- A megkülönböztetett egyesítés lehetővé teszi, hogy különböző típusokat használjunk ott, ahol egyébként csak egyetlen típust használhatnánk (vö. objektum-orientált programozás, ahol pl. *alakzat* osztálynak *téglalap*, *háromszög* vagy *kör* nevű leszármazottai lehetnek).

- Az SML-ben megkülönböztetett egyesítéssel tudunk létrehozni *különböző típusú elemekből* álló listát:

```
[In2 King, In1 "Skócia"] : ((string, person) disun) list;
[In1 "zsarnok", In2 1040] : ((string, int) disun) list
```

- A lehetséges eseteket most is *mintaillesztéssel* elemezhetjük, pl.

```
(* concat : (string, 'a) disun list -> string
   concat d = a d diszjunkt unió In1 címkéjű
               elemeinek konkatenációja *)
fun concat [] = ""
  | concat (In1 s :: ls) = s ^ concat ls
  | concat (In2 _ :: ls) = concat ls;
```

## Polimorf adattípusok: megkülönböztetett egyesítés

- Következő példánk két típus *megkülönböztetett egyesítése*, más néven diszjunkt uniója:

```
datatype ('a, 'b) disun = In1 of 'a | In2 of 'b
```

- Itt három dolgot definiáltunk:

1. a kétargumentumú `disun` típusoperátort,
2. az `In1 : 'a -> ('a, 'b) disun` és
3. az `In2 : 'b -> ('a, 'b) disun` *adatkonstruktorfüggvényeket*.

- `('a, 'b) disun` az `'a` és `'b` típusok megkülönböztetett egyesítése. *Megkülönböztetettnek* nevezzük az egyesítést, mert később is bármikor meg tudjuk mondani, hogy egy `('a, 'b) disun` típusú pár egyik vagy másik eleme melyik alaptípusból származik. Az új típusba tartozó értékek `In1 x` alakúak, ha `x 'a` típusú, és `In2 y` alakúak, ha `y 'b` típusú.

- Az `In1` és `In2` konstruktorfüggvények olyan *címkének* tekinthetők, amelyek az `'a` típust megkülönböztetik a `'b` típustól.

## Megkülönböztetett egyesítés (folyt.)

- Egy példa `concat` alkalmazására:

```
concat [In1 "Ó! ", In2 King, In1 "Skócia"];
> val it = "Ó! Skócia" : string
```

- Az `In1` konstruktorfüggvény típusa `'a -> ('a, 'b) disun`, ezért a `string` típusú `"Ó!"` argumentumra alkalmazva `(string, 'b) disun` típusú érték az eredmény.

- Az `In2` konstruktorfüggvény típusa `'b -> ('a, 'b) disun`, ezért a `person` típusú `King` kifejezésre alkalmazva `('a, person) disun` típusú érték az eredmény.

- Az `[In1 "Ó!", In2 King, In1 "Skócia"]` kifejezésben mindkét alaptípust lekötjük, ezért ennek a listának a típusa: `((string, person) disun) list`.

- Az `[In2 "Ó", In2 King, In1 "Skócia"]` kifejezés kiértékelése hibajelzést eredményez, mert a `'b` típusváltozót nem lehet ugyanabban a kifejezésben egyszer így, másszor úgy lekötöni.

## LOKÁLIS ÉRVÉNYŰ DEKLARÁCIÓ, EGYIDEJŰ DEKLARÁCIÓ

### Deklaráció lokális érvényű deklarációval: local-deklaráció (folyt.)

- Láttuk, hogy a let-kifejezést milyen jól lehet használni lokális értékek létrehozására, többek között lokális függvények definiálására.
- Amikor egy függvényt egy másik függvény *belsejében* egy let-kifejezésben definiálunk, akkor a lokális függvény törzse az őt felhasználó függvény *hatókörében* van, ezért a lokális függvény törzsében az utóbbi paramétere látszik, használható.
- Ha ellenben local-deklarációt használunk, akkor a lokális függvény törzse *nincs* az őt felhasználó függvény hatókörében, ezért a lokális függvény törzsében az utóbbi paramétere nem látszik, nem használható.
- Milyen esetben kell mégis local-deklarációt alkalmaznunk? Akkor, ha a lokális függvényt az őt felhasználó függvény *egynél több klózában* akarjuk alkalmazni. Példa:

```
local fun g x = ... x ...
in
  fun f 0 y = ... g y ...
    | f _ y = ... g y ...
end
```

- Ha ilyen esetekben let-kifejezést használnánk, akkor a lokális függvényt minden klózban újra kellene definiálnunk.

### Deklaráció lokális érvényű deklarációval: local-deklaráció

- Már megismertük a lokális deklarációt használó *kifejezést*, az ún. let-kifejezést.
- Lokális deklarációt használó *deklarációt* használunk, ha egyes deklarációkat fel akarunk használni más deklarációkban, miközben *el akarjuk rejtetni* őket a program többi része előtt.
- Szintaxisa:
 

```
local d1 ahol d1 egy nemüres deklarációsorozat,
in d2          d2 egy másik nemüres deklarációsorozat.
end
```
- Példa:

```
(* length : 'a list -> int
   length zs = a zs lista hossza *)
local
  (* len : 'a list * int -> int
     len (zs, n) = az n és a zs lista hosszának összege *)
  fun len ([], n) = n
    | len (_::zs, n) = len(zs, n+1)
in
  fun length zs = len(zs, 0)
end
```

### Egyidejű deklaráció, újra

- Egyidejű deklarációt kell használni kölcsönösen rekurzív függvények definiálására. Példa:

```
fun even 0 = true | even n = odd(n-1)
and odd 0 = false | odd n = even(n-1);
```

- Egyidejű deklarációt használhatunk két vagy több kötés egyidejű felcserélésére. Példa:

```
val v1 = "a"; val v2 = "z"; val v1 = v2 and v2 = v1;
```

- Egyidejű deklarációt használunk, ha fölülről lefelé haladva akarunk programot írni. Példa:

```
fun length zs = len zs 0
and len [] i = i | len (_::xs) i = len xs (i+1);
```

- A polimorf függvényeket a szekvenciális és az egyidejű deklaráció eltérően kezeli, mivel a típuslevelet az SML-értelmező a teljes kifejezésre alkalmazza. Példa:

```
fun id x = x; fun hi () = id 3; fun nr () = id 4.0;
fun id x = x and hi () = id 3 and nr () = id 4.0;
```

Az első sor kiértékelésekor id 'a -> 'a típusú. A második sor kiértékelésekor id int -> int és real -> real típusú lenne egyszerre, ami lehetetlen.

## ESETSZÉTVALASZTÁS, OPCIONÁLIS ÉRTÉK

### Esetsztváltás (case)

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

Az SML-értelmező – balról jobbra és fölülről lefelé haladva – megpróbálja E-t P1-re illeszteni, ha nem sikerül, P2-re s.f.t. A case-kifejezés eredménye az E kifejezésre illeszkedő első P<sub>i</sub> mintához tartozó E<sub>i</sub> kifejezés lesz.

A case is csak szintaktikus édesítőszers, ui. helyettesíthető fn-jelöléssel:

```
(fn P1 => E1 | P2 => E2 | ... | Pn => En) E
```

Például a lady függvényt így is definiálhattuk volna:

```
datatype degree = Duke | Marquis | Earl | Viscount | Baron
(* lady : degree -> string
   lady p = p főnemes
           hitvesének rangja *)
fun lady p =
  case p of
    Duke    => "Duchess "
  | Marquis => "Marchioness"
  | Earl    => "Countess"
  | Viscount => "Viscountess"
  | Baron   => "Baroness"

(* lady : degree -> string
   lady p = p főnemes
           hitvesének rangja *)
fun lady p =
  (fn
    Duke    => "Duchess "
  | Marquis => "Marchioness"
  | Earl    => "Countess"
  | Viscount => "Viscountess"
  | Baron   => "Baroness"
  ) p
```

Deklaratív programozás. BME VIK, 2006. őszi félév

(Funkcionális programozás)

Esetsztváltás, opcionális érték FP-123

### Opcionális érték kezelése ('a option)

```
datatype 'a option = NONE | SOME of 'a
```

Függvények az Option könyvtárból:

```
val getOpt      : 'a option * 'a -> 'a
val isSome     : 'a option -> bool
val valOf      : 'a option -> 'a
val filter     : ('a -> bool) -> 'a -> 'a option
val map        : ('a -> 'b) -> 'a option -> 'b option
val mapPartial : ('a -> 'b option) -> ('a option -> 'b option)
```

`getOpt (xopt, d) = x` if xopt is SOME x, d otherwise.

`isSome xopt = true` if xopt is SOME x, false otherwise.

`valOf xopt = x` if xopt is SOME x, raises Option otherwise.

`filter p x = SOME x` if p x is true, NONE otherwise.

`map f xopt = SOME(f x)` if xopt is SOME x, NONE otherwise.

`mapPartial f xopt = f x` if xopt is SOME x, NONE otherwise.

Deklaratív programozás. BME VIK, 2006. őszi félév

(Funkcionális programozás)

### Példák opcionális értékek kezelésére

- Egészlista legnagyobb elemének kiválasztása

Üres listának nincs legnagyobb eleme; egyelemű lista egyetlen eleme a „legnagyobb”; legalább kételemű lista legnagyobb eleme az első elem és a maradéklista elemei közül a legnagyobb.

```
(* maxl : int list -> int option
   maxl ns = az ns egészlista legnagyobb eleme *)
fun maxl []      = NONE      (* üres *)
  | maxl [n]    = SOME n    (* egyelemű *)
  | maxl (n::ns) =          (* legalább kételemű *)
    SOME(Int.max(n, valOf(maxl ns)))
```

- Füzér elején álló karaktersorozat átalakítása egész számmá

```
val Int.fromString : string -> int option (* Overflow *)
```

```
Int.fromString s = SOME i if a decimal integer numeral can be scanned
from a prefix of string s, ignoring any initial whitespace;
NONE otherwise. A decimal integer numeral, after any initial
whitespace, must have the form: [+~]?[0-9]+
```

```
Int.fromString "1234"; Int.fromString "-1234"; Int.fromString "~1234";
Int.fromString "+1234"; Int.fromString "+007"; Int.fromString "alma"
```

Deklaratív programozás. BME VIK, 2006. őszi félév

(Funkcionális programozás)

## AZ ORDER TÍPUS

### Az order típus

Az order típus definíciója (ld. `General.sig`)

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

[order] is used as the return type of comparison functions.

Példák az SML-alapkönyvtárból (SML Basis Library)

```
Int.compare      : int * int -> order
Char.compare     : char * char -> order
Real.compare     : real * real -> order
String.compare   : string * string -> order
Time.compare     : time * time -> order
```

## KIVÉTELKEZELÉS

### Kivételkezelés

- Kivételt az `exception` kulcsszóval deklarálunk, a `raise` kulcsszóval jelzünk, a `handle` kulcsszóval bevezetett kifejezésben kezelünk.
- A kivételt általában hibák jelzésére használjuk, de használhatjuk visszalépés kezelésére is (az utóbbira példa a `valtas` függvényben látható a következő fóliák egyikén).
- A kivételdeklaráció az adattípus-deklarációra (`datatype`-deklarációra) emelkedtet: `exception name; exception name of ty.`
- Példák kivétel deklarálására: `exception Valt; exception Hiba of char * int.`
- A kivételkonstruktor állandó vagy függvény lehet.  
Példák: `Valt : exn, Hiba : char * int -> exn.`
- A kivételdeklaráció speciális adattípus-deklaráció, ui. az utóbbival ellentétben dinamikusan bővíti a kivételkonstruktorok halmazát.
- Kivétel jelzésére a `raise` kulcsszóval kezdődő speciális kifejezést kell használunk.
- Példák kivétel jelzésére: `raise Valt, raise Hiba("#N", 4).`
- `raise` (hipotetikus) típusa: `exn -> 'a.` (A `raise` nem függvény!)



## Kivételkezelés (folyt.)

- `raise` alkalmazásának eredménye az ún. *kivételcsomag*. Mivel a kivételcsomag polimorf típusú, bármely más típusal kompatibilis.
- A kivétel kezelése a `case`-szerkezetre emlékeztet: `E handle P1 => E1 | ... | Pn => En`
- Ha `E` „közönséges” értéket ad eredményül, a kivételkezelő egyszerűen továbbadja az eredményt.
- Ha `E` eredménye *kivételcsomag*, az SML megpróbálja illeszteni a `P1, ..., Pn` mintákra.
  - Ha `Pi` ( $1 \leq i \leq n$ ) az első illeszkedő minta, akkor `Ei` a kivételkezelő eredménye.
  - Ha egyetlen minta sem illeszkedik a kivételcsomagra, a kivételkezelő továbbpasszolja.
- Példák kivétel kezelésére:
  - `erme :: valtas (erme::ermelista) (osszeg-erme)`  
`handle Valt => valtas ermelista osszeg`
  - `(fn i => kivKez i handle Hiba(c, i) => (print(str c); i-1)) 0`
- `handle` (hipotetikus) típusa: `exn -> 'a`. (A `handle` nem függvény!)
- Legyen `Ex` `exn` típusú kivétel, `e` pedig tetszőleges kifejezés; ekkor az `e handle Ex => c` (kivételkezelőt tartalmazó) kifejezésben `c`-nek `e`-vel azonos típusúnak kell lennie.

## Kivételkezelés (folyt.)

- Példa visszalépés programozására kivételkezeléssel

```
exception Valt;

(* valtas : int list -> int -> int list
   valtas ermelista osszeg = a lehető legkevesebb érmét tartalmazó olyan
   érmelista, amely elemeinek összege osszeg
   PRE : ermelista = a váltásra használható érmék csökkenő értéksorrendben
   osszeg >= 0
*)
fun valtas _ 0 = []
  | valtas [] _ = raise Valt
  | valtas (erme::ermelista) osszeg =
    if (* ha az adott érme túl nagy, a következővel próbálkozunk *)
      erme > osszeg then valtas ermelista osszeg
    (* ha az adott érmétől kezdve sikerül felváltani, készen vagyunk;
       ha nem, a következő érmével kezdjük újra az adott ponttól *)
    else erme :: valtas (erme::ermelista) (osszeg-erme)
      handle Valt => valtas ermelista osszeg;

valtas [50, 20, 10, 5, 2] 197 = [50, 50, 50, 20, 20, 5, 2];
```

## Kivételkezelés (folyt.)

- A következő programrészletek példák kivétel deklarálására, jelzésére és kezelésére

```
exception Ex of string;

(raise Ex "ex") handle Ex t => t^"eption";
(raise Div) handle Ex t => t^"eption";

exception Hiba of char * int;

fun kivKez 0 = raise Hiba("#N", 4)
  | kivKez ~9 = raise Hiba("#M", 9)
  | kivKez n = n;

fun kivKezel i =
  kivKez i handle Hiba("#N", i) => (print "N"; i)
    | Hiba("#M", i) => (print "M"; i-1);

kivKezel 0 = 4;
kivKezel ~9 = 8;
kivKezel 7 = 7;
```

## Kivételkezelés (folyt.)

- A leggyakoribb belső kivételek

Név	Művelet, amely a kivételt kiválthatja
Bind	Értékdeklarációban a jobb oldali kifejezés nem illeszkedik a bal oldali mintára.
Chr	<code>chr pred succ</code>
Div	<code>/ div mod</code>
Domain	Az érték kilóg az értelmezési tartományból.
Empty	<code>hd tl last</code>
Fail	<code>compile load loadOne Fail : string -&gt; exn</code>
Interrupt	Megszakítás <code>ctrl/c</code> -vel.
Io	Ki/beviteli hiba. <code>Io : {cause : exn, function : string, name : string}</code>
Match	Mintaillesztési hiba <code>case</code> és <code>handle</code> kifejezésben, vagy függvényalkalmazásban.
Option	Hiba egy <code>Option</code> könyvtárbeli függvény alkalmazásakor.
Overflow	<code>~ + - * / div mod abs ceil floor round trunc</code>
Size	<code>^ array concat fromList implode tabulate translate vector</code>
Subscript	<code>copy drop extract nth sub substring take update</code>

- `Fail` és `Io` kivételkonstruktorfüggvények, a többi `exn` típusú kivételkonstruktorállandó.
- `Option` csak `Option`. `Option` néven használható, ha nem nyitjuk meg az `Option` könyvtárat.

## Kivételkezelés (folyt.)

```

• Példák belső kivételek jelzésére
- val (x::xs) = [];
! Uncaught exception:
! Bind
- chr 256;
! Uncaught exception:
! Chr
- Math.sqrt(~1.0);
! Uncaught exception:
! Domain
- fun f 0 = "igaz" | f 1 = "hamis"; f 2;
...
! Uncaught exception:
! Match
- String.sub("alma",4);
! Uncaught exception:
! Subscript

```

Deklaratív programozás. BME VIK, 2006. őszi félév

(Funkcionális programozás)

Egy nagyobb példa a visszalépéses programozásra FP-135

 $n$  vezér a sakktáblán

Hányféleképpen rakható  $n$  vezér egy  $n * n$  méretű sakktáblára úgy, hogy ne üssék egymást?

- A sakktáblát egy olyan  $n$  hosszúságú sorvektorral írjuk le, amelynek egy-egy mezőjébe írt  $s$  szám a sakktábla egy-egy oszlopába lerakott vezér sorának a sorszáma ( $0 <= s < n$ ).

- Példa  $n=4$  esetén:

```

+-----+
| 2 | 0 | 3 | 1 |
+-----+
0 -----> n-1

+-----+
|   | q |   |   |
+-----+
|   |   |   | q |
+-----+
V | q |   |   |   |
+-----+
n-1 |   |   | q |   |
+-----+

```

- A sorvektort listával valósítjuk meg.
- Egy listához *balról könnyű* új elemeket fűzni, ezért a táblát és a vezérek helyzetét leíró listát függőlegesen tükrözzük.

```

+-----+
|   | 3 | 0 | 2 |
+-----+
n-1 <----- 0

+-----+
|   |   | q |   |
+-----+
0

+-----+
|   |   |   | q |
+-----+
V

+-----+
|   | q |   |   |
+-----+
n-1

```

- Egy  $n$  hosszúságú sorvektor  $i$ -edik eleme az  $n - (i + 1)$ -edik elem a listában.

Deklaratív programozás. BME VIK, 2006. őszi félév

(Funkcionális programozás)

## VISSZALÉPÉSES PROGRAMOZÁS

Egy nagyobb példa a visszalépéses programozásra FP-136

 $n$  vezér a sakktáblán (folyt.)

Azt, hogy az új vezért ütik-e más vezérek a táblán, a *sorvektor* vizsgálatával dönthetjük el: az egyes elemek sorszáma által meghatározott oszlopban az értékük által meghatározott sorban vezér van. A lerakás feltételei a következők:

- Az új vezér nem kerülhet egy sorba egyetlen más vezérrel sem, azaz az új listaem *értéke* nem fordulhat elő a lista már felépített részében.
- Az új vezér átlós irányban sem lehet egy vonalban más vezérrel. Ez azt jelenti, hogy ha a lista elejére (a 0. elemébe!) az  $s$  sorindexet akarjuk írni, akkor az  $i$ -edik listaelem értéke, feltéve, hogy van ilyen elem, nem lehet  $s - i$ , sem  $s + i$ .  
A következő példa megvilágítja az esetet.

Ha a tábla 1. sorába akarjuk lerakni az új vezért, akkor az  $x$ -szel megjelölt mezőket kell megvizsgálnunk. Az új mezővel együtt a listának már három eleme van. Az 1-es indexű elem nem lehet  $s + 1$ , sem  $s - 1$ , a 2-es indexű elem pedig nem lehet  $s + 2$ , sem  $s - 2$ .

```

+-----+
|   |   |   |   |
+-----+
|   | 1 |   |   |
+-----+
n-1 <----- 1 0

+-----+
|   |   | x |   |
+-----+
0

+-----+
|   | q |   |   |
+-----+
1

+-----+
|   |   | x |   |
+-----+
V

+-----+
|   |   |   | x |
+-----+
n-1

```

A lista rekurzív algoritmussal dolgozható fel.

Deklaratív programozás. BME VIK, 2006. őszi félév

(Funkcionális programozás)

## *n* vezér a sakktáblán: „ütésben van”-vizsgálat

```
(* utesbenVan : int list -> bool
   utesbenVan zs = igaz, ha a (hd zs) vezért legalább egy
                   (tl zs)-beli vezér üti
*)
fun utesbenVan [] = false
  | utesbenVan (z::zs) =
    let
      (* uV : int -> int -> int list -> bool
         uV s1 s2 rs = igaz, ha a z vezért legalább egy
                    rs-beli vezér üti
      *)
      fun uV _ _ [] = false
        | uV s1 s2 (r::rs) = z = r orelse
                              s1 = r orelse
                              s2 = r orelse
                              uV (s1-1) (s2+1) rs
    in
      uV (z-1) (z+1) zs
    end
```

Deklaratív programozás. BME VIK, 2006. őszi félév

(Funkcionális programozás)

Egy nagyobb példa a visszalépéses programozásra FP-139

## *n* vezér a sakktáblán: több megoldás előállításával visszalépéssel

```
(* vezerek1 : int -> int list list
   vezerek1 n = a feladvány összes megoldásának listája n vezér esetén
*)
fun vezerek1 n =
  let (* vez: int -> int list -> int list list
       vez z zs: az összes megoldás listája n vezér esetén
     *)
      fun vez z zs =
        if (* vissza kell lépni, ha z=0 és ütésben van,
           vagy ha már minden sort megpróbált *)
           z = 0 andalso utesbenVan zs orelse z = n
        then raise Zsakutca
        else if length zs = n
        then [rev zs] (* megvan egy megoldás, listában adja vissza *)
        else (* folytatja a következő sorral, majd hozzáfűzi ... *)
              (vez (z+1) zs handle Zsakutca => []) @
              (* ... a 0. sortól kezdve a következő vezér lerakásával
                 kapott megoldásokat *)
              (vez 0 (z::zs) handle Zsakutca => [])
    in
      (* a 0. sorral kezd: ilyenkor nem lehet ütésben *)
      vez 0 []
    end
```

Deklaratív programozás. BME VIK, 2006. őszi félév

(Funkcionális programozás)

## *n* vezér a sakktáblán: egy megoldás előállítása

```
exception Zsakutca
(* vezerek0 : int -> int list
   vezerek0 n = a feladvány egy megoldása n vezér esetén
*)
fun vezerek0 n =
  let (* vez : int -> int list -> int list
       vez z zs = egy megoldás n vezér esetén *)
      fun vez z zs =
        if (* vissza kell lépni, ha z=0 és ütésben van, ... *)
           z = 0 andalso utesbenVan zs orelse
           (* ... vagy ha már minden sort megpróbált *)
           z = n
        then raise Zsakutca
        else if length zs = n
        then rev zs (* megvan egy megoldás *)
        else (* folytatja a 0. sortól a következő vezér lerakásával,
              és ha elakad, visszalép a következő sorra *)
              vez 0 (z::zs) handle Zsakutca => vez (z+1) zs
    in
      (* a 0. sorral kezd: ilyenkor nem lehet ütésben *)
      vez 0 []
    end
```

Deklaratív programozás. BME VIK, 2006. őszi félév

(Funkcionális programozás)

Egy nagyobb példa a visszalépéses programozásra FP-140

## *n* vezér a sakktáblán: több megoldás előállítása listák listájával

Az előző megoldás sémája sokszor használható, de ebben az egyszerű esetben felesleges: a kivétel jelzése helyett üres listát adhatunk eredményül, hiszen a kivételkezelők is csak ezt teszik.

```
(* vezerek2 : int -> int list list
   vezerek2 n = a feladvány összes megoldásának listája n vezér esetén
*)
fun vezerek2 n =
  let (* vez: int -> int list -> int list list
       vez z zs: az összes megoldás listája n vezér esetén
     *)
      fun vez z zs =
        if z = 0 andalso utesbenVan zs orelse z = n
        then []
        else if length zs = n
        then [rev zs]
        else vez (z+1) zs @ vez 0 (z::zs)
    in
      vez 0 []
    end
```

Deklaratív programozás. BME VIK, 2006. őszi félév

(Funkcionális programozás)

## *n* vezér a sakktáblán: több megoldás előállítása listák listájával (folyt.)

Akkumulátor alkalmazásával:

```
(* vezerek3 : int -> int list list
   vezerek3 n = a feladvány összes megoldásának listája
               n vezér esetén
*)
fun vezerek3 n =
  let (* vez: int -> int list -> int list list -> int list list
       vez z zs: az összes megoldás listája n vezér esetén
     *)
      fun vez z zs zss =
          if z = 0 andalso utesbenVan zs orelse z = n
          then zss
          else if length zs = n
              then rev zs :: zss
              else vez 0 (z::zs) (vez (z+1) zs zss)
    in
      vez 0 [] []
    end
```

## HALMAZMŰVELETEK

### Halmazműveletek: „benne van-e?” (isMem) és „ha új, tedd bele” (newMem)

- isMem igaz értéket ad eredményül, ha a keresett elem benne van a listában.

```
(* isMem : 'a * 'a list -> bool
   isMem(x, ys) = x eleme-e ys-nek
*)
fun op isMem (_, []) = false
  | op isMem (x, y::ys) = x = y orelse op isMem(x, ys)
infix isMem
```

Megjegyzés: az op operátor nélkül az infix deklarációt követően a függvénydefiníciót nem lehetne újrafordítani.

- newMem egy új elemet rak be egy listába, ha még nincs benne.

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

newMem, ha a sorrendtől eltekintünk, halmazt hoz létre.

### Halmazműveletek: „listából halmaz” (setof)

- setof halmazt készít egy listából úgy, hogy kiszedi belőle az ismétlődő elemeket. Rossz hatékonyságú.

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

- Öt halmazműveletet definiálunk:

- unió (union,  $S \cup T$ ),
- metszet (inter,  $S \cap T$ ),
- részhalmaza-e (isSubset,  $T \subseteq S$ ),
- egyenlők-e (isSetEq,  $S = T$ ),
- hatványhalmaz (powerSet,  $pS$ ).

- Rendezetlen listával ábrázoljuk most a halmazokat.

- Gyakorlófeladatnak meghagyjuk a halmazműveletek megvalósítását rendezett listákkal és rendezett fákkal. (A vizsgán is kaphatnak ilyen feladatokat.)

## Halmazműveletek: „unió” (union) és „metszet” (inter)

- Két halmaz uniója

```
(* union : 'a list * 'a list -> 'a list
   union(xs, ys) = az xs és ys elemeiből álló halmazok uniója
*)
fun union ([], ys)      = ys
  | union (x::xs, ys) = newMem(x, union(xs, ys))
```

- Két halmaz metszete

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

## LISTÁK RENDEZÉSE

## Halmazműveletek: „részhalmaza-e” (isSubset) és „egyenlők-e” (isSetEq)

- Részhalmaza-e egy halmaz egy másiknak?

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

infix isSubset
```

- Két halmaz egyenlősége (a listák egyenlőségvizsgálata beépített művelet az SML-ben, halmazokra mégsem használható, mert pl. [3, 4] és [4, 3] listaként ugyan különböznek, de halmazként egyenlők)

```
(* isSetEq : 'a list * 'a list -> bool
   isSetEq(xs, ys) = az xs elemeiből álló halmaz egyenlő-e
                       az ys elemeiből álló halmazzal
*)
fun isSetEq (xs, ys) = (xs isSubset ys) andalso (ys isSubset xs)
```

## Listák rendezése

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

## Beszűrő rendezés

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

```
(* ins : real * real list -> real list
   ins (x, ys) = ys kibővítve x-szel a <= reláció szerint
   PRE: ys a <= reláció szerint rendezve van *)
fun ins (x, y::ys) = if x <= y then x::y::ys else y::ins(x, ys)
| ins (x : real, []) = [x]
```

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

```
(* inssort : ('a * 'b list -> 'b list) -> 'a list -> 'b list
   inssort f xs = az xs elemeiből álló, az f felhasználásával
   rendezett lista *)
fun inssort f (x::xs) = f(x, inssort f xs)
| inssort _ [] = [];
```

- Példa `inssort` alkalmazására:

```
inssort ins [4.24, 4.1, 5.67, 1.12, 4.1, 0.33, 8.0] =
[0.33, 1.12, 4.1, 4.1, 4.24, 5.67, 8.0];
```

## Beszűrő rendezés, generikus változat (folyt.)

- `inssort` eddigi változatai előbb elemeire szedik szét a rendezendő listát, majd hátulról visszafelé haladva, rendezés közben építik fel az újat.
- A jobbrekurziót és akkumulátort használó változatnak (`inssort2`) kisebb veremre van szüksége, mivel a listáról leválasztott elemeket balról jobbra haladva azonnal berakja a helyükre az eredménylistában. (A két megoldás futási idejét később összehasonlíthatjuk).

```
(* inssort2 : ('a * 'a -> bool) -> 'a list -> 'a list
   inssort2 cmp xs = az xs elemeiből álló, a cmp reláció
   szerint rendezett lista *)
fun inssort2 cmp xs =
  let (* sort : 'a list -> 'a list -> 'a list
       sort xs zs = zs kibővítve az xs-nek a cmp reláció
       szerint rendezett elemeivel
       PRE: zs cmp szerint rendezve van *)
      fun sort (x::xs) zs = sort xs (ins cmp (x, zs))
        | sort [] zs = zs
      in
        sort xs []
      end
```

## Beszűrő rendezés, generikus változat

- Az `ins` függvényt generikussá tesszük:

```
(* ins : ('a * 'a -> bool) -> 'a * 'a list -> 'a list
   ins cmp (x, ys) = ys kibővítve x-szel a cmp reláció szerint
   PRE: ys a cmp reláció szerint rendezve van *)
fun ins cmp (x, ys) =
  let fun ins0 (y::ys) =
        if cmp(x, y) then x::y::ys else y::ins0 ys
      | ins0 [] = [x]
      in
        ins0 ys
      end
```

- Ezzel `inssort` egy újabb változata:

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

## Beszűrő rendezés foldr-rel és foldl-lel

- A második argumentumát akkumulátorként használó `foldl` kisebb vermet használ `foldr`-nél, ezért `inssortL` hosszabb listákat tud rendezni:

```
fun inssortR cmp = foldr (ins cmp) [];
fun inssortL cmp = foldl (ins cmp) [];
```

- Példák `inssort`-tal és `inssort2`-vel:

```
inssort op<= [4.24, 4.1, 5.67, 1.12, 4.1, 0.33, 8.0] =
[0.33, 1.12, 4.1, 4.1, 4.24, 5.67, 8.0];
inssort2 op>= [4, 4, 5, 1, 0, 8] = [8, 5, 4, 4, 1, 0];
inssort op< (explode "qwer") = [#"e", #"q", #"r", #"w"];
```

- Példák `foldr` és `foldl` felhasználásával:

```
fun inssortRi cmp = foldr (ins cmp) [];
fun inssortLr cmp = foldl (ins cmp) ([] : real list);

inssortRi op>= [4, 4, 5, 1, 0, 8] = [8, 5, 4, 4, 1, 0];
inssortLr op<= [4.24, 4.1, 5.67, 1.12, 4.1, 0.33, 8.0] =
[0.33, 1.12, 4.1, 4.1, 4.24, 5.67, 8.0];
```

## A futási idők mérése, összehasonlítása

- 5000 elemet tartalmazó, véletlenszerű és növekvő sorrendű listák rendezéséhez szükséges futási időt mérünk.

- Véletlen eloszlású egészlistát állít elő a `Random.rangelist` függvény:

```
val xs5000R =
  Random.rangelist (1, 100000) (5000, Random.newgen());
```

- Növekvő sorrendű egészlistát állít elő a `---` operátor:

```
infix ---;
fun fm --- to =
  let fun upto to zs =
        if to < fm then zs else upto (to-1) (to::zs)
      in
        upto to []
      end;
  val xs5000N = 1 --- 5000;
```

## A futási idők mérése, összehasonlítása (folyt.)

- Egy Intel Pentium M 1.4 GHz CPU-n, linux operációs rendszer alatt a következő időket mértük:

```
Int sort with inssort, recursive, op<=, length = 5000 (incr), time = 0.00s
Int sort with inssort, iterative, op<=, length = 5000 (incr), time = 4.24s
Int sort with inssort, recursive, op>=, length = 5000 (incr), time = 4.65s
Int sort with inssort, iterative, op>=, length = 5000 (incr), time = 0.00s
Int sort with inssort, recursive, op<=, length = 5000 (rand), time = 1.96s
Int sort with inssort, iterative, op<=, length = 5000 (rand), time = 1.78s
Int sort with inssort, recursive, op>=, length = 5000 (rand), time = 1.95s
Int sort with inssort, iterative, op>=, length = 5000 (rand), time = 1.77s
```

- Jól látszik, hogy az akkumulátort nem használó (rekurzív) és az akkumulátort használó (iteratív) változatok futási ideje csak kismértékben különbözik egymástól.

- A rekurzív és az iteratív változat között lényeges különbség a veremhasználatban van.

- A rekurzív változat már egy 500 ezer elemű listát sem tud kezelni:

```
inssort op< (1---500000);
! Uncaught exception:
! Out_of_memory
```

## A futási idők mérése, összehasonlítása (folyt.)

- A futási időt az alábbi függvénnyel mérhetjük:

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

fun futIdo (sort, sortFn) (cmp, cmpFn) (xs, kind) =
  let val starttime = Timer.startCPUTimer()
      val zs = sort cmp 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;

futIdo (inssort, "inssort, recursive") (op<=, "op<=") (xs5000N, "incr");
futIdo (inssort2, "inssort, iterative") (op<=, "op<=") (xs5000N, "incr");
futIdo (inssort, "inssort, recursive") (op>=, "op>=") (xs5000N, "incr");
futIdo (inssort2, "inssort, iterative") (op>=, "op>=") (xs5000N, "incr");
futIdo (inssort, "inssort, recursive") (op<=, "op<=") (xs5000R, "rand");
futIdo (inssort2, "inssort, iterative") (op<=, "op<=") (xs5000R, "rand");
futIdo (inssort, "inssort, recursive") (op>=, "op>=") (xs5000R, "rand");
futIdo (inssort2, "inssort, iterative") (op>=, "op>=") (xs5000R, "rand");
```

## A futási idők mérése, összehasonlítása (folyt.)

- Az iteratív változatnak egy 5 millió elemű listával sincs még gondja:

```
inssort2 op< (rev(1---5000000));
> val it = [1, 2, ...] : int list
```

- A futási idő rendezett lista esetén attól függ, hogy mi a listaelemek sorrendje a lista bejárás irányához képest:

- ha a lista kezdetben megfelelő sorrendű és jobbról balra járjuk be (első rekurzív eset), akkor hátulról visszafelé haladva a már addig létrehozott eredménylista elejére kell befűzni a következő elemet, ami gyorsan megy;
- ha a lista kezdetben megfelelő sorrendű és balról jobbra járjuk be (első iteratív eset), akkor előlről hátrafelé haladva a következő elemet a már addig létrehozott lista végére kell befűzni, ami lassan megy;
- ha a lista kezdetben fordított sorrendű és jobbról balra járjuk be (második rekurzív eset), akkor hátulról visszafelé haladva a már addig létrehozott eredménylista végére kell befűzni a következő elemet, ami lassan megy;
- ha a lista kezdetben fordított sorrendű és balról jobbra járjuk be (második iteratív eset), akkor előlről hátrafelé haladva a következő elemet a már addig létrehozott lista elé kell befűzni, ami gyorsan megy.