1. Polimorf típusellenőrzés

A típus nélküli (Lisp, Prolog stb.), ill. a gyengén típusos nyelvek (C, Java stb.) a programozónak nagyobb szabadságot adnak, de a tévedés lehetősége is nagyobb. Az erősen típusos nyelvek (pl. Pascal, Ada, SML) a programozót jobban korlátozzák, de biztonságosabbak.

Az SML-ben *polimorf típusellenőrzés* van: a szigorú típusellenőrzés flexibilis, automatikus típuslevezetéssel (type derivation, type inference) társul.

Nézzük a következő definíciót!

```
fun id x = x
```

Milyen típusú itt az `x`? Mindegy! Az `id` függvény ún. *polimorf függvény,* az `x` pedig *politípusú* azonosító. A típussémák elméletében a politípus jele α, β, γ stb., az SML-ben `´a`, `´b`, `´c` stb. Az SML-értelmező válasza a fenti definícióra tehát a következő:

```
> val id = fn :  ´a -> ´a
```

A *percjellel* kezdődő típusneveket (`´a`-t, `´b`-t, `´c`-t stb.) *típusváltozónak* nevezzük, és *alfának, bétának, gammának* stb. olvassuk.

A polimorfizmus több válfaját alkalmazzuk a programozásban.

- Egy *polimorf név* **egyetlen** olyan algoritmust azonosít, amely tetszőleges típusú argumentumra alkalmazható; ez a *paraméteres polimorfizmus.*
- Egy *többszörösen terhelt név* **több** algoritmust azonosít: ahány típusú argumentumra alkalmazható, annyifélét; ez az ad-hoc vagy *többszörös terheléses polimorfizmus*.
- A polimorfizmus harmadik változatát *öröklődéses polimorfizmusnak* nevezzük.[1]

A polimorf típus: *típusséma.* Amikor a típusváltozót konkrét típussal helyettesítjük, e séma egy-egy példányát kapjuk.

Az egyenlőségvizsgálatot is megengedő ún. *egyenlőségi típusok* (equality types) típusváltozóinak jelölésére *két percjellel* kezdődő neveket használunk: `´´a`, `´´b`, `´´c` stb.

Végül nézzünk két újabb, nagyon egyszerű példát polimorf függvények definiálására! Egy pár első, ill. második tagjának kiválasztására használhatók az alábbi *projekciós* függvények, ahol `´a` és `´b` nem feltétlenül különböző típusok:

```
(* fst :  ´a * ´b -> ´a
*)
fun fst(x, _) = x
(* snd :  ´a * ´b -> ´b
*)
fun snd(_, y) = y
```

2. Rekurzív függvények

Rekurzív megoldás esetén a megoldandó feladatot olyan részfeladatokra bontjuk, amelyek közül egyet vagy többet (de nem mindet!) az eredeti feladathoz hasonló módon oldunk meg.

[1] Öröklődéses polimorfizmust az objektum-orientált programozásban alkalmaznak.

2.1. Faktoriális „naív” rekurzióval

A *faktoriális* matematikai definíciója:
fac $0 = 1$
fac $n = n*$fac$(n-1)$
A faktoriális-függvény megvalósítása SML-ben:

```
(* fac n = n!
   fac :  int -> int
*)
fun fac n = if n = 0 then 1 else n * fac(n-1)
```

Mohó kiértékelésének menete $n = 4$ mellett:

```
fac(4)→4*fac(4-1)→4*fac(3)→4*(3*fac(3-1))
   →4*(3*fac(2))→···→4*(3*(2*1))→···→24
```

A rekurzív kiértékelés szigorúan követi a matematikai definíciót.

2.1.1. Faktoriális iterációval (jobbrekurzióval)

A fenti kiértékelésben az a rossz, hogy a rekurzív végrehajtás során minden részeredményt tárolni kell. Ha a szorzás asszociativitását kihasználnánk, nem kellene tárolni az összes tényezőt, csak az aktuális részeredményt. A számítógép a szorzás e tulajdonságát (és bármely más tulajdonságot) persze csak akkor alkalmaz, ha utasítjuk rá. Írjunk ilyen függvényt!

Először a

```
(* faci(n, p)= p*n!
   faci :  int * int -> int
*)
fun faci (n, p) = if n = 0 then p else faci(n-1, n*p)
```

segédfüggvényt definiáljuk, majd felhasználjuk az eredetivel azonos hívási felületű függvény megvalósítására:

```
(* fac n = n!
   fac :  int -> int
*)
fun fac n = faci(n, 1)
```

Nézzük a kiértékelését (egyes triviális lépéseket összevonunk):

```
faci(4,1)→faci(4-1,4*1)→faci(3,4)→faci(3-1,3*4)
   →faci(2,12)→···→faci(0,24)→24
```

Kiértékelés közben a `p` segédváltozóban (az ún. *gyűjtőargumentumban, akkumulátorban)* gyűjtjük a részeredményt, ezért a tárigény állandó marad. A kiértékelés tehát *iteratív* jellegű. Az ilyen rekurziót *terminális rekurziónak* vagy *jobbrekurziónak* (angolul: *tail* vagy *terminal recursion)* nevezzük. A jó fordítóprogramok felismerik az iteratívvá alakítható rekurziót, és még hatékonyabb tárgykódot állítanak elő. A `faci(n-1, n*p)` rekurzív hívás eredménye – további számítások nélkül – közvetlenül `faci(n, p)` eredményét adja. Az ilyen, ún. *terminális hívást* végre lehet hajtani úgy, hogy az értelmező- vagy fordítóprogram az `n` és a `p` lokális változókba beírja az `n` és a `p` új értékét, majd visszaugrik a kód elejére ahelyett, hogy ténylegesen, újból meghívná a függvényt. Az előző változatban a `fac(n-1)` hívás *nem* terminális hívás, mert az eredményét még meg kell szorozni `n`-nel.

`faci`-t rekurzív függvényként definiáltuk, ezért viszonylag könnyű belátni a helyességét, ugyanakkor a kiértékelése a segédváltozó bevezetésével iteratívvá vált. Nagyon sok rekurzív függvény (de nem mind!) iteratívvá alakítható segédváltozó bevezetésével, és így tárterületet takaríthatunk meg. Sokszor a végrehajtási idő is csökken, de sajnos néha nőhet is. Ha nem nyilvánvaló a nyereség, a lehető legtermészetesebb módon kell felírni az algoritmust. Esetünkben `faci` valamivel hatékonyabb `fac`-nál, ugyanakkor a működése nehezebben érthető meg.

2.2. Fibonacci-számok

A Fibonacci-számok jól ismert definíciója:
$F_0 = 0,$
$F_1 = 1,$
$F_n = F_{n-2} + F_{n-1},\ n > 1.$
Ha ezt így, ahogy van, átírjuk SML-re, használhatatlan programot kapunk: pl. F_{35}-öt egy 133 MHz-es Pentium processzoros számítógépen, linux alatt csaknem 35 s alatt számolja ki az MOSML és csaknem 12 s alatt az smlnj! Keressünk jobb megoldást!

Először is `nextfib` néven olyan függvényt írunk, amely egy Fibonacci-számpárból előállítja a következő Fibonacci-számpárt:

```
(* netxfib(p,c) = a (p,c) Fibonacci-számpárt követő
                  Fibonacci-számpár
   nextfib :  int * int -> int * int
*)
fun nextfib (prev, curr) = (curr, prev + curr)
```

A megoldás jó, hiszen az SML-ben a függvény eredménye is tetszőleges típusú érték lehet! `nextfib` felhasználásával:

```
(* fibpair n = az n-edik Fibonacci-számpár (n>0)
   fibpair :  int -> int * int
*)
fun fibpair n =
      if n = 1 then (0, 1) else nextfib(fibpair(n-1))
```

A kiértékelése elég nehezen követhető, ugyanis `fibpair` a

```
nextfib(nextfib (··· (nextfib(0, 1) ···)))
```

hívássorozatot állítja elő. Ugyanakkor a végrehajtása nagyon gyors, például a `fibpair 44` hívás azonnal kiírja a `(433494437, 701408733)` számpárt, amelynek a második tagja az `Int.Maxint`-nál még éppen nem nagyobb Fibonacci-szám. A keresett Fibonnaci-számot a számpár második tagjának kiválasztásával kapjuk, például:

```
- #2(fibpair 44);
> val it = 701408733 :  int
```

Szép, áttekinthető és gyors megoldást kapunk *iterációval*:

```
(* iterfib(n,p,c) = a (p,c) Fibonacci-számpárt követő
                    n-edik Fibonacci-szám (n>0)
   iterfib :  int * int * int -> int
*)
fun iterfib (1, prev, curr) = curr
  | iterfib (n, prev, curr) = iterfib(n-1, curr, prev + curr)
```

Az `else` ágban a rekurzió terminális. `fib` `iterfib`-et hívja meg az `n`-edik Fibonacci-szám előállításához:

```
(* fib n = az n-edik Fibonacci-szám
   fib :  int -> int
*)
fun fib 0 = 0
```



```
  | fib n = iterfib(n, 0, 1)
```

Nézzünk egy példát `fib` redukciójára:

```
fib 7→iterfib(7,0,1)→iterfib(6,1,1)→iterfib(5,1,2)
   →···→iterfib(1,8,13)→13
```

3. Egyidejű deklaráció

Az *egyidejű* (más néven *szimultán)* deklaráció elsősorban kölcsönösen rekurzív függvények definiálására használható. Kölcsönösen rekurzív függvényeket szekvenciális deklarációval nem lehet deklarálni.
`val` $id_1 = E_1$ `and` ... `and` $id_n = E_n$
Az egyidejű deklaráció előbb kiszámítja az *összes* E_i-t (kiszámításuk sorrendje, mivel nem lehet mellékhatásuk, közömbös), majd a kiszámított értékeket balról jobbra haladva, rendre hozzárendeli a megfelelő id_i-hez. Példaként bemutatunk egy nem igazán hatékony megoldást az egész számok páros, ill. páratlan voltát tesztelő `even`, ill. `odd` függvény megvalósítására:

```
(* even n = igaz, ha n páros
   even :  int -> bool
   odd n = igaz, ha n páratlan
   odd :  int -> bool
*)
fun even 0 = true
  | even n = odd(n-1)
and odd 0 = false
  | odd n = even(n-1)
```

Az egyidejű deklaráció azonosítók értékének felcserélésére is használható, például:

```
- val alma = "PIROS";
- val korte = "BARNA";
- val alma = korte and korte = alma;
> val alma = "BARNA" :  string
> val korte = "PIROS" :  string
```

4. Listák

A *lista* azonos típusú elemek végtelen (gyakorlatilag véges) sorozata. Példák:

```
[3, 5, 9, 13, 17, 21]
["alma", "meggy", "szilva"]
```

A listát *rekurzív adatszerkezetnek* is tekinthetjük. A rekurzív definíció szerint a lista

- vagy üres,
- vagy egy elemből és ezt az elemet követő listából áll.

Az üres listát – a listaműveletek *egységelemét* – `[]`-lel vagy `nil`-lel jelöljük. A legalább egy elemből álló lista első elemét a lista *fejének,* a többi elemből álló listát a lista *farkának* nevezzük.

A listában az elemek sorrendje fontos. Egyes elemek ismétlődhetnek. Az elemek típusa tetszőleges, de egy listának csak azonos típusú elemei lehetnek.[2]

```
> val it = [3, 5, 9, 13, 17, 21] :  int list
> val it = ["alma","meggy","szilva"] :  string list
```

[2] Más funkcionális nyelvekben, pl. a LISP-ben a listának különböző típusú elemei is lehetnek.

4.1. Lista létrehozása

Két konstruktorművelet használható lista létrehozására: a `[]` (`nil`) *konstruktorállandó* és az *infix* pozíciójú `::` *konstruktoroperátor.*[3]

Egy lista vagy üres (`[]`) lehet, vagy `x::xs` alakú, ahol `x`-szel a lista *fejét*, `xs`-sel pedig a lista *farkát,* azaz az eredetinél eggyel rövidebb részlistáját jelöljük. Könnyen elérhető egy lista *első,* sok munkával az *utolsó* eleme.

A `[3, 5, 9]` jelölés[4] rövidítés, mégpedig a `3::(5::(9::nil)` jelölés rövidítése. Azért, hogy ne kelljen zárójelet használni, az *infix* `::\` *(*négyespont*)* operátor *jobbra köt:* `3::5::9::nil`. Egy lista elemeiként tetszőleges kifejezéseket adhatók meg.

Első példánk a lista alkalmazására legyen egy olyan rekurzív függvény, amely az `m` és `n` közötti egészek listáját adja eredményül. Ha `m>n`, az eredmény legyen az üres lista.

```
(* upto(m ,n) = az [m,n] tartományba eső egészek listája
   upto :  int * int -> int list
*)
fun upto (m, n) =
      if m > n then [] else m ::  upto(m+1, n)
```

4.2. Lista elemeinek szorzata

A rekurzív megoldást általában az előforduló esetek elemzésével, a jellemző esetek szétválasztásával találjuk meg: listák esetén általában az *üres* és a *nem üres* lista esetét kell megkülönböztetnünk. Az üres lista nem létező elemeinek *szorzatát* célszerű 1-nek választani (miért is?): az 1 a szorzás egységeleme.

A `[]` *minta* csak az üres listára illeszkedik. Az `n::ns` minta csak olyan listára illeszkedik, amelynek legalább egy eleme van; a mintát zárójelbe kell rakni, mert *a függvényalkalmazás precedenciája nagyobb a négyespontnál.*

```
(* prod xs = az xs egészlista elemeinek szorzata
   prod :  int list -> int
*)
fun prod [] = 1
  | prod (n::ns) = n * prod ns
```

Az üres, ill. a nem üres listát kezelő *ágak* (a Prolog szóhasználatával: *klózok)* a jelen esetben *kölcsönösen kizárják* egymást (a minták diszjunktak), ezért a két ág sorrendje *az eredmény szempontjából* közömbös. De nem közömbös a sorrendjük *hatékonysági szempontból:* mivel a vizsgált lista mindaddig nem üres, amíg a rekurzív feldolgozás során el nem fogynak az elemei, az üreslista-vizsgálat az utolsó eset kivételével meghiúsul. Ezért a hatékonyság érdekében a két ágat célszerű fordított sorrendben felírni:

```
fun prod (n::ns) = n * prod ns
  | prod [] = 1
```

Ha a függvény meghívásakor már az első minta illeszkedik az argumentumra, a második ág kiértékelésére nem kerül sor. (Amint tudjuk, a kifejezések kiértékelése balról jobbra és felülről lefelé halad.) Hasonló esetekben mindig törekedni fogunk arra, hogy hatékony megoldást alkalmazzunk.

A listaelemek *összege* hasonlóan képezhető. Az összeadás egységeleme a 0.

[3] `[]` és `nil` jelentése azonos. A `::` konstruktoroperátor helyett sokszor a vele azonos hatású, de prefix pozíciójú `cons` *konstruktorfüggvényről* beszélünk, amely azonban nincs belső függvényként definiálva az SML-ben.

[4] Az SML-lista szintaxisa *csak hasonló* a Prolog-listáéhoz! A Prologban ui. `[5|[6]]` és `[5,6]` azonos listát jelölnek. Az SML-ben az `[5::[6]]`-tal jelölt lista `[[5,6]]`-tal azonos.

4.3. Lista legnagyobb eleme

Kicsit más a feladat egy lista *legnagyobb* (legkisebb) elemének megkeresésekor:

- üres listának nincs legnagyobb eleme,
- egyelemű listában az egyetlen elem a legnagyobb,
- legalább kételemű lista esetén a legnagyobb elemet úgy kapjuk meg, hogy vesszük a két elem közül a nagyobbat, továbbá a maradéklista elemei közül a legnagyobbat, és e kettő közül kiválasztjuk a nagyobbat.

```
(* maxl ns = az ns egészlista legnagyobb eleme
   maxl :  int list -> int
*)
fun maxl (m::n::ns) =
      if m > n then maxl(m::ns) else maxl(n::ns)
  | maxl [m] = m
```

Az SML-értelmező erre a függvénydefinícióra figyelmeztető üzenettel válaszol:

```
      !  Warning:  pattern matching is not exhaustive
```

Megjegyzések:

1. `maxl` üres listára nem alkalmazható; erre figyelmeztet a fenti üzenet. Később megmutatjuk, hogyan kell kezelni az ilyen, ún. *kivételeket.*
2. az `[m]` minta csak *egyetlen elemből* álló listára illeszkedik.
3. az `(m::n::ns)` minta csak olyan listára illeszkedik, amelynek legalább két eleme van.
4. Az algoritmus szempontjából mindegy lenne, hogy a lista elemei milyen típusúak, de a > reláció, mint tudjuk, többszörösen terhelhető módon polimorf (l. 1 szakasz). Mivel a programozó nem hozhat létre többszörösen terhelhető neveket az SML-ben, a függvény definiálásakor el kell dönteni, hogy a > relációnak melyik változatát kell beépíteni `maxl`-be (alapértelmezés szerint `int` a többszörösen terhelhető műveletek argumentumainak típusa). Később megmutatjuk, hogyan kell ún. *generikus* algoritmusokat írni.

4.4. Karakter, füzér és lista

Az SML-ben a füzér egydimenziós karaktertömb (karaktersorozat), nem lista. A füzért a rekurzív feldolgozás során esetleg többször át kell alakítani listává, majd vissza füzérré. Két belső függvény van erre a célra az SML-ben:

```
- explode "mosml";
> val it = [#"m", #"o", #"s", #"m", #"l"] :  char list
```

A kapott lista minden eleme egyetlen karakter.

```
- implode it;
> val it = "mosml" :  string
```

Füzérekből álló lista elemeit egyesíteni a `concat`-tal lehet:

```
- concat ["mo","sml"];
> val it = "mosml" :  string
```

A következő szakaszokban néhány fontos listakezelő függvényt definiálunk.

4.5. Listák vizsgálata és darabokra szedése

Három függvényt mutatunk be ebben a csoportban: a `null` egy lista üres voltát vizsgálja, a `hd` egy nem üres lista első elemét, a `tl` egy nem üres lista első elemét követő részlistáját (a lista farkát) adja eredményül.[5]

```
(* null xs = igaz, ha az xs lista üres
   null :  ´a list -> bool
*)
fun null (_::_) = false
  | null [] = true
```

Az aláhúzás (_) a már ismert *mindenesjel*. Az eredmény szempontjából a minták felírásának sorrendje közömbös ebben a függvényben is.

```
(* hd xs = a nem üres xs lista feje
   hd :  ´a list -> ´a
*)
fun hd (x::_) = x
```

Ez a `hd` csak nemüres listára alkalmazható, amire az SML-értelmező figyelmeztet. Később bemutatjuk az üres listát is kezelni képes változatát.

```
(* tl xs = a nem üres xs lista farka
   tl :  ´a list -> ´a list
*)
fun tl (_::xs) = xs
```

Ez a `tl` is csak nemüres listára alkalmazható, amire az SML-értelmező figyelmeztet. Később bemutatjuk az üres listát is kezelni képes változatát.

`hd` és `tl` *szelektorfüggvény,* `null` pedig *tesztelőfüggvény.*

4.6. Listák és egész számok

Ebben a csoportban is három függvényt mutatunk be: `length` egy lista hosszát adja eredményül; `take` egy lista elejéről vett adott számú elemből, `drop` egy lista elejéről adott számú elem elhagyásával képez eredménylistát.[6] Ha

$xs = [x_0, x_1, \ldots, x_{i-1}, x_i, x_{i+1}, \ldots, x_{n-1}]$

akkor

`length xs=` n
`take(xs,i)=` $[x_0, x_1, \ldots, x_{i-1}]$
`drop(xs,i)=` $[x_i, x_{i+1}, \ldots, x_{n-1}]$

`length` naív változata a következő:

```
(* nlength xs = xs elemeinek száma
   nlength :  ´a list -> int
*)
fun nlength (_::xs) = 1 + nlength xs
  | nlength [] = 0
```

Egy példa a függvény alkalmazására:

```
- nlength [[1,2,3],[4,5,6]];
```

[5] `hd` a *head* (fej), `tl` a *tail* (farok) szóból származik. `null`, `hd` és `tl` fenti definíciója csak illusztráció, ugyanis mind a három *belső* függvényként van definiálva az SML-ben.

[6] `length` *belső* függvény, `take` és `drop` pedig a List könyvtárban van definiálva az SML-ben.


```
> val it = 2 :  int
```

`nlength` rossz hatékonyságú, mert nem iteratív: az 1-esek a veremben csak gyűlnek, gyűlnek, amíg a maradéklista ki nem ürül. A függvény javított, iteratív változata:

```
(* length xs = xs elemeinek száma
   length :  ´a list -> int
*)
local
   fun addlen (n, _::xs) = addlen (n+1, xs)
     | addlen (n, []) = n
in
   fun length xs = addlen(0, xs)
end
```

A *nagyon gyakran használt* (pl. könyvtárazott) függvények *hatékonysága* és *robosztussága*[7] fontos, még ha kevésbé szépek, kevésbé olvashatók is. A speciális feladatokra írt, *ritkábban használt* függvények azonban legyenek *könnyen olvashatók,* és a *helyességüket* is *egyszerűen* lehessen belátni, bizonyítani.

`take` első változata:

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

A bemutatott változat az `if-then-else` alkalmazása miatt nem olyan elegáns, de robosztus: negatív `i`-re az üres listát adja eredményül. Majdnem ugyanez valamivel szebben:

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

`take` második változata negatív `i`-re *a teljes listát* visszaadja. (Miért?) *Figyelem:* ebben a definícióban az ágak sorrendje nem közömbös! (Miért nem?)[8]

Nézzünk egy példát `take` egyszerűsítésére (egyes triviális lépéseket összevonunk)!

```
take([9,8,7,6],3)→9::take([8,7,6],2)→9::8::take([7,6],1)
    →···→9::8::7::[]→9::8::[7]→9::[8,7]→[9,8,7]
```

Az egyszerűsítési folyamatot bemutató példában a négyespontot (`::`) nem lista létrehozására használjuk, mint a programokban, hanem olyan listakifejezéseket írunk fel vele, amelyeket az egyszerűsítés során az

[7] Egy eljárás, függvény, program stb. akkor robosztus, ha szélsőséges körülmények között is a specifikációjának megfelelően, megbízhatóan, kiszámíthatóan viselkedik. Szélsőséges körülménynek számít például, ha egy függvényt ritkán előforduló, extrém értékre alkalmazunk.

Az SML-ben egy függvény robosztussága azt jelenti, hogy a függvény az értelmezési tartományába eső minden lehetséges argumentumra specifikiálva van, és e specifikáció szerint viselkedik. Például a belső `hd` és `tl` függvény, ha üres listára alkalmazzuk, meghatározott *kivételt* jelez az SML-ben. A kivételkezelésről később lesz szó.

[8] Robosztusnak tekinthető-e `take` itt bemutatott két változata? Abban az értelemben igen, hogy `i` minden lehetséges értékére definiálva vannak: a rekurzió minden esetben biztosan befejeződik; ha `i` negatív, az első az üres listát, a második az eredeti listát adja eredményül. Ugyanakkor ennek a robosztus megoldásnak hátránya is van: valószínű, hogy `i<0`-ra szándékosan ritkán fogják alkalmazni `take`-et, ha pedig valamilyen hiba folyományaként lesz negatív az `i`, az eredeti hiba hatása majd csak jóval később, a program egy egészen más pontján jelentkezik, és ezért nehezebb lesz felderíteni. Ezért a könyvtári `List.take` (és `List.drop`) `i<0`-ra *kivételt* jelez.

SML-értelmezőnek ki kell értékelnie. A lista elemeit az SML-értelmező előbb egyesével berakja a verembe, majd hátulról visszafelé haladva megint elővesz őket az eredménylista előállításához.

Érdemes-e megírni `take` iteratív változatát? Próbáljuk meg: a részeredményeket gyűjtsük az egyik argumentumban.

```
(* rtake(i, xs, zs) =
   rtake :  int * ´a list * ´a list -> ´a list
*)
fun rtake (_, [], taken) = taken
  | rtake (i, x::xs, taken) =
      if i>0 then rtake(i-1, xs, x::taken) else taken;
```

Van-e valami furcsa ebben a megoldásban? Igen, van: az `x::taken` művelet miatt a listaelemek sorrendje megfordul! Ha ez nem engedhető meg, nem nyerünk semmit, mert a visszafordítása legalább ugyanannyiba kerülne, mint a `take` végrehajtása.

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

Ez a megoldás kézenfekvő és szerencsére iteratív is. Az else ágban lévő `x::xs` lista ugyanaz, mint a `drop` második argumentuma; később bemutatunk tömörebb jelölést – a *réteges mintát* –, amely ilyen esetekben alkalmazható.

4.7. Listák összefűzése és megfordítása

Ebben a szakaszban két függvényt, az `append`-et és a `rev`-et mutatjuk be. Az `append` *infix* változatát a `@` jellel jelöljük. A két listát egybefűző `append` így specifikálható:

$$[x_1, \ldots, x_m] mathbin@ [y_1, \ldots, y_n] = [x_1, \ldots x_m, y_1, \ldots y_n]$$

Az `xs`-t először az elemeire bontjuk, majd hátulról visszafelé haladva fűzzük az elemeket az `ys`-hez, ugyanis a listákat csak elölről tudjuk felépíteni.

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

Infix változatát pl. így definiálhatjuk:

```
- infix 5 @;
- val @ = append;
```

Itt is, akárcsak `take`-nél, a lista építésének költsége meghaladja a veremhasználat – ti. az adatok veremben való tárolásának – költségét, ezért nem érdemes iteratív változatot kidolgozni.

A Pascal, a C explicit mutatókkal kezeli a listákat, ezért az egyik lista végén a mutató átirányítható a másik listára. Az ilyen, ún. destruktív frissítés gyorsabb, mint a másoló frissítés. Csakhogy ez veszélyes lehet! Pl. mi van akkor, ha mindkét argumentum ugyanarra a listára mutat?

Most nézzük a listát megfordító `rev` egy naív megoldását:

```
(* nrev xs = xs megfordítva
   nrev :  ´a list -> ´a list
```



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

és egy példát `nrev` redukciójára:

```
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]→...
```

Ez eddig n lépés volt. A továbbiakban az aktuális bal szélső listát megint elemeire kell bontani, majd összerakni $0, 1, 2, \ldots, n-1$ lépésben.

`nrev` nagyon rossz hatékonyságú: $O(n^2)$. De emlékezzünk csak vissza `rtake`-re: ott megfordult a listaelemek sorrendje, bár nem akartuk. Most *pontosan ezt* akarjuk, használjunk tehát segédargumentumot a `revto` segédfüggvényben:

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

`revto` lépésszáma arányos a lista hosszával. Segítségével `rev` definíciója (`revto` lokális is lehetne `rev`-ben):

```
(* rev xs = xs megfordítva
   rev :  ´a list -> ´a list
*)
fun rev xs = revto (xs, []);
```

Egy 1000 elemű listát `rev` 1000 lépésben, `nrev` $\frac{1000 \cdot 1001}{2} = 50500$ lépésben fordít meg. Hatalmas a nyereség!

4.8. Listákból álló lista, párokból álló lista

Ebben a szakaszban megint három függvényt definiálunk. `flat` kétszeres mélységű listából egyszeres mélységűt készít; `combine` két azonos hosszúságú lista elemeiből egyetlen, párokból álló listát állít elő; `split` pedig `combine` inverz függvénye.

`flat` specifikációja és definíciója:

$$mboxflat([x_1, x_2, \ldots, x_m], [y_1, y_2, \ldots, y_n]) = [x_1, x_2, \ldots, x_m, y_1, y_2, \ldots, y_m]$$

```
(* flat xss = a kétszeres mélységű xss lista részlistáinak
              elemeiből képzett lista
   flat :  ´a list list -> ´a list
*)
fun flat [] = []
  | flat (ls::lss) = ls @ flat lss;
```

Az algoritmus elég gyors, ha `ls` jóval rövidebb `lss`-nél. `combine` specifikációja és definíciója:

$$mboxcombine([x_1, x_2, \ldots, x_m], [y_1, y_2, \ldots, y_m]) = [(x_1, y_1), (x_2, y_2), \ldots, (x_m, y_m)]$$

```
(* combine(xs,ys) = az xs és ys elemeiből képzett párok listája
   combine :  ´a list * ´b list -> (´a * ´b) list
*)
```

```
fun combine ([], []) = []
  | combine (x::xs, y::ys) = (x,y)::combine(xs, ys);
```

Az SML-értelmező figyelmeztet: nem fedtünk le minden esetet (ui. az argumentumként átadott listák különböző hosszúságúak lehetnek). Az ilyen esetek kezelésére a később ismertendő *kivételkezelés* alkalmas.

`split`-et `combine` inverz függvényeként definiáljuk:

```
(* split xys = a párok listájából előállított listapár
   split :  (´a * ´b) list -> ´a list * ´b list
*)
fun split [] = ([], [])
  | split ((x, y)::pairs) =
      let val (xs, ys) = split pairs
      in (x::xs, y::ys)
      end;
```

Iteratív változata segédargumentumokban gyűjti külön-külön a két listát, de sajnos megfordítva:

```
(* rsplit :  (´a * ´b) list * ´a list * ´b list ->
                                      ´a list * ´b list
*)
fun rsplit ([], xs, ys) = (xs, ys)
  | rsplit ((x, y)::pairs, xs, ys) = rsplit(pairs, x::xs, y::ys);
```

5. Részlegesen alkalmazható függvények

Tudjuk, hogy az SML-ben egy függvénynek csak egyetlen argumentuma van, de ez egy pár, egy ennes, egy másik függvény stb. is lehet. Több argumentumú függvényt olyan függvénnyel is megvalósíthatunk, amely függvényt ad eredményül.

A *részlegesen alkalmazható* (partially applicable) függvényeket H. B. Curry amerikai matematikus után *curried* függvényeknek is nevezik, noha a jelölést egy másik amerikai matematikusnak, Schönfinkelnek köszönhetjük. Egy részlegesen alkalmazható függvény argumentumait egymástól egy vagy több szóköz jellegű karakterrel kell elválasztani.

Nézzük a következő függvénydefiníciókat:

```
(* prefix pre post = pre és post konkatenációja
*)
- fun prefix pre post =
        let fun cat post = pre ^ post
        in cat post
        end;
> val prefix = fn :  string -> (string -> string)

- val prefix = fn pre => (fn post => pre ^ post);
> val prefix = fn :  string -> (string -> string)
```

A két definíció ekvivalens, mindkettő a részlegesen alkalmazható `prefix` függvényt definiálja. A függvény típusát leíró *típuskifejezésből* kiolvasható, hogy ha a `prefix` függvényt `string` típusú argumentumra alkalmazzuk, *függvényt* ad eredményül, amely ugyancsak `string` típusú argumentumra alkalmazható és `string` típusú értéket ad eredményül.

Egy részlegesen alkalmazható függvény alkalmazható csak az első, az első és a második stb. argumentumára. A részleges alkalmazás eredménye maga is *függvény.*

A részlegesen alkalmazható `prefix` függvény *kétargumentumú függvényként* viselkedik. Nézzünk néhány példát a használatára:


```
- prefix "Sir ";
> val it = fn :  string -> string
- it "Georg Solti"
> val it = "Sir Georg Solti" :  string

- val knightify = prefix "Sir "
- val dukify = prefix "The Duke of "
- val lordify = prefix "Lord "
```

`prefix` fenti definíciói nehézkesek, az alábbi változat jóval olvashatóbb:

```
- fun prefix pre post = pre ^ post;
> val prefix = fn :  string -> (string -> string)
```

Természetesen a részlegesen alkalmazható függvények is lehetnek rekurzívak, pl.

```
(* replist n x = n db x értékből álló lista
   replist :  int -> ´a -> ´a list
*)
fun replist 0 x = []
  | replist n x = x::replist (n-1) x
```

`replist` olyan függvény, amelyet `int` típusú értékre alkalmazva olyan függvényt kapunk, amely `´a` típusú értékre alkalmazva `´a list` típusú értéket ad eredményül. Gyűjtőargumentummal javíthatunk `replist` hatékonyságán:

```
fun replist n x =
      let fun rpl 0 xs = xs
            | rpl n xs = rpl (n-1) (x::xs)
      in rpl n []
      end
```

Összefoglalva, a függvényalkalmazás olyan $E\,E_1$ alakú összetett kifejezés, amelyben az E függvényértéket eredményező, az E_1 pedig tetszőleges kifejezés. Az $E\,E_1\,E_2 \cdots E_n$ kifejezés nem más, mint a $(\cdots((E\,E_1)\,E_2)\cdots E_n)$ kifejezés rövidítése.

Mint tudjuk, az SML-értelmezők a kifejezéseket *balról jobbra* haladva értékelik ki. A függvényalkalmazás *erősen köt,* precedenciája a lehető legnagyobb. A -> típusoperátor (a *leképezés* jele) *jobbra köt,* ezért például a `string -> (string -> string)` típuskifejezés ekvivalens a `string -> string -> string` típuskifejezéssel (az SML-értelmezők a típuskifejezést redundáns zárójelek nélkül írják ki a képernyőre).

A részlegesen *nem* alkalmazható függvényt *uncurried* függvénynek is nevezik. Ha egy részlegesen nem alkalmazható függvény típusa `(´a * ´b) -> ´c`, akkor vele ekvivalens, részlegesen alkalmazható változatának a típusa `´a -> (´b -> ´c)`.

6. Magasabb rendű függvények

A magasabb rendű függvények (angolul *higher-order functions* vagy *functionals)* többek között arra használhatók, hogy előre definiált magasabb rendű függvények alkalmazásával elkerüljük az explicit rekurziót, ezáltal könnyebben olvasható és bizonyítható programokat írjunk.

6.1. `secl` és `secr`

Gyakran hasznos, ha egy *infix* operátor egyik operandusát rögzítjük, a másikat szabadon hagyjuk, például

`("Sir " ^)` ekvivalens a `knightify` függvénnyel,

(/ 2.0) olyan függvény, amely `2.0`-vel oszt.

Sajnos, ezek a jelölések így nem használhatók az SML-ben, `secl` és `secr` segítségével azonban éppen ilyen függvényeket definiálhatunk. A nevet záró `l`, ill. `r` betű arra utal, hogy a bal *(left),* ill. a jobb *(right)* operandust kötjük-e le.

```
(* secl x f y = f bal oldali argumentumát lekötő függvény
   secl : ´a -> (´a * ´b -> ´c) -> ´b -> ´c
*)
fun secl x f y = f(x, y)

(* secr f y x = f jobb oldali argumentumát lekötő függvény
   secr : (´a * ´b -> ´c) -> ´b -> ´a -> ´c
*)
fun secr f y x = f(x, y)
```

`secl` és `secr` paramétereinek nevét és sorrendjét úgy választottuk meg, hogy egyrészt utaljanak a paraméterek szerepére és pozíciójára, másrészt tegyék lehetővé `secl` és `secr` *részleges* alkalmazását. Mielőtt folytatná az olvasást, vezesse le önállóan a két függvény típusát!

```
secl : ´a -> ((´a * ´b) -> ´c) -> ´b -> ´c
secr : ((´a * ´b) -> ´c) -> ´b -> ´a -> ´c
```

Végül bemutatunk néhány példát a két függvény alkalmazására.

```
(* knightify n = a "Sir " és n konkatenációja
   knightify : string -> string
*)
val knightify = secl "Sir " op^

(* recip r = az r valós szám reciproka
   recip : real -> real
*)
val recip = secl 1.0 op/

(* halve r = az r valós szám fele
   halve : real -> real
*)
val halve = secr op/ 2.0
```

6.2. Két függvény kompozíciója

Két függvény kompozícióját általában a $\circ$ operátorral jelölik, mi az `o` betűt fogjuk használni.

```
(* f o g = az f és g függvények kompozíciója
   o : (´a -> ´b) * (´c -> ´a) -> ´c -> ´b
*)
infix o;
fun (f o g) x = f(g x)
```


6.3. `map` és `filter`

`map` egy paraméterként átadott függvényt alkalmaz egy lista minden elemére, eredménye egy új lista. `filter` egy listából összegyűjti és egy új listába fűzi azokat az elemeket, amelyek a paraméterként átadott *predikátumot* kielégítik.

```
(* map f ls = az ls elemeiből az f transzformációval
              előálló elemek listája
   map : (´a -> ´b) -> ´a list -> ´b list
*)
fun map f [] = []
  | map f (x::xs) = f x :: map f xs

(* filter p ls = ls elemei közül a p predikátumot
                 kielégítő elemek listája
   filter : (´a -> bool) -> ´a list -> ´a list
*)
fun filter p [] = []
  | filter p (x::xs) = if p x
                       then x :: filter p xs
                       else filter p xs
```

Lássunk néhány példát az alkalmazásukra!

```
- map (map (fn n => n * 2)) [[1], [2, 3], [4, 5, 6]];
> val it = [[2], [4, 6], [8, 10, 12]]: int list list
```

Két halmaz *metszete* ($S \cap T$) például így definiálható `filter`-rel (`ss`-ben S, `ts`-ben T elemeit tároljuk):

```
(* inter(ss, ts) = az ss és ts halmazok metszete
   inter : ''a list * ''a list -> ''a list
```

PRE: $\forall i, j \bullet i \neq j,\ s_i \in ss,\ s_j \in S \bullet s_i \neq s_j, \forall i, j \bullet i \neq j,\ t_i \in T,\ t_j \in T \bullet t_i \neq t_j$

```
*)
fun inter (ss, ts) = filter (secr (op isMem) ts) ss
```

6.4. `takewhile` és `dropwhile`

Korábban `take`-kel és `drop`-pal találkoztunk: `take` egy lista elejéről vett adott számú elemből, `drop` egy lista elejéről adott számú elem elhagyásával képez listát. Néha olyan függvényekre van szükség, amelyek egy lista elejéről vett, adott predikátumot kielégítő elemekből, ill. egy lista elejéről adott predikátumot kielégítő elemek elhagyásával képeznek listát. Ilyen függvényeket írunk most `takewhile`, ill. `dropwhile` néven.

```
(* takewhile p xs = az xs elejéről vett, p-t kielégítő
                    elemek listája
   takewhile : (´a -> bool) -> ´a list -> ´a list
*)
fun takewhile p [] = []
  | takewhile p (x::xs) =
                 if p x then x::takewhile p xs else []

(* dropwhile p xs = xs elejéről a p-t kielégítő
                    elemek elhagyásával
                    előálló lista
```

```
   dropwhile :  (´a -> bool) -> ´a list -> ´a list
*)
fun dropwhile p [] = []
  | dropwhile p (x::xs) =
                 if p x then dropwhile p xs else x::xs
```

dropwhile-ban *réteges mintát* is alkalmazhatunk:

```
fun dropwhile p [] = []
  | dropwhile p (xxs as x::xs) =
                 if p x then dropwhile p xs else xxs
```

6.5. exists és forall

exists és forall a logikából jól ismert *kvantorok* (∃, ∀) megvalósítása SML-ben:

```
(* exists p xs = igaz, ha xs-nek van p-t kielégítő eleme
   exists :  (´a -> bool) -> ´a list -> bool
*)
fun exists p [] = false
  | exists p (x::xs) = p x orelse exists p xs

(* forall p xs = igaz, ha xs összes eleme kielégíti p-t
   forall :  (´a -> bool) -> ´a list -> bool
*)
fun forall p [] = true
  | forall p (x::xs) = p x andalso forall p xs
```

A korábban már definiált isMem függvényt újradefiniálhatjuk exists alkalmazásával:

```
(* x isMem xs = igaz, ha x eleme xs-nek
   isMem :  ''a * ''a list -> bool
*)
infix isMem;
fun x isMem xs = exists (secl x op=) xs
```

forall segítségével definiálhatjuk a halmazok diszjunkt voltát tesztelő disjoint függvényt:

```
(* disjoint(xs, ys) = igaz, ha xs és ys metszete üres
   disjoint :  ´a list * ´a list -> bool
*)
fun disjoint (xs, ys) =
         forall (fn x => forall (fn y => x <> y) ys) xs
```

Az utóbbit első ránézésre elég nehéz megérteni. Próbáljuk meg együtt! A függvénynek akkor kell igaz értéket adnia, ha az xs és az ys által ábrázolt halmazoknak egyetlen közös eleme sincs.

Az `fn y => x <> y` függvény akkor ad igaz értéket, ha y nem egyenlő valamilyen – e függvény számára külső és rögzített – x értékkel. Ezt a függvényt a zárójelen belüli forall hívás az összes ys-beli értékkel meghívja, és akkor ad igazat eredményül, ha egyetlen ys-beli érték sem egyenlő x-szel.

Az `fn x => forall ··· ys` függvény vezeti be x-et mint argumentumot. A külső forall az összes xs-beli értékkel meghívja ezt a függvényt, és akkor igaz az eredménye, ha nincs olyan ys-beli érték, amely egyenlő lenne valamely xs-beli értékkel.

6.6. foldl és foldr

foldl balról jobbra (**l**eft to right), foldr jobbról balra (**r**ight to left) haladva egy *kétargumentumú prefix* függvényt (pl. op+, op*) alkalmaz egy lista minden elemére. A két függvény specifikációját *infix* operátorral (⊕) írjuk föl, mert *infix* jelöléssel könnyebb megérteni a működésüket. (⊕ tetszőleges infix operátort helyettesít.)

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

Itt e az ún. egységelem. Látható, hogy ha elvégezzük a kijelölt műveleteket, mindkét függvény *egyszerűsíti* az eredeti kifejezést. Kifejezéseket általában – de nem mindig! – jobbról balra haladva egyszerűsítünk, ezért foldr-t néha reduce néven definiálják.

Asszociatív műveletek esetén mindegy, hogy foldl-t vagy foldr-t alkalmazzuk-e, *nemasszociatív műveletek* esetén azonban az *egységelem kötésétől* függ a dolog: ha $e \oplus x = x$, az egységelem bal oldali, ha $x \oplus e = x$, az egységelem jobb oldali. Ha mindkét egyenlőség fennáll – és asszociatív műveletek esetén ez a helyzet -, egyszerűen egységelemről beszélünk.

Közismert, hogy az összeadás és a szorzás asszociatív műveletek, a kivonás és az osztás azonban nem, ezért nem mindegy, hogy pl. $x - 0$ vagy $0 - x$ lesz-e a kivonás egyszerűsítésének az eredménye. E kis példából az is látható, hogy a kivonásnak – és hozzá hasonlóan az osztásnak – jobb oldali egységeleme van.

foldl-t és foldr-t nem specifikáltuk arra az esetre, amikor a lista üres, pótoljuk:

foldl op⊕ e [] = e és foldr op⊕ e [] = e

Jól látszik, hogy e valóban az op⊕ művelet egységeleme, hiszen op⊕-t az üres listára alkalmazva e-t kapjuk eredményül. Most már definiálhatjuk foldl és foldr SML-változatát.[9]

```
(* 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

(* 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
```

Mindkét függvénynek `´a * ´b -> ´b` típusú függvény az első argumentuma. Vegyük észre, hogy foldl *iteratív* függvény: második argumentuma, amely kezdetben az egységelem, gyűjtőargumentumként viselkedik. Sajnos, foldr a veremben gyűjti a részeredményeket, és majd csak a lista kiürülésekor hajtja végre a kijelölt műveleteket.

A két függvény definícióját és típusát nem is olyan könnyű megjegyezni. Javasoljuk az olvasónak, hogy csukja be a jegyzetet, és próbálja meg *a specifikáció alapján* rekonstruálni a két definíciót és levezetni a típust!

Számos függvény írható fel foldl és foldr alkalmazásával, ha megadjuk a lista szomszédos elemein végrehajtandó műveletet és az egységelemet. Lássunk néhányat:

```
(* sum xs = xs elemeinek összege
   sum :  int list -> int
*)
```

[9] Mindkettő belső függvény az SML-ben.

```
fun sum xs = foldl op+ 0 xs

(* prod xs = xs elemeinek szorzata
   prod :  int list -> int
*)
fun prod xs = foldl op* 1 xs

(* flat xss = az xss részlistáinak konkatenálásával
              előálló lista
   flat :  ´a list list -> ´a list
*)
fun flat xss = foldl op@ [] xss
```

A `length` függvény egy iteratív változata (`inc` olyan kétargumentumú segédfüggvény, amelyik nem használja a második argumentumát):

```
local (* inc(n,_) = n+1
        inc :  int * ´a -> int
     *)
     fun inc (n, _) = n + 1
in
   (* length ls = az ls lista hossza
      length :  ´a list -> int
   *)
   fun length ls = foldl inc 0 ls
end
```

Az `append` függvény is felírható így, de `foldr`-rel, mert a `::` művelet nem asszociatív és jobbra köt. Jobb "egységelemnek" azt a listát vesszük, amelyhez a bal oldali lista elemeit – jobbról balra haladva – egyesével fűzzük hozzá .

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

A beszúró rendezés egy újabb változata `foldr`-rel:

```
(* ins(x, ys) = az ys rendezett egészlista elemeiből és az
                ys-be a <= reláció szerint beszúrt x-ből
                álló lista
   ins :  int * int list -> int list
*)
fun ins (x, []) = [x]
  | ins (x, y::ys) =
              if x <= y then x::y::ys else y::ins(x, ys)

(* inssort ls = az ls egészlista elemeinek <= szerint
                rendezett listája
   inssort :  int list -> int list
*)
fun inssort xs = foldr ins [] xs
```


6.7. `curry` és `uncurry`

Most már könnyen definiálhatunk egy-egy olyan függvényt, amelyik egy részlegesen alkalmazható függvényt részlegesen nem alkalmazható függvénnyé, ill. egy részlegesen nem alkalmazható függvényt részlegesen alkalmazható függvénnyé alakít:

```
(* curry f x y = f részlegesen nem alkalmazható alakban
   curry :  (´a * ´b -> ´c)-> ´a -> ´b -> ´c
*)
fun curry f x y = f(x,y)

(* uncurry f(x, y) = f részlegesen alkalmazható alakban
   uncurry :  (´a -> ´b -> ´c) -> (´a * ´b) -> ´c
*)
fun uncurry f(x,y)= f x y
```

6.8. További rekurzív függvények

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

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

Sok függvény fejezhető ki `repeat` segítségével. Példák:

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

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