# BEVEZETÉS A FUNKCIONÁLIS PROGRAMOZÁSBA



## Az előadássorozat áttekintése

- Bevezetés. Az SML nyelv alapjai.
- Egyszerű és összetett adattípusok. Programfejlesztés.
- Polimorfizmus. Listaműveletek. A legfontosabb programkönyvtárak.
- Programhelyesség, programbizonyítás.
- Magasabbrendű függvények.
- Modulok. Absztrakt adattípusok. Paraméterezhető modulok.
- Nemlineáris rekurzív adattípusok.
- Nagyobb SML-példák.
- Új irányzatok a funkcionális programozásban.





## A funkcionális programozás motivációi

- Rekurzió, teljes indukció (vö. gépi kód, Fortran, Basic) – 1950-es évek
- Lineáris rekurzív adatszerkezet (lista, vö. ciklus)
- Függvények – vissza a matematikához! (vö. mellékhatás) – 1960-as évek
- Erős típusok, ellenőrzés fordításkor (vö. típusnélküli nyelvek) – 1970-es évek
- Rekurzív adattípusok (fa, vö. láncolt adatszerkezetek)
- Absztrakt adattípusok (vö. objektumok)
- Végrehajtható specifikációk (vö. tesztelés) – 1990-es évek

Mi az alapvető különbség a deklaratív és az imperatív programozás között?

- A deklaratív programozás *időtlen*, nem törődik az idővel.
- Idő $\longrightarrow$ állapot $\longrightarrow$ emlékezet.





## A funkcionális programozás rövid története

- A függvényfogalom fejlődése – l. külön fóliákon: `fffp.pdf`.
- Euler (1748): $\sin.x$ később $\sin x$ vagy $\sin(x)$
- Alfred N. Whitehead, Bertrand Russel (1910) ... Alonzo Church: $\lambda$-***kalkulus,*** $\lambda$-jelölés: $\lambda x.x + x$
- Church, 1936: $\lambda$-kalkulus (funkcionális) $\equiv$ Turing-gép (imperatív) $\longrightarrow$ funkcionális programozás $\equiv$ imperatív programozás
- Church-tétel: kiszámítható függvények halmaza $\equiv$ rekurzív függvények halmaza – ez a funkcionális programozás alapja
- 1960: ALGOL (ALGOrithmic Language) – rekurzív eljárás és függvényeljárás (!)
- 1960: LISP (LISt Processing language) – alapja a $\lambda$-kalkulus, eredeti célja: *szimbolikus differenciálás*
- 1962-től: APL, ML, HOPE, ERLANG, Miranda, SML, Haskell, gofer, clean stb.

## Az ML (Meta Language) rövid története és jelene

### Az ML rövid története

- ML, Edinborough 1977, tételbizonyításra (kijelentések igazolására)
- Definition of Standard ML, 1990
  - Alapnyelv (Core Language)
  - Modulnyelv (Module Language)
- Revised Definition of Standard ML, 1997
- SML Basis Library (Alapkönyvtár), 1997

### SML-megvalósítások

- Moscow ML (mosml): `http://www.dina.kvl.dk/ sestoft/mosml.html`
- Standard ML of New Jersey (sml):
  `http://cm.bell-labs.com/cm/cs/what/smlnj`





## Információk a funkcionális programozásról

### Hálózati információforrások:

Comp.Lang.ML FAQ

`http://www.cis.ohio-state.edu/hypertext/faq/usenet/meta-lang-faq/faq.html`

Andrew Cumming: A Gentle Introduction to ML

`http://www.dcs.napier.ac.uk/course-notes/sml/manual.html`

Stephen Gilmore: Programming in Standard ML '97

`http://www.dcs.ed.ac.uk/home/stg`

Robert Harper: Programming in Standard ML

`http://www.cs.cmu.edu/People/rwh/`

Fox project at CMU

`http://foxnet.cs.cmu.edu/sml.html`





## SML-irodalom (csak angolul)

Forrásművek az előadásokhoz

Jeffrey D. Ullman: *Elements of ML Programming* (2nd Edition, ML97)
MIT Press 1997
`http://www-db.stanford.edu/ ullman/emlp.html`

Lawrence C. Paulson: *ML for the Working Programmer* (2nd Edition, ML97)
Cambridge University Press 1996
`http://www.cl.cam.ac.uk/users/lcp/MLbook/`

Richard Bosworth: *A Practical Course in Functional Programming Using Standard ML*
McGraw-Hill 1995



# A FÜGGVÉNY FOGALMA ÉS TULAJDONSÁGAI

## A típus és a függvény fogalma

- A típus fogalma
  - A típus értékek egy halmaza (pl. egész típus = az egész számok halmaza)
  - Jelölése: $\alpha$, $\beta$, ...(az ún. *típuselméletben* így használják)
- A függvény fogalma
  - A függvény valamely $D$ halmaznak valamely $R$ halmazba való olyan *egyértelmű* leképzése, amelyet meghatároz a $(d; r)$ rendezett párok halmaza, ahol $d \in D$ és $r \in R$.
  - A $d$ a függvény argumentuma (paramétere), az $r$ az eredménye
  - A $D$ a függvény értelmezési tartománya, az $R$ az értékkészlete
  - A típusos nyelvekben $d$ is, $r$ is *meghatározott* típusú
  - Függvény értelmezési tartománya $\subseteq$ argumentum típusa
  - Függvény értékkészlete $\subseteq$ eredmény típusa





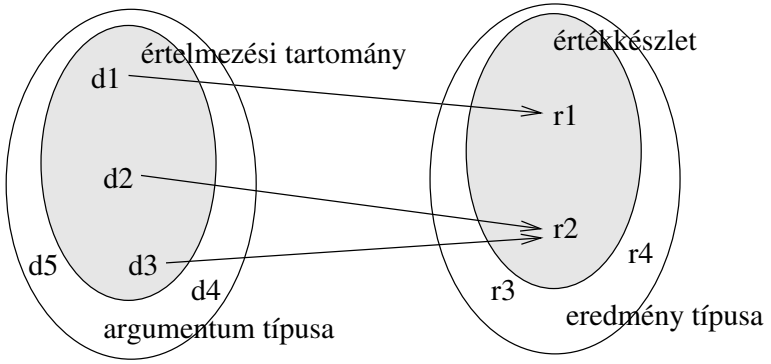
## A függvény mint érték

- A függvény „teljes jogú” *(first-class)* érték a funkcionális programozási nyelvekben
  - A függvény típusa általában: $\alpha \to \beta$, ahol az $\alpha$ az argumentum, a $\beta$ az eredmény típusát jelöli
  - A függvény – érték: *függvényérték*
  - Fontos: a függvényérték *nem* a függvény *alkalmazásának* az eredménye!
  - Példák függvényértékre
    - sin (a típusa: *valós* $\to$ *valós*)
    - round (a típusa: *valós* $\to$ *egész*)
    - $f \circ g$ (a típusa: $\alpha \to \beta$)
  - Példa függvényalkalmazásra
    - round $5.4 = 5$, azaz ennek a függvényalkalmazásnak egy *egész* típusú érték az eredménye





## A függvény mint leképzés








## Függvények tulajdonságai és osztályozása

- Parciális függvény: értelmezési tartomány $\subset$ argumentum típusa

  Figyelem: ez hibák forrása lehet!
- Teljes függvény: értelmezési tartomány = argumentum típusa
- Szürjektív függvény: értékkészlet = eredmény típusa
- Nem-szürjektív függvény: értékkészlet $\subset$ eredmény típusa
- Injektív függvény: a leképzés kölcsönösen egyértelmű
- Az $f : \alpha \to \beta$ injektív függvény inverze: $f^{-1} : \beta \to \alpha$
- Bijektív = injektív + szürjektív, azaz $f$ bijektív, ha $f^{-1}$ teljes függvény

## Függvények alkalmazása

- ***Függvényalkalmazást*** jelöl az $f$ és $e$ jelek egymás mellé írása (*„juxtapozicionálása”*): $f\,e$ azt jelenti, hogy $f$-et alkalmazzuk $e$-re.
- Általánosabban: az $fe$ kifejezésben az $e$ tetszőleges olyan kifejezés, amelynek az értéke az $f$ értelmezési tartományába esik.
- Még általánosabban: az $fe$ kifejezésben az $f$ függvényértéket eredményező tetszőleges kifejezés, $e$ pedig tetszőleges olyan kifejezés, amelynek az értéke az $f$ értelmezési tartományába esik.





## Két- vagy többargumentumú függvények

- Függvény alkalmazása két- vagy több argumentumra
  1. Az argumentumokat ***összetett adatnak*** – párnak, rekordnak, listának stb. – tekintjük, pl. $f(1,2)$
     - az $f$ függvény alkalmazását jelenti az $(1,2)$ párra.
  2. A függvényt több egymás utáni lépésben alkalmazzuk az argumentumokra, pl. $f12 \equiv (f1)2$ azt jelenti, hogy
     - az első lépésben az $f$ függvény alkalmazzuk az $1$ értékre, ami egy függvényt ad eredményül,
     - a második lépésben az első lépésben kapott függvényt alkalmazzuk a $2$ értékre, így kapjuk meg az $f12$ függvényalkalmazás (vég)eredményét.
- Infix jelölés: $x \oplus y \equiv$ az $\oplus$ függvény alkalmazása az $(x,y)$ párra mint argumentumra



# FÜGGVÉNYEK AZ SML-BEN



## Függvények alkalmazása az SML-ben

- Az SML-ben az `f` és az `e` tetszőleges ***név*** lehet, amelyeket megfelelően ***szeparálni*** kell egymástól: `f e`, vagy `f(e)`, vagy `(f)e`
- Szeparátor: nulla, egy vagy több ***formázó*** karakter (␣, \t, \n stb.). Nulla db formázó karakter elegendő pl. a ( előtt és a ) után.
- A szeparátor a legerősebb balra kötő infix operátor az SML-ben.
- Példák:
  `Math.sin 1.00, (Math.cos)Math.pi, round(3.17), 2 + 3, (real) (3 + 2 * 5)`
- Függvények egy csoportosítása az SML-ben
  - Beépített függvények, pl. `+`, `*` (infix), `real`, `round` (prefix)
  - Könyvtári függvények, pl. `Math.sin`, `Math.cos`, `Math.pi`
  - Felhasználó által definiálható függvények, pl. `terulet`, `/\`, `head`

## SML-példa: Egyszeres Hamming-távolságú ciklikus kód

- A függvényt *táblázattal* adjuk meg:

| | |
|---|---|
| 00 | 01 |
| 01 | 11 |
| 11 | 10 |
| 10 | 00 |

```
fn 00 => 01
 | 01 => 11
 | 11 => 10
 | 10 => 00
```

- Változatok („klózok”): minden lehetséges esetre egy változat.
- Az `fn` (olvasd: *lambda*), névtelen függvényt, ***függvénykifejezést*** vezet be.
- A függvény néhány alkalmazása:
  - `(fn 00 => 01 | 01 => 11 | 11 => 10 | 10 => 00) 10`
  - `(fn 00 => 01 | 01 => 11 | 11 => 10 | 10 => 00) 11`
  - `(fn 00 => 01 | 01 => 11 | 11 => 10 | 10 => 00) 111`
- Mintaillesztés, egyesítés
- Érthető, de nem robusztus (vö. parciális a függvény!).





## SML-példa: modulo $n$ alapú inkrementálás

- A függvényt most *algoritmussal* adjuk meg, nem táblázattal
  - `n` nem lehetne változó, túl sok változatot kellene felírni stb.
- `fn i => (i + 1) mod n`
  - az `i` ún. kötött változó, a névtelen függvény argumentuma
  - az `n` ebben a kifejezésben szabad változó, és nincs értéke (!)
  - az `n`-et is le kell kötni mint a függvény argumentumát
- `fn i => fn n => (i + 1) mod n`
- A függvény néhány alkalmazása:
  - `(fn i => (fn n => (i + 1) mod n) 4) 1)`
  - `(fn i => (fn n => (i + 1) mod n) 128) 111`
  - `(fn i => (fn n => (i + 1) mod n) 4) ~7`
  - `(fn i => (fn n => (i + 1) mod n) 128) 6.0` – **hibás!**





## Értékdeklaráció SML-ben: függvényérték deklarálása

- Név kötése függvényértékhez
  - `val incMod = fn i => fn n => (i + 1) mod n`
  - `val kovKod = fn 00 => 01 | 01 => 11 | 11 => 10 | 10 => 00`
- Szintaktikai édesítőszerrel
  - `fun incMod n i = (i + 1) mod n`
    - Figyelem: `i` és `n` sorrendje megfordult!
  - ```
    fun kovKod 00 = 01
      | kovKod 01 = 11
      | kovKod 11 = 10
      | kovKod 10 = 00
    ```
- Alkalmazásuk argumentumra
  - `incMod 128 111`
  - `kovKod 01`





## Fejkomment

Legyen ***fejkomment*** minden (függvény)érték-deklarációhoz!

```
(* incMod n i = (i+1) modulo n szerint
   PRE: n > 0, n > i >= 0
*)
fun incMod n i = (i + 1) mod n
```

```
(* kovKod cc = a kétbites, egyszeres Hamming-távolságú, ciklikus
               kódkészlet cc-t követő eleme
   PRE: cc in {00, 01, 11, 10}
*)
fun kovKod 00 = 01
  | kovKod 01 = 11
  | kovKod 11 = 10
  | kovKod 10 = 00
```
```

# A FÜGGVÉNYFOGALOM KIALAKULÁSA



## A függvényfogalom: absztrakció

Absztrakció, oksági viszony

- A függvényfogalom minden bizonnyal olyan absztrakció, amelynek eredete az emberek által már igen régen észrevett oksági viszonyban gyökrezik. Ennek a mind tudatosabbá váló ok-okozati viszonynak már a korai matematikában jelentkeztek különböző megfogalmazásai.

Már a régi görögök, sőt már az egyiptomiak, babiloniak is...

- Lényegében ezt fejezték ki a számolást megkönnyítő egyiptomi és babiloni táblázatok. Az ógörög matematikusok a mennyiségi törvényeket kutatva, a függvényfogalmat rejtő összefüggések tömegét fogalmazták meg...





## A XVII. század – Descartes, Fermat, Leibniz, Bernoulli testvérek

Végül is a francia Descartes (René, du Peron, 1596-1650) nagy matematikai tette volt a függvényfogalom első definíciója. Csodálatos lényeglátással a függvényt megfeleltetésnek definiálta, bár ő még csak az algebrai műveletekkel meghatározott függvényekkel foglalkozott. Descartes, és vele egy időben és ugyanolyan érdemekkel, a francia Fermat (Pierre, 1601-1665) megteremtette a változó mennyiségek matematikáját.
A függvényfogalom további alakítása a német Leibniz (Gottfried Wilhelm, 1646-1716) nevéhez fűződik, de az ő értelmezése szűkebb Descartes-énál. Ő használta először 1692-ben a latin *functio* szót valamely görbe egy pontjához tartozó olyan szakaszra, amely változik, ha a pont végigfut a görbén (ordináta, abszcissza, szubtangens stb.). Ekkor vezette be a paraméter, az állandó, a változó és más kifejezéseket is.
A XVII. század végétől, a XVIII. század elejétől függvénynek tekintették azt az analitikus kifejezést, amely kifejezte a változók és az állandók közötti kapcsolatot. Ilyen értelemben használta a függvény szót a két svájci Bernoulli testvér (Jacob 1654-1705, Johann 1667-1748), és ezt a fogalmat Johann B. zárójel nélkül $\varphi x$-szel jelölte.





## A XVIII. század – Euler

Ezt az értelmezést vette át a svájci Euler (Leonhard, 1707-1783) is, aki a $\varphi$ betű helyett az $f$-et kezdte használni, és megengedte a komplex változókat is. Euler szerint tehát függvényen értjük a változók és a konstansok közötti kapcsolatot leíró kifejezést, ha az analitikus műveleteket (négy alapművelet, hatványozás, gyökvonás, sorbafejtés, differenciálás, integrálás) tartalmaz. Ez még mindig a függvényeknek csak azt az osztályát ölelte át, amelyeket teljes értelmezési tartományukban már meghatároz grafikonjaik bármilyen kicsiny darabja. Ezenkívül azonban Euler foglalkozott az $e^x$, az $\ln x$ és a trigonometrikus függvényekkel is...
Euler kezdetben azt hitte, hogy minden függvény hatványsorba, azaz

$$f z = a_0 + a_1 z + a_2 z + a_3 z + \cdots$$

alakba fejthető. A differenciálegyenletek vizsgálatánál azonban olyan függvényekre bukkant, amelyeket megadhatott tetszőleges alakú grafikon is. Ezekről azt gondolta, hogy nem analítikus függvények, azaz nem fejthetők hatványsorba...

## A XIX. század – Bolzano, Dirichlet, Cauchy, Weierstrass, Gauss, Riemann

A függvényelmélet voltaképpen a cseh Bolzano (Bernhard, 1781-1848) és a német Dirichlet (Peter Gustav Lejeune, 1805-1859) eredményei alapján indult igazán fejlődésnek, és a XIX. században az előzmények biztos talaján a francia Cauchy (Augustin Louis 1789-1857) és a német Weierstrass (Karl, 1815-1897) a legnagyobb szabatossággal önthette szavakba a függvénytani tulajdonságokat és fogalmakat.
A valós függvénytan kialakulása után a komplex változós függvénytan is biztos alapokra talált a komplex számoknak a német Gauss (Carl Friedrich, 1777-1855) alkotta elmélete segítségével.
A komplex változójú függvények elméletének megteremtésében Cauchy és Weierstrass mellett nagy szerepe volt a német Riemann-nak (Georg Friedrich Bernhard, 1826-1866) is, aki a geometriai függvénytan életre hívásával a komplex függvénytan új megalapozását tette lehetővé.





## A XX. század első fele – Volterra, Fréchet, Riesz, Hilbert

A függvényfogalom halmazelméleti definíciója nem teszi szükségessé, hogy a megfeleltetés számok halmazait kapcsolja össze. Az értelmezési tartomány és az értékkészlet elemei tetszőleges matematikai objektumok lehetnek.
Az absztrakciónak ez a további fokozata a függvénytan új területeit hozta létre. Ha az értelmezési tartomány függvények halmaza és az értékkészlet számok halmaza, akkor az egymáshoz rendelést funkcionálnak nevezzük. Ha pedig az értelmezési tartomány és az értékkészlet is függvények halmaza, akkor a megfeleltetés neve operátor.
A funkcionálokkal és az operátorokkal foglalkozó funkcionálanalízis különálló kutatási területnek az olasz Volterra (Vito, 1860-1940) munkássága óta számít.
A funkcionálelmélet kibontakozásában kimagasló érdemei vannak a francia Fréchet-nek (Maurice, 1878-1973), a magyar Riesz Frigyesnek (1880-1956) és a német Hilbertnek (David, 1862-1943).





## A függvény

A függvény két halmaz között olyan megfeleltetés, amely az egyik halmaz minden eleméhez hozzárendeli egy másik halmaz pontosan egy elemét.
Formálisabb megfogalmazások:

- A függvény olyan $(x; y)$ rendezett párok halmaza, amelyben minden $x$-hez pontosan egy $y$ tartozik.
- A függvény olyan bináris reláció, amelyre igaz, hogy ha $(x; y)$ és $(x; y')$ mindegyike eleme a relációnak, akkor $y = y'$.
- A függvény valamely $X$ halmaznak valamely $Y$ halmazba való olyan egyértelmű leképzése, amelyet meghatároz az $(x; y)$ rendezett párok halmaza, ahol $x \in X$ és $y \in Y$.

Irodalom


- Sain Márton: Nincs királyi út! Matematikatörténet. Gondolat, Budapest, 1986., pp. 697-702.
- Sain Márton: Matematikatörténeti ABC. Tankönykiadó, Budapest, 1987., p. 122.




# TÍPUSOK ÉS ÉRTÉKEK AZ SML-BEN

## Típusok

- Típusok és programozási nyelvek
  - Típus nélküli nyelvek, pl. assembly, LISP, Prolog
  - Gyengén típusos nyelvek, pl. Fortran, Algol, BASIC, C, C++, Pascal
  - Erősen típusos nyelvek, pl. Ada, SML, clean
  - Erős típus: a típusok ($\sim$ halmazok) diszjunktak (nincs közös elemük)
- Egyszerű SML-típusok
  - `int` – előjeles egész szám, a $Z$ egy részhalmaza
  - `word`, `word8` – előjel nélküli pozitív egész, az $N_0$ egy részhalmaza
  - `real` – előjeles racionális (valós?!) szám, a $Q$ egy részhalmaza
  - `bool`, `char`, `order`, `unit`
  - `string`
- Összetett SML-típusok (példák)
  - `rekord`
  - `lista`





## Értékdeklaráció az SML-ben: név kötése tetszőleges értékhez

- Függvényértéket így kötöttünk tetszőleges névhez:
  `val incMod = fn i => fn n => (i + 1) mod n`
- Tetszőleges típusú érték köthető tetszőleges névhez:

| | | |
|---|---|---|
| `val harom = 2 + 1` | `: int` | |
| `val MHz = 94.5` | `: real` | |
| `val veege = true` | `: bool` | `true`, `false` |
| `val kisA = #"a"` | `: char` | |
| `val palindrom = "ABBA"` | `: string` | |
| `val kisebb = LESS` | `: order` | `LESS`, `EQUAL`, `GREATER` |
| `val ezNemSemmi = ()` | `: unit` | Egyetlen érték a `()`! |
| `val rat = {num = 3, den = 4}` | `: {den : int, num : int}` | Mezőnevek ábécé sorrendben. |
| `val bLista = [2,3,4] @ [3,2]` | `: int list` | |
| `val telenek = [0w123, 0wxcd]` | `: word list` | |

- Típusmegkötés:

| | |
|---|---|
| `val id = fn (n : int) => n` | Példák: `id 3;`, `id 4.5;` |
| `val telenek = [0w65, 0wx41 : word8]` | Típusa: `word8 list` |



# EGYSZERŰSÍTETT SML-SZINTAXIS



## SML-szintaxis: különleges állandó

- Előjeles egész állandó
  Példák: `0 ~0 4 ~04 999999 0xFFFF ~0x1ff`
  Ellenpéldák: `0.0 ~0.0 4.0 1E0 -317 0XFFFF -0x1ff`
- Valós állandó
  Példák: `0.7 ~0.7 3.32E5 3E~7 ~3E~7 3e~7 ~3e~7`
  Ellenpéldák: `23 .3 4.E5 1E2.0 1E+7 1E-7`
- Előjel nélküli egész állandó
  Példák: `0w0 0w4 0w999999 0wxFFFF 0wx1ff`
  Ellenpéldák: `0w0.0 ~0w4 -0w4 0w1E0 0wXFFFF 0WxFFFF`
- Füzérállandó: `"`-ek között álló nulla vagy több nyomtatható karakter, szóköz vagy \ jellel kezdődő *escape-szekvencia* (l. a táblázatot a következő lapon).
- Karakterállandó: `#` jelet közvetlenül követő, egykarakteres füzérállandó.
  Példák: `#"a" #"\n" #"\^Z" #"\255" #"\""`
  Ellenpéldák: `# "a" #c #"""`

## SML-szintaxis: escape-szekvenciák

- Escape-szekvenciák

| | |
|---|---|
| `\a` | Csengőjel (BEL, ASCII 7). |
| `\b` | Visszalépés (BS, ASCII 8). |
| `\t` | Vízszintes tabulátor (HT, ASCII 9). |
| `\n` | Újsor, soremelés (LF, ASCII 10). |
| `\v` | Függőleges tabulátor (VT, ASCII 11). |
| `\f` | Lapdobás (FF, ASCII 12). |
| `\r` | Kocsi-vissza (CR, ASCII 13). |
| `\^`$c$ | Vezérlő karakter, ahol $64 \leq c \leq 95$ (@ ... _), és `\^`$c$ ASCII-kódja 64-gyel kevesebb $c$ ASCII-kódjánál. |
| `\`*ddd* | A *ddd* kódú karakter (*d* decimális számjegy). |
| `\u`*xxxx* | A *xxxx* kódú karakter (*x* hexadecimális számjegy). |
| `\"` | Idézőjel ("). |
| `\\` | Hátratört-vonal (\). |
| `\`$f \cdots f$`\` | Figyelmen kívül hagyott sorozat. $f \cdots f$ nulla vagy több formázókaraktert (szóköz, HT, LF, VT, FF, CR) jelent. |





## SML-szintaxis: név

- Alfanumerikus: kis- és nagybetűk, számjegyek, percjelek (') és aláhúzás-jelek (_) olyan sorozata, amely betűvel vagy percjellel kezdődik
  - Példák: `tothGyorgy` `Toth_3_Gyorgy` `toth'gyorgy`
- Szimbolikus: az alábbi jelek tetszőleges, nem üres sorozata

  `! % & $ # + - / : < = > ? @ \ ~ ' ^ | *`
  - Példák: `++` `<->` `|||` `##` `|=|`
- Speciális a szerepe az alábbi fenntartott jeleknek

  `( ) [ ] { } , ; . ...`
- Más jelentés nem rendelhető az alábbi fenntartott nevekhez

```
abstype and andalso as case do datatype else end eqtype exception
fn fun functor handle if in include infix infixr let local nonfix
of op open orelse raise rec sharing sig signature struct structure
then type val where with withtype while : :: :> _ | = => -> #
```





## SML-szintaxis: szintaktikai kategóriák (egyszerűsítve)

- A nevek és más azonosítók *szintaktikai kategóriákba* sorolhatók

| | | | |
|---|---|---|---|
| *vid* | értéknév | value identifier | long |
| *tyvar* | típusváltozó | type variable | |
| *tycon* | típuskonstruktor | type constructor | long |
| *lab* | mezőnév | record label | |
| *strid* | struktúranév | structure identifier | long |
| *sigid* | szignatúranév | signature identifier | |
| *unitid* | állománynév | unit identifier | |

- Az *értéknév* tetszőleges név; jelölhet állandó értéket, függvényértéket, adatkonstruktort, kivételkonstruktort. Példák: `pi + sin nil true Match`
- A *típusváltozó* percjellel kezdődő alfanumerikus név. Példa: `'a`.
- A *típuskonstruktor* tetszőleges név; jelölhet típusállandót vagy típusfüggvény-értéket. Példák: `int order $ * -> list`
- A *mezőnév* tetszőleges név vagy (nem 0-val kezdődő) pozitív egész szám. Példák: `num` `2`





## SML-szintaxis: szintaktikai kategóriák (folyt.)

- Minden, az előző felsorolásban „long”-gal megjelölt ***X*** szintaktikai kategóriának van egy *long**X*** párja. A *long**X*** szintaktikai kategóriába tartozó nevek rövid és hosszú (ún. minősített) alakban is felírhatók. A *rövid alak* csak egy névből, a *hosszú alak* egy hosszú struktúranévből, egy pontból és egy névből áll:

| | | | | |
|---|---|---|---|---|
| *longx* | ::= | *x* | név | identifier |
| | | *longstrid.x* | minősített név | qualified identifier |

Példák:

- `explode`
- `Real.toString`
- `Int.+`
- `List.filter`

## SML-szintaxis: szintaktikai kategóriák (folyt.)

- A *struktúranév* és a *szignatúranév* a *modulnyelv* fogalomkörébe tartozó tetszőleges nevek.

  Példák: `Char` `Int` `List` `TextIO`
- Az *állománynév* a *modulnyelv* fogalomkörébe tartozó tetszőleges olyan név, amelyet az adott operációs rendszer is megenged; forráskodú vagy tárgykódú struktúra- vagy szignatúra-állományt azonosít.
- A *strid* struktúranév a `unitid.uo` tárgykódú struktúra-állományra hivatkozik, ahol `unitid` = *strid*. A `unitid.sml` struktúra-állomány fordításakor már léteznie kell a `unitid.ui` tárgykódú szignatúra-állománynak, összeszerkesztésekor pedig már léteznie kell a `unitid.uo` tárgykódú struktúra-állománynak.
- A *sigid* szignatúranév a `unitid.ui` tárgykódú szignatúra-állományra hivatkozik, ahol `unitid` = *sigid*. A `unitid.ui` tárgykódú szignatúra-állományt a `unitid.sig` forráskodú szignatúra-állomány lefordításával kell előállítani.





## Struktúra, szignatúra, tárgykódú és forráskódú állományok

Példák

- Struktúra a megfelelő szignatúrával
  - `structure Rat :> Rat = struct` *implementáció* `end`
  - `signature Rat = sig` *specifikáció* `end`
- A `Rat` struktúrát és szignatúrát tartalmazó állományok
  - `Rat.sml`: a forráskódú struktúra-állomány (a `.sml` kiterjesztés használata ajánlott, de nem kötelező)
  - `Rat.sig`: a forráskódú szignatúra-állomány (a `.sig` kiterjesztés használata kötelező)
  - `Rat.uo`: a tárgykódú struktúra-állomány (a `.uo` kiterjesztés használata kötelező)
  - `Rat.ui`: a tárgykódú szignatúra-állomány (a `.ui` kiterjesztés használata kötelező)





## Függvényjel helyzete és kötése

- Függvényjel helyzete és kötése (általában)
  - Egy függvényjel *prefix*, *infix* vagy *postfix* helyzetű lehet.
  - Az infix helyzetű függvényjelet gyakran *operátornak* nevezik.
  - Egy (infix helyzetű) operátor lehet *asszociatív* vagy *nem-asszociatív*, köthet balra vagy jobbra. Asszociatív operátor esetén a kötési iránynak nincs jelentősége.
- Infix Prolog-operátor kötése
  - `xfx` = `f` mindkét oldalán `f` csak zárójelben ismétlődhet,
  - `yfx` = `f` bal oldalán `f` zárójelezés nélkül ismétlődhet (`f` „balra köt”),
  - `xfy` = `f` jobb oldalán `f` zárójelezés nélkül ismétlődhet (`f` „ jobbra köt”).





## Függvényjel helyzete és kötése az SML-ben

- Kifejezések és típuskifejezések az SML-ben
  - Az SML-ben a szokásos kifejezések mellett vannak *típuskifejezések* is.
  - A függvényeket *értékekre*, a típusfüggvényeket *típusokra* alkalmazhatjuk.
- Függvényjel és típusfüggvényjel helyzete és kötése az SML-ben
  - Függvényjel: *prefix* vagy *infix*.
  - Típusfüggvényjel: *infix* vagy *postfix*.
  - Az *infix* helyzetű függvényjel és típusfüggvényjel (szokásos nevén operátor, ill. típusoperátor) vagy balra, vagy jobbra köt.
- Infix helyzetben csak a két beépített típusoperátor (`*` és `->`) lehet.
- A `*` balra, a `->` jobbra köt. A `*` erősebben köt, mint a `->`.
- A típusoperátorok erősebben kötnek az összes többi operátornál.

## Függvényjel helyzete és kötése az SML-ben

- Tetszőleges kétargumentumú függvényjelet lehet adott preferenciájú (infix helyzetű) operátorként deklarálni az `infix` vagy az `infixr` direktívával.
- Az `infix` balra, az `infixr` jobbra kötő operátort deklarál.
- Egy minősített nevet, vagy egy olyan nevet, amelyet az `op` direktíva előz meg, csak *prefix* helyzetben lehet alkalmazni.
- A `nonfix` direktíva az (infix helyzetű) operátort tartósan prefix helyzetűvé alakítja. (Az `op` direktíva csak átmenetileg teszi prefix helyzetűvé.)
- A $d$ 0 és 9 közötti számjegy, az operátor precedenciája (opcionális, alapértelmezés szerinti értéke 0). Nagyobb szám erősebb kötést jelent (éppen fordítva, mint a Prologban!).
- Az $\mathtt{id}_i$ tetszőleges név ($n \geq 1$).

| | | | |
|---|---|---|---|
| `infix` | $<d>$ $id_1 \ldots id_n$ | balra köt | binds to the left |
| `infixr` | $<d>$ $id_1 \ldots id_n$ | jobbra köt | binds to the right |
| `nonfix` | $id_1 \ldots id_n$ | prefix | prefix |





## A beépített operátorok és precedenciájuk az SML-ben

Az alábbi táblázatban `wordint`, `num` és `numtxt` az alábbi típusnevek helyett állnak.

`wordint` = `int`, `word`, `word8`. `num` = `int`, `real`, `word`, `word8`.

`numtxt` = `int`, `real`, `word`, `word8`, `char`, `string`.

| *Prec.* | *Operátor* | *Típus* | *Eredmény* | *Kivétel* |
|---|---|---|---|---|
| 7 | `*` | `num * num -> num` | szorzat | `Overflow` |
| | `/` | `real * real -> real` | hányados | `Div`, `Overflow` |
| | `div`, `mod` | `wordint * wordint -> wordint` | hányados, maradék | `Div`, `Overflow` |
| | `quot`, `rem` | `int * int -> int` | hányados, maradék | `Div`, `Overflow` |
| 6 | `+`, `-` | `num * num -> num` | összeg, különbség | `Overflow` |
| | `^` | `string * string -> string` | egybeírt szöveg | `Size` |
| 5 | `::` | `'a * 'a list -> 'a list` | elemmel bővített lista (jobbra köt) | |
| | `@` | `'a list * 'a list -> 'a list` | összefűzött lista (jobbra köt) | |
| 4 | `=`, `<>` | `''a * ''a -> bool` | egyenlő, nem egyenlő | |
| | `<`, `<=` | `numtxt * numtxt -> bool` | kisebb, kisebb-egyenlő | |
| | `>`, `>=` | `numtxt * numtxt -> bool` | nagyobb, nagyobb-egyenlő | |
| 3 | `:=` | `'a ref * 'a -> unit` | értékadás | |
| | `o` | `('b -> 'c) * ('a -> 'b) -> ('a -> 'c)` | a két függvény kompozíciója | |
| 0 | `before` | `'a * 'b -> 'a` | a bal oldali argumentum | |





## SML-szintaxis: nemterminális szimbólumok, nyelvtani jelölések

- Minden nemterminális szimbólumot *változatok* sorozataként definiálunk, soronként egy változattal. Üres sor üres változatot jelent.
- A $<$ és a $>$ csúcsos zárójelpárok opcionális kifejezést fognak közre.
- Bármely $X$ nemterminális szimbólumra az alábbiak szerint definiáljuk az $Xseq$ nemterminális szimbólumot:

| | | | |
|---|---|---|---|
| $Xseq$ ::= | $X$ | egyelemű sorozat | singleton sequence |
| | | üres sorozat | empty sequence |
| | $X_1$ , ... , $X_n$ | sorozat, $n \geq 1$ | sequence, $n \geq 1$ |

- A változatokat prioritásuk csökkenő sorrendjében soroljuk föl.
- A változatokat számozzuk, a példákban utalunk az alkalmazott változatra.
- A függvényjelek és operátorok általában balra kötnek, az eltérést jelezzük.
- Minden ismétlődő konstrukció (pl. a *klózsorozat)* a lehető legmesszebb terjeszkedik jobbra. Ezért pl. egy `case`-kifejezést egy másik `case`- vagy `fn`-kifejezésen, valamint egy `fun`-definíción belül zárójelbe kell tenni.





## SML-szintaxis: kifejezések és klózsorozatok (egyszerűsítve)

- Kifejezés (*exp:* expression)

| | | | | |
|---|---|---|---|---|
| (1) | *exp* ::= | *infexp* | | |
| (2) | | *exp* **:** *ty* | típusmegkötés | type constraint |
| (3) | | `raise` *exp* | kivételjelzés | raise exception |
| (4) | | `case` *exp* `of` *match* | esetszétválasztás | case analysis |
| (5) | | `fn` *match* | függvénykifejezés | function expression |

- Példák:

```
fn (n : int) => n;                                          vö. (2), (5)
case c of 00 => 01 | 01 => 11 | 11 => 10 | 10 => 00;        vö. (4), (19)
fn 00 => 01 | 01 => 11 | 11 => 10 | 10 => 00;               vö. (5), (19)
fn 00 => 01 | 01 => 11 | 11 => 10 | 10 => 00
                                  | _ => raise Domain;     vö. (3), (5), (19)
```

## SML-szintaxis: kifejezések és klózsorozatok (folyt.)

- Infix kifejezés (*infexp:* infix expression)

| | | | | |
|---|---|---|---|---|
| (6) | *infexp* ::= | *appexp* | | |
| (7) | | $infexp_1$ *id* $infexp_2$ | infix alkalmazás | infixed application |

- Applikatív kifejezés (*appexp:* applicative expression)

| | | | | |
|---|---|---|---|---|
| (8) | *appexp* ::= | *atexp* | | |
| (9) | | *appexp atexp* | (prefix) alkalmazás | (prefixed) application |

- Példák:

```
3 + 4;                          vö. (7)
Real.toString 3.56;             vö. (9)
Int.toString(round 3.56);  vö. (9), (17)
```





## SML-szintaxis: kifejezések és klózsorozatok (folyt.)

- Atomi kifejezés (*atexp:* atomic expression)

| | | | | |
|---|---|---|---|---|
| (10) | *atexp* ::= | *scon* | különleges állandó | special constant |
| (11) | | <op> *longvid* | értéknév | value identifier |
| (12) | | {<*exprow*>} | rekord | record |
| (13) | | # *lab* | rekordszelektor | record selector |
| (14) | | ($exp_1$ , $exp_2$) | pár | pair |
| (15) | | () | nullas | 0-tuple |
| (16) | | [$exp_1$, ..., $exp_n$] | lista, $n \geq 0$ | list, $n \geq 0$ |
| (17) | | (*exp*) | kifejezés zárójelben | parenthesized expr. |

- Példák:

```
1.12, #"Z", 0w123                    vö. (10)
Math.pi, false, Math.sin, sin        vö. (11)
#den {num=1, den=2}          vö. (12), (13), (18)
(2, 3.5), (), [1, 2, 3]      vö. (14), (15), (16)
```





## SML-szintaxis: kifejezések és klózsorozatok (folyt.)

- Kifejezéssor (*exprow:* expression row)

(18) *exprow* ::= *lab* = *exp* <, *exprow*>

- Klózsorozat (*match*)

(19) *match* ::= *mrule* <| *match*>

- Klóz (*mrule:* match rule)

(20) *mrule* ::= *pat* => *exp*

- Példák:

```
num=1, den=2                                  vö. (18)
00 => 01 | 01 => 11 | 11 => 10 | 10 => 00  vö. (19), (20)
```





## SML-szintaxis: deklarációk és kötések

- Deklaráció (*dec:* declaration)

| | | | | |
|---|---|---|---|---|
| (20) | *dec* ::= | `val` *tyvarseq valbind* | értékdeklaráció | value declaration |
| (21) | | `fun` *tyvarseq fvalbind* | függvénydeklaráció | function declaration |
| (22) | | `type` *typbind* | típusdeklaráció | type declaration |
| (23) | | | üres deklaráció | empty declaration |
| (24) | | $dec_1$ <;> $dec_2$ | deklaráció-sorozat | sequential declaration |
| (25) | | `infix` <*d*> $id_1$ ... $id_n$ | infix-direktíva, $n \geq 1$ | infix (left) directive |
| (26) | | `infixr` <*d*> $id_1$ ... $id_n$ | infixr-direktíva, $n \geq 1$ | infix (right) directive |
| (27) | | `nonfix` $id_1$ ... $id_n$ | nonfix-direktíva, $n \geq 1$ | nonfix directive |

- Példák:

```
val xy = "XY"; fun ++ x y = x ^ y  vö. (20), (21), (24)
type Rat = {num : int, den : int}              vö. (22)
infixr 4 ++; fun x ++ y = x ^ y          vö. (21), (26)
```

## SML-szintaxis: deklarációk és kötések (folyt.)

- Értékkötés (*valbind:* value binding)

| | | | | |
|---|---|---|---|---|
| (28) | *valbind* ::= | *pat* = *exp* <`and` *valbind*> | értékkötés | value binding |
| (29) | | `rec` *valbind* | rekurzív kötés | recursive binding |

- Függvényérték-kötés (*fvalbind:* function value binding)

| | | | |
|---|---|---|---|
| (30) | *fvalbind* ::= | <`op`> *var* $atpat_{11}$ ... $atpat_{1n}$ <: *ty*> = $exp_1$ | $m, n \geq 1$ |
| | | \| <`op`> *var* $atpat_{21}$ ... $atpat_{2n}$ <: *ty*> = $exp_2$ | |
| | | \| ... | |
| | | \| <`op`> *var* $atpat_{m1}$ ... $atpat_{mn}$ <: *ty*> = $exp_m$ | |
| | | <`and` *fvalbind*> | |

*Megjegyzés:* Ha *var* infix, akkor egy *fvalbind* definícióban vagy infix helyzetben kell használni, vagy elé kell írni az `op` direktívát; azaz a definícióban a bal oldalon (*atpat var atpat'*) vagy `op` *var* (*atpat, atpat'*) írható. A zárójelek elhagyhatók, ha *atpat'* után közvetlenül :*ty* vagy = áll.

- Példák:

```
val even = fn 0 => true | x => not(odd(x-1))
and odd = fn 0 => false | y => not(even(y-1));   vö. (28)
fun (f o g) x = g(f x);                          vö. (30)
```





## SML-szintaxis: típuskifejezések

- Típus (*ty:* type)

| | | | | |
|---|---|---|---|---|
| (31) | *ty* ::= | *tyvar* | típusváltozó | type variable |
| (32) | | *tycon* | típuskonstruktor | type constructor |
| (33) | | { <*tyrow*> } | rekordtípus-kifejezés | record type expression |
| (34) | | $ty_1$ * $ty_2$ | pár-típus | pair type |
| (35) | | $ty_1$ -> $ty_2$ | függvénytípus-kifejezés | function type expression |
| (36) | | ( *ty* ) | típus zárójelben | parenthesized type |

- Típuskifejezés-sor (*tyrow:* type-expression row)

(37) *tyrow* ::= *lab* : *ty* <, *tyrow*>

- Példák:

| | |
|---|---|
| `'a, 'c, 'gamma` | vö. (31) |
| `int, real, word, word8, char, bool, string, order` | vö. (32) |
| `int * int -> int, unit -> unit` | vö. (34), (35) |
| `('a -> 'b) -> ('a list -> 'b list)` | vö. (35), (36) |
| `{num : int, den : int}, num : int, den : int` | vö. (33), (37) |





## SML-szintaxis: minták

- Atomi minta (*atpat:* atomic pattern)

| | | | | |
|---|---|---|---|---|
| (38) | *atpat* ::= | `_` | mindenesjel | wildcard |
| (39) | | *scon* | különleges állandó | special constant |
| (40) | | <`op`> *longvid* | értéknév | value identifier |
| (41) | | { <*patrow*> } | rekord | record |
| (42) | | ($pat_1$ * $pat_2$) | pár | pair |
| (43) | | (), {} | nullas | 0-tuple |
| (44) | | [$pat_1$, ..., $pat_n$] | lista, $n \geq 0$ | list, $n \geq 0$ |
| (45) | | ( *pat* ) | minta zárójelben | parenthesized pattern |

- Példák:

```
fun le GREATER = false | le EQUAL = true | le LESS = true;      vö. (40)
fun le GREATER = false | le _ = true;                          vö. (38), (40)
fun neg Bool.false = true | neg (true) = Bool.false;           vö. (40), (45)
fun prod [a, b] = a*b | prod [a, b, c] = a*b*c
  | prod [a] = a | prod () = 1;                                vö. (43), (44)
```





## SML-szintaxis: minták (folyt.)

- Mintasor (*patrow:* pattern row)

| | | | | |
|---|---|---|---|---|
| (46) | *patrow* ::= | ... | mindenesjel | wildcard |
| (47) | | *lab* = *pat* <, *patrow*> | mintasor | pattern row |
| (48) | | *lab* <: *ty*> <, *patrow*> | mezőnév mint változó | label as variable |

- Példák:

```
fun // {den = 0, ...} = raise Domain
  | // {num = n, den = d} = (real n) / (real d);   vö. (46), (47)
fun // {den = 0, ...} = raise Domain
  | // {num, den} = (real num) / (real den);       vö. (46), (48)
```

## SML-szintaxis: minták (folyt.)

- Minta (*pat:* pattern)

| | | | | |
|---|---|---|---|---|
| (49) | *pat* ::= | *atpat* | atomi minta | atomic pattern |
| (50) | | <op> *longvid atpat* | értékkonstrukció | value construction |
| (51) | | $pat_1$ *vid* $pat_2$ | infix értékkonstrukció | infixed value constr. |
| (52) | | *pat* : *ty* | minta típusmegkötéssel | typed pattern |
| (53) | | <op> *var* <: *ty*> as *pat* | réteges minta | layered pattern |

- Példa:

```
fun sum [] = 0                                          vö. (50)
| sum [a :  real] = a                                   vö. (52)
| sum (x ::  z ::  (yxs as y::xs)) = x + z + sum yxs  vö. (51), (53)
| sum (x ::  y ::  xs) = x + y + sum xs                 vö. (51)
| sum (op::(x, xs)) = x + sum xs                        vö. (50)
```



# RACIONÁLIS SZÁMOK



## SML-szintaxis: szintaktikai korlátozások

- Nem illeszthető minta kétszer ugyanarra a névre *(vid)*. Nem illeszthető kifejezéssor, mintasor vagy típuskifejezés-sor kétszer ugyanarra a mezőnévre *(lab)*.
- Ugyanaz a név nem köthető le kétféleképpen egy *valbind, typbind, datbind* vagy *exbind* deklarációban. A *datbind* deklarációban ugyanez érvényes az adatkonstruktorokra is.
- Ugyanaz a típusváltozó *(tyvar)* nem szerepelhet kétszer egy *tyvarseq* sorozatban valamely *typbind* vagy *datbind* deklaráció bal oldali *tyvarseq tycon* részében. Minden olyan típusváltozónak *(tyvar)*, amelyik előfordul a jobb oldalon, szerepelnie kell *tyvarseq*-ben.
- A `rec`-et követő minden *pat* = *exp* értékkötésben az *exp*-nek, szükség esetén zárójelben, `fn` *match* alakúnak kell lennie, ahol egy vagy több névhez típusmegkötés is társítható.
- `true`, `false`, `nil`, `::` és `ref` nem kaphat értéket *valbind, datbind* vagy *exbind*, `it` pedig *datbind* vagy *exbind* deklarációban.





## Példa: racionális számok

- A racionális számokat *rekordként* ábrázoljuk; az új (gyenge) típus neve `rat`.

```
type rat = {num : int, den : int};
```

- Nevet adunk néhány állandónak.

```
val ratZero = {num = 0, den = 1}; val ratOne   = {num = 1, den = 1};
val ratHalf = {num = 1, den = 2}; val ratThird = {num = 1, den = 3};
```

- A `rat` típusú számokat *normalizált* alakban tároljuk, különben pl. $\frac{1}{2}$ és $\frac{2}{4}$ nem lenne egyenlő. A normalizáláshoz szükségünk van a számláló és a nevező legnagyobb közös osztójára (`gcd`). A közös osztó egyik fontos tulajdonsága, hogy $d|n$ és $d|m \Rightarrow d|n \bmod m$.

```
(* gcd : int -> int -> int
   gcd n m = n és m legnagyobb közös osztója
*)
fun gcd n 0 = abs n
  | gcd n m = gcd (abs m) (abs(n mod m));
```

## Példa: racionális számok (folyt.)

- `gcd` ún. *részlegesen alkalmazható* függvény. Ha összes argumentumánál kevesebbre alkalmazzuk, *függvényértéket* ad eredményül.
- Sajnos, a `normalize` függvényben `n` és `m` legnagyobb közös osztóját kétszer is kiszámoljuk: később látni fogjuk, hogyan javíthatunk a hatékonyságán.

```
(* normalize : rat -> rat
   normalize r = r normalizált alakban
*)
fun normalize {num = n, den = 0} = raise Domain
  | normalize {num = n, den = d} = {num = n div (gcd n d), den = d div (gcd n d)}
```

- Két egészből *konstruktorfüggvénnyel* (`toRat`) érdemes létrehozni a racionális számot, különben a normalizált tárolás követelménye sérülhet.

```
(* toRat : int -> int -> rat
   toRat n d = n nevezőjű és d számlálójú racionális szám, normalizált alakban
*)
fun toRat n d = normalize {num = n, den = d};
```



# PÁR ÉS TÍPUSA



## Kitérő: pár és típusa

### Mi a +-szal jelölt összeadás-művelet típusa SML-ben?

- A `+` kétoperandusú művelet, argumentuma egy *pár,* pl. `3 + 4`.
- `+ : int * int -> int` vagy `+ : real * real -> real`, ahol `*` egy újabb típusművelet, a *keresztszorzat* (*Descartes-szorzat*) jele.
- A `+` műveleti jel (függvényjel) *többszörös terhelésű.*
- `+` prefix helyzetben is használható, ha eléírjuk az `op` kulcsszót, pl. `op+(3, 4)`. Ilyenkor az operandusait *párként, zárójelbe zárva* kell megadni.

### A beépített infix típusoperátorok precedenciája és kötése

- Két beépített infix típusoperátor van az SML-ben: `->` (leképzés) és `*` (keresztszorzat). A `*` precedenciája a nagyobb. A `*` balra, a `->` jobbra köt.
- Példák: `'a * 'b * 'c = ('a * 'b) * 'c`
  `'a -> 'b -> 'c = 'a -> ('b - 'c)`
  `'a * 'b -> 'c = ('a * 'b) -> 'c`



# RACIONÁLIS SZÁMOK

## Példa: racionális számok – a négy alapművelet

```
(* **, //, ++, -- : rat * rat -> rat
   r1 ** r2 = az r1 és r2 racionális számok szorzata
   r1 // r2 = az r1 és r2 racionális számok hányadosa
   r1 ++ r2 = az r1 és r2 racionális számok összege
   r1 -- r2 = az r1 és r2 racionális számok különbsége
*)
infix 7 ** //; infix 6 ++ --;

fun (r1 : rat) ** (r2 : rat) = toRat (#num r1 * #num r2) (#den r1 * #den r2);

fun (r1 : rat) // (r2 : rat) = toRat (#num r1 * #den r2) (#num r2 * #den r1);

fun {num= n1, den= d1} ++ {num= n2, den= d2} = toRat (n1*d2 + n2*d1) (d1*d2);

fun {num= n1, den= d1} -- {num= n2, den= d2} = toRat (n1*d2 - n2*d1) (d1*d2);
```





## Példa: racionális számok – relációs műveletek

- Az = és a <> relációt *készen kapjuk:* két összetett érték strukturálisan összehasonlítható, ha az elemeiken az egyenlőségvizsgálat elvégezhető.

```
(* <<, >>, <<=, >>= : rat * rat -> bool
   r1 << r2 = igaz, ha r1 kisebb r2-nél
   r1 >> r2 = igaz, ha r1 nagyobb r2-nél
   r1 <<= r2 = igaz, ha r1 nem nagyobb r2-nél
   r1 >>= r2 = igaz, ha r2 nem nagyobb r1-nél
*)
infix 4 << >> <<= >>=;

fun (r1 : rat) << (r2 : rat) = #num r1 * #den r2 < #num r2 * #den r1;

fun (r1 : rat) >> (r2 : rat) = #num r1 * #den r2 > #num r2 * #den r1;

fun r1 <<= r2 = not(r1 >> r2);    fun r1 >>= r2 = not(r1 << r2);
```



# POLIMORFIZMUS



## Polimorfizmus

- Nézzük az identitásfüggvényt: `fun id x = x`.
- Mi az `x` típusa? Bármilyen típusú lehet: típusát *típusváltozó* jelöli.

  `> val 'a id = fn : 'a -> 'a`
- `id` *polimorf* függvényt jelöl, `x` és `id` *politípusú* nevek.
- A *percjellel* kezdődő típusnév (pl. `'a`, olvasd *alfa*): *típusváltozó*.

Polimorfizmus többféle változatban fordul elő a programozásban.

- Egy *polimorf név* egyetlen olyan algoritmust azonosít, amely tetszőleges típusú argumentumra alkalmazható; ez a *paraméteres polimorfizmus*.
- Egy *többszörösen terhelt név* több különböző algoritmust azonosít: ahány típusú argumentumra alkalmazható, annyifélét; ez az *ad-hoc* vagy *többszörös terheléses* polimorfizmus.
- A polimorfizmus harmadik változata az *öröklődéses polimorfizmus* (vö. objektum-orientált programozás).

# KÉT FÜGGVÉNY KOMPOZÍCIÓJA



## Kitérő: két függvény kompozíciója

- Az $f \circ g$ függvénykompozíció az SML-ben

```
(* f o g = az f és g függvények kompozíciója
*)
```

`infix 2 o; fun (f o g) = fn x => f(g x);` **vagy** `fun (f o g) x = f(g x);`

- Az `o` típusa `? * ? -> ?` szerkezetű. Mit írjunk a `?`-ek helyébe? Vezessük le!
  - A függvénydefiníció jobb oldalán álló kifejezés elemzésével kezdjük.

    `x : 'a` `g : 'a -> 'b` `f : 'b -> 'c`
  - A függvénydefinícióban az egyenlőségjel (`=`) bal és jobb oldalán álló kifejezéseknek azonos értéket kell eredményül adniuk, ezért `f o g` és `f` eredményének azonos a típusa (azaz `'c`).

    `(f o g) : 'a -> 'c` `o : ('b -> 'c) * ('a -> 'b) -> ('a -> 'c)`
- Példa: `round : real -> int`, `chr : int -> char`
  `chr o round : real -> char`



# RACIONÁLIS SZÁMOK



## Példa: racionális számok (folyt.)

- A racionális számokon értelmezett `<<=` és `>>=` másképpen:

```
val op<<= = not o op>>;    val op>>= = not o op<<;
```

- Egy racionális számot füzérré alakítás után írunk ki a képernyőre.

```
(* toString : rat -> string
   toString r = az r racionális szám füzérként (számláló/nevező alakban,
                ha a nevező = 1, egyébként egészként)
*)
fun toString {num, den = 1} = Int.toString num
  | toString {num, den}     = Int.toString num ^ "/" ^ Int.toString den
```

- Példák `rat` típusú értékek használatára

```
normalize (toRat 15 3);       toString(toRat 2 3 ** toRat 5 4);
normalize (toRat 15 ~3);      toString(toRat 2 3 // toRat 5 3);
normalize (toRat ~15 3);      toString(toRat 1 4 ++ toRat 3 10);
normalize (toRat ~15 ~3);     toString(toRat 3 10 -- toRat 1 4);
```

## Példa: racionális számok (folyt.)

- Példák `rat` típusú értékek használatára (folyt.)

```
toRat 2 3 << toRat 5 4;          toRat 2 3  >> toRat 5 3;
toRat 1 4 << toRat 3 10;         toRat 3 10 >> toRat 1 4;

infix 8 /-/;
fun n /-/ d = toRat n d;

toString(2/-/3 **  5/-/4);         2/-/3  <<   5/-/4;     1/-/4  <<   3/-/10;
toString(2/-/3 //  5/-/3);         2/-/3  <<   2/-/3;     3/-/10 >>   1/-/4;
toString(1/-/4 ++  3/-/10);        2/-/3  <<=  2/-/3;
toString(3/-/10 -- 1/-/4);         2/-/3  >>   5/-/3;     3/-/10 >>=  3/-/10;
```

- Példák `gcd` részleges alkalmazására

```
(* gcd120 : int -> int                                    gcd120 45;
  gcd m = m legnagyobb közös osztója 120-szal             gcd120 48;
*)                                                        gcd120 ~96;
val gcd120 = gcd 120;                                     gcd120 630;
```



# POLIMORFIZMUS



## Paraméteres polimorfizmus

- Az identitásfüggvény és típusa: `fun id x = x;  id : 'a -> 'a.`
  Az `mosml` válasza: `val 'a id = fn : 'a -> 'a`. Az `id` ***politípusú*** név.
- Az `=` és a `<>` műveletet ***készen kapjuk*** a legtöbb típusra (vö. `rat`).
  A típusuk: `=, <> : ''a * ''a -> bool`. A `''` ***egyenlőségi típust*** jelöl, az ilyen típusú értékeken az egyenlőségvizsgálat elvégezhető.
- Az egyenlőségvizsgálat ***korlátozottan*** polimorf: nem minden értékre végezhető el. Pl. egy $f$ és egy $g$ függvény akkor és csak akkor egyenlő, ha $\forall x \cdot fx = gx$. Ezt ***általánosságban*** lehetetlen eldönteni.
- Mi a `<`, `>`, `<=`, `>=` típusa?
  Pl. az `op<=`-re az `mosml` válasza: `val it = fn : int * int -> bool`.
  E négy művelet *ad-hoc* módon polimorf, a nevek ***többszörösen terhelhetők***, alapértelmezés szerint `int` típusú értékekre alkalmazhatók.
- Az `=` részlegesen alkalmazható változata legyen: `fun eq x y = x = y`.
  Típusa: `eq : ''a -> ''a -> bool`.





## Példák `eq` használatára (`''a eq : ''a -> ''a -> bool`)

| A kifejezés | Az `mosml` válasza |
|---|---|
| `eq 3 3;` | `> val it = true : bool` |
| `eq "id" "idn";` | `> val it = false : bool` |
| `eq id id;` | `! Toplevel input:` |
| | `! eq id id;` |
| | `!    ^^` |
| | `! Type clash: expression of type` |
| | `!   'e -> 'e` |
| | `! cannot have equality type ''f` |
| `eq 3;` | `> val it = fn : int -> bool` |
| `eq "id";` | `> val it = fn : string -> bool` |
| `val eqStr_id = eq "id";` | `> val eqStr_id = fn : string -> bool` |

- Az `id` függvény, típusa (`'e -> 'e`) nem egyenlőségi típus!
- Az `eq "id"` függvényértéket ad eredményül, ezért az `eqStr_id` függvényt jelöl. Olyan függvényt, amely az `"id"` füzérre alkalmazva `true`, minden más esetben `false` értéket ad eredményül.

## Példák `id` használatára (`'a id : 'a -> 'a`)

| A kifejezés | Az `mosml` válasza |
|---|---|
| `id 3;` | `> val it = 3 : int` |
| `id "id";` | `> val it = "id" : string` |
| `id round;` | `> val it = fn : real -> int` |
| `id id;` | `! Warning: Value polymorphism:` |
| | `! Free type variable(s) at top level in value identifier it` |
| | `> val it = fn : 'b -> 'b` |
| `id id 6.9;` | `> val it = 6.9 : real` |
| `fn x => id id x;` | `> val 'b it = fn : 'b -> 'b` |

- Az SML ún. *érték-polimorfizmust* használ.
  - Az SML a típusváltozókat, ahol csak tudja, általánosítja (pl. `fn x => id id x`).
  - Az `mosml` a nem általánosítható típusváltozókat meghagyja *szabad típusváltozónak* (pl. `id id`).





## Érték-polimorfizmus

- Tekintsük a `val x = e` deklarációt.
  - Az SML az `x` típusában előforduló szabad típusváltozókat akkor általánosítja, ha `e` ún. *nem-expanzív* kifejezés.
  - Ez csupán *szintaktikai* követelmény: egy kifejezés *nem-expanzív*, ha megfelel a *nexp* szintaktikai kategóriát leíró nyelvtani szabályoknak.





## Nem-expanzív kifejezés (egyszerűsítve)

- Nem-expanzív kifejezés (*nexp:* non-expansive expression)

| | | | |
|---|---|---|---|
| *nexp* ::= | *scon* | különleges állandó | special constant |
| | *longvid* | (esetleg minősített) értéknév | (possibly qualified) value identifier |
| | { <*nexprow*> } | nem-expanzív elemekből álló rekord | record of non-expansive expressions |
| | ( *nexp* ) | nem-expanzív kifejezés zárójelben | parenthesized non-expansive expression |
| | *nexp* : *ty* | nem-expanzív kifejezés típusmegkötéssel | typed non-expansive expression |
| | `fn` *match* | függvénykifejezés | function expression |

- Nem-expanzív kifejezéssor (*nexprow:* non-expansive expression row)

*nexprow* ::= *lab* = *nexp* <, *nexprow*>





## Példák nem-expanzív és expanzív kifejezésekre

- Egy nem-expanzív kifejezés egyszerűen: érték (azaz tovább nem egyszerűsíthető, ún. *kanonikus* kifejezés).

```
  val x = length;
> val 'a x = fn : 'a list -> int
```

`length` egy név, ezért nem-expanzív. Az `x` típusát leíró `'a list -> int` típuskifejezésben az `'a` szabad típusváltozó általánosítható, ezt tükrözi a definíció bal oldalán az `'a x`.

- Az `(fn f => f) length` kifejezés értéke is `length`, de expanzív, mert nem vezethető le a fenti nyelvtani szabályok alapján.

```
  val x = (fn f => f) length;
! Warning: Value polymorphism:
! Free type variable(s) at top level in value identifier x
> val x = fn : 'a list -> int
```

## Példák nem-expanzív és expanzív kifejezésekre (folyt.)

Az 'a típusváltozót az SML nem általánosítja. Az mosml meghagyja szabad típusváltozónak, és majd csak az x *első alkalmazásakor* köti le.

```
  x ["abc", "def"];
! Warning: the free type variable 'a has been instantiated to string
> val it = 2 : int
  x;
> val it = fn : string list -> int
```

- Ha már az 'a-t lekötöttük, más típushoz nem köthető; x nem politípusú név.

```
  x [123, 456, 789];
! Toplevel input:
!   x [123, 456, 789];
!     ^^^
! Type clash: expression of type
!   int
! cannot have type
!   string
```





## $\eta$-expanzió

- A típusváltozó általánosítása mindig kikényszeríthető a deklaráció jobb oldalának *$\eta$-expanziójával.*

  Az $\eta$-expanzió az e kifejezést a nem-expanzív fn y => e y kifejezéssel helyettesíti.

```
  val x1 = fn y => ((fn f => f) length) y;
> val 'b x1 = fn : 'b list -> int
```

  A fenti deklarációban a külső zárójelpár el is hagyható:

```
  val x1 = fn y => (fn f => f) length y;
```

- Az x1 politípusú név.

```
  x1 ["abc", "def"];
> val it = 2 : int
  x1 [123, 456, 789];
> val it = 3 : int
```



# LISTÁK



## Lista: definíciók, konstruktorok

- Definíciók
  1. A *lista* azonos típusú elemek véges (de nem korlátos!) sorozata.
  2. A lista olyan *rekurzív* lineáris adatszerkezet, amely azonos típusú elemekből áll, és
     - vagy üres,
     - vagy egy elemből és az elemet követő listából áll.
- Konstruktorok
  - Az üres lista jele a nil *konstruktorállandó*. nil típusa 'a list.
  - A :: *konstruktoroperátor* új listát hoz létre egy elemből és egy (esetleg üres) listából (infix, 5-ös precedenciájú, jobbra köt, típusa 'a * 'a list -> 'a list).
  - A nil helyett általában a [] jelet használjuk (szintaktikai édesítőszer).
  - A ::-ot négyespontnak vagy *cons*-nak olvassuk (vö. *constructor*, ami a függvény hagyományos neve a $\lambda$-kalkulusban és egyes funkcionális nyelvekben).

## Lista: jelölések, minták

- Példák
  - Lista létrehozása konstruktorokkal

```
[]          nil          #"" :: nil
3 :: 5 :: 9 :: nil     = 3 :: (5 :: (9 :: nil))
```

  - Szintaktikus édesítőszer lista jelölésére

```
[3, 5, 9]              =  3 :: 5 :: 9 :: nil
```

  - Vigyázat! A Prolog listajelölése hasonló, de vannak lényeges különbségek:

| SML | Prolog | | SML | Prolog | |
|---|---|---|---|---|---|
| `[]` | `[]` | azonos | `(x::xs)` | `[X\|Xs]` | különböző |
| `[1,2,3]` | `[1,2,3]` | azonos | `(x::y::z::zs)` | `[X,Y,Z\|Zs]` | különböző |

- Minták

  A `[]` és a `nil` állandók, a `::` operátor, valamint a `[x1, x2, ..., xn]` listajelölés mintában is alkalmazhatók.





## Lista: fej (`hd`), farok (`tl`)

- A nem-üres lista első eleme a lista *feje*.

```
(* hd : 'a list -> 'a *)
fun hd (x :: _) = x;
```

- A nem-üres lista első utáni elemeiből áll a lista *farka*.

```
(* tl : 'a list -> 'a list *)
fun tl (_ :: xs) = xs;
```

- `hd` és `tl` *parciális* függvények. Ha könyvtárbeli megfelelőiket (`List.hd`, `List.tl`) üres listára alkalmazzuk, `Empty` néven *kivételt* jeleznek.

  Fontos: a parciális függvények nem tévesztendők össze a parciálisan (azaz részlegesen) alkalmazható függvényekkel!





## Lista: hossz (`length`), elemek összege (`isum`), szorzata (`rprod`)

- Egy lista hosszát adja eredményül a már látott `length` függvény (l. `List.length`).

```
(* length : 'a list -> int *)
fun length (_ :: xs) = 1 + length xs
  | length []        = 0;
```

- Egy egész számokból álló lista elemeinek összegét adja eredményül `isum`.

```
(* isum : int list -> int *)
fun isum (x :: xs) = x + isum xs
  | isum []        = 0;
```

- Egy valós számokból álló lista elemeinek szorzatát adja eredményül `rprod`.

```
(* rprod : real list -> real *)
fun rprod (x :: xs) = x * rprod xs
  | rprod []        = 1.0;
```





## Példák: `hd`, `tl`, `length`, `isum`, `rprod`

- `hd`, `tl`

| A kifejezés | Az `mosml` válasza |
|---|---|
| `List.hd [1, 2, 3];` | `> val it = 1 : int` |
| `List.hd [];` | `! Uncaught exception:`<br>`! Empty` |
| `List.tl [1, 2, 3];` | `> val it = [2, 3] : int list` |
| `List.tl [];` | `! Uncaught exception:`<br>`! Empty` |

- `length`, `isum`, `rprod`

| A kifejezés | Az `mosml` válasza |
|---|---|
| `length [1, 2, 3, 4];` | `> val it = 4 : int` |
| `length [];` | `> val it = 0 : int` |
| `isum [1, 2, 3, 4];` | `> val it = 10 : int` |
| `isum [];` | `> val it = 0 : int` |
| `rprod [1.0, 2.0, 3.0, 4.0];` | `> val it = 24.0 : real` |
| `rprod [];` | `> val it = 1.0 : real` |

## Lista: adott transzformáció alkalmazása minden elemre (map)

- Példa: vonjunk négyzetgyököt egy valós számokból álló lista minden eleméből!

  `map Math.sqrt [1.0, 4.0, 9.0, 16.0] = [1.0, 2.0, 3.0, 4.0]`
- Általában: `map f [x1, x2, ..., xn] = [f x1, f x2, ..., f xn]`
- A függvény típusa: `map : ('a -> 'b) -> 'a list -> 'b list`
- Egy-egy klózt írunk a triviális és a nem-triviális eset lefedésére
  - `map f [] = []`
  - `map f (x :: xs) = f x :: map f xs`

  ```
  fun map f (x :: xs) = f x :: map f xs | map f [] = [];
  ```
- `map` típusa, ha egyargumentumú függvénynek tekintjük (ui. -> jobbra köt):

  `map : ('a -> 'b) -> ('a list -> 'b list)`.

  Azaz ha `map`-et egy `'a -> 'b` típusú függvényre alkalmazzuk, akkor olyan függvényt ad eredményül, amelyet egy `'a list` típusú listára alkalmazva egy `'b list` típusú listát kapunk.



# PROGRAMHELYESSÉG



## A program helyességének igazolása a map példáján

- A rekurzív programról be kell látnunk, hogy
  - funkcionálisan helyes (azt kapjuk eredményül, amit várunk),
  - a kiértékelése biztosan befejeződik (nem esik „végtelen ciklusba”).
- Bizonyítása hossz szerinti *strukturális indukcióval* (amely visszavezethető a teljes indukcióra) lehetséges.

  ```
  fun map f (x :: xs) = f x :: map f xs | map f [] = [];
  ```
  - Feltesszük, hogy a `map` jó eredményt ad az eggyel rövidebb listára (azaz a lista farkára). Alkalmazzuk az `f`-et a lista első elemére (a fejére). A fej transzformálásával kapott eredményt a farok transzformálásával kapott lista elé fűzve valóban a várt eredményt kapjuk.
  - A kiértekelés véges számú lépésben befejeződik, mert a lista véges, a `map` függvényt a *rekurzív ágban* minden lépésben egyre rövidülő listára alkalmazzuk, és gondoskodtunk a rekurzió leállításáról (a *triviális eset* kezeléséről, ui. van nem rekurzív ág).



# LISTÁK

## Lista: adott predikátumot kielégítő elemek kiválogatása (`filter`)

- Kitérő: `explode`, `implode`
  - `explode : string -> char list`, pl. `explode "abc" = [#"a", #"b", #"c"]`
  - `implode : char list -> string`, pl. `implode [#"a", #"b", #"c"] = "abc"`
- Példa: gyűjtsük ki a kisbetűket egy karakterlistából!

```
List.filter Char.isLower (explode "VaLtOgAtVa") =
                                  [#"a", #"t", #"g", #"t", #"a"]
```

- Általában: ha `p` $x_1$ `= true`, `p` $x_2$ `= false`, `p` $x_3$ `= true`, ..., `p` $x_n$ `= true`, akkor `filter p [`$x_1$`, `$x_2$`, `$x_3$`, ..., `$x_n$`] = [`$x_1$`, `$x_3$`, ..., `$x_n$`]`.
- A függvény típusa: `filter : ('a -> bool) -> 'a list -> 'a list`
- Egy-egy klózt írunk a triviális és a nem-triviális eset lefedésére
  - `filter p [] = []`
  - `filter p (x :: xs) = if p x then x :: filter p xs else filter p xs`





## Lista: `filter` (folyt.)

- Ezzel `filter` definíciója

```
fun filter p (x :: xs) =
      if p x then x :: filter p xs else filter p xs
  | filter _ [] = [];
```

- `filter` típusa, ha egyargumentumú függvénynek tekintjük (-> jobbra köt!):
  `filter : ('a -> bool) -> ('a list -> 'a list)`.
  Azaz ha `filter`-t egy `'a -> bool` típusú függvényre (predikátumra) alkalmazzuk, akkor olyan függvényt ad eredményül, amelyet egy `'a list` típusú listára alkalmazva egy `'a list` típusú listát kapunk.





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

- Vissza-visszatérő feladat egy lista redukciója kétoperandusú művelettel. Közös, hogy $n$ db értékből egyetlen értéket kell előállítani (vö. *redukció*).
- `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 [`$x_1$`, `$x_2$`, ..., `$x_n$`] = (`$x_1$ ⊕ ($x_2$ ⊕ ... ⊕ ($x_n$ ⊕ e) ...))`
  `foldr op⊕ e [] = e`
  `foldl op⊕ e [`$x_1$`, `$x_2$`, ..., `$x_n$`] = (`$x_n$ ⊕ ... ⊕ ($x_2$ ⊕ ($x_1$ ⊕ e)) ...)`
  `foldl op⊕ e [] = e`
- Asszociatív műveleteknél `foldr` és `foldl` eredménye azonos.





## Példák `foldr` és `foldl` alkalmazására

- A $\oplus$ művelet `e` operandusa néhány gyakori műveletben – összeadás, szorzás, konjunkció (logikai „és”), alternáció (logikai „vagy”) – a (jobb oldali) *egységelem* szerepét tölti be.
- `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` vagy `foldr` felhasználásával. Kétoperandusú műveletként olyan segédfüggvényt (`inc`) alkalmazunk, amelyik *nem használja* az első paraméterét.

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

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

## Példák `foldr` és `foldl` alkalmazására (folyt.)

- 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, 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`)





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

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

  `foldr op⊕ 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;
```

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

  `foldl op⊕ 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;
```





## Lista redukciója bal oldali egységelemű függvénnyel (`foldL`)

- A kivonás művelete balra köt: $x_1 - x_2 - x_3 - x_4 = ((x_1 - x_2) - x_3) - x_4$.
- Nem feleltethető meg sem `foldr`-nek, sem `foldl`-nek.

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

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

- Nevezzük `foldL`-nek a listában *balról jobbra* haladó, alábbi specifikációjú függvényt. Vegyük észre, hogy ⊕ bal oldali egységelemet vár.

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

- `foldL` olyan kétargumentumú függvényt vár, amelynek az „egységelem” (valójában: a részeredmény) az *első* argumentuma: `f : 'a * 'b -> 'a`.

```
(* foldL : ('a * 'b -> 'a) -> 'a -> 'b list -> 'a
   foldL f e xs = az xs elemeire balról jobbra haladva alkalmazott,
                  kétoperandusú, e egységelemű f művelet eredménye *)
fun foldL f e (x::xs) = foldL f (f(e, x)) xs
  | foldL f e [] = e;
```





## Példák listaelemek különbségének és hányadosának képzésére

- Az `e` argumentum aktuális értéke a sorozat *első* eleme – a *kisebbítendő,* ill. az *osztandó*.

```
foldL op- 20 [] = 20;
foldL op- 20 [5, 6, 7] =
        (((20 - 5) - 6) - 7);

foldL (op div) 180  [] = 180;
foldL (op div) 180  [2, 3, 5] =
           (((180 div 2) div 3) div 5);
```

- Ha többször használjuk e műveleteket, érdemes nekik nevet adni. A kisebbítendő, ill. az osztandó speciális kezelését elrejtjük.

```
fun subtract ns = foldL op- (hd ns) (tl ns);
subtract [20, 5, 6, 7] = (((20 - 5) - 6) - 7);

fun divide ns = foldL op div (hd ns) (tl ns);
divide [180, 2, 3, 5] = (((180 div 2) div 3) div 5);
```

## Listaelemek különbsége és hányadosa foldl-lel és foldr-rel

- Igazság szerint foldL felesleges: a feladat jól megoldható foldl-lel vagy foldr-rel is.

```
fun subtract1 ns = hd ns - foldl op+ 0 (tl ns);
subtract1 [20, 5, 6, 7] = (((20 - 5) - 6) - 7);

fun divide1 ns = hd ns div foldl op* 1 (tl ns);
divide1 [180, 2, 3, 5] = (((180 div 2) div 3) div 5);
```

- foldr és foldl típusa, ha egyparaméteres függvénynek tekintjük őket (a -> jobbra köt!):

  `foldr, foldl : ('a * 'b -> 'b) -> ( 'b -> 'a list -> 'b)`

  Azaz ha foldr-t vagy foldl-t egy 'a -> * 'b -> 'b típusú függvényre alkalmazzuk, akkor olyan függvényt ad eredményül, amelyet egy 'b típusú egységelemre és egy 'a list típusú listára alkalmazva 'b típusú (redukált) értéket kapunk.



# KIFEJEZÉSEK KIÉRTÉKELÉSE



## Mohó kiértékelés: faktoriális kiszámítása naív rekurzióval

- A faktoriális matematikai definíciója és megvalósítása SML-ben

  fac $0 = 1$ fac $n = n *$ fac $(n-1)$

```
(* fac : int -> int       (-- fontos a klózok sorrendje! --)
   fac n = n!
   PRE n >= 0 *)
fun fac 0 = 1 | fac n = n * fac(n-1);
```

- fac mohó kiértékelése $n = 4$ esetén (egyes triviális lépéseket elhagyunk).

  `fac 4` $\rightarrow$ `4 * fac (4-1)` $\rightarrow$ `4 * fac 3` $\rightarrow$ `4 * (3 * fac (3-1))` $\rightarrow$
  $\rightarrow$ `4 * (3 * fac (2))` $\rightarrow \cdots \rightarrow$ `4 * (3 * (2 * (1 * 1)))` $\rightarrow \cdots \rightarrow$ `24`

- A rekurzív kiértékelés követi a matematikai definíciót.
- Rontja a hatékonyságot, hogy a rekurzív végrehajtás során minden részeredményt a veremben tárolni kell.
- Ha a szorzás asszociativitását kihasználjuk, nem kell tárolni az összes tényezőt, csak az aktuális részeredményt.





## Faktoriális kiszámítása jobbrekurzióval

- Először egy *akkumulátort* (gyűjtőargumentumot) használó *segédfüggvényt* definiálunk. Vegyük észre, hogy a rekurzív hívás *jobbrekurzív:* eredménye közvetlenül, további műveletek elvégzése nélkül adja a végeredményt.

```
(* faci : int -> int -> int    (-- fontos a klózok sorrendje! --)
   faci n p = p * n!           (-- p az akkumulátor --)
*)
fun faci 0 p = p
  | faci n p = faci (n-1) (n*p);
```

- faci-t felhasználjuk az egyparaméteres fac függvény definiálására. Az akkumulátornak alkalmas *kezdőértéket* adunk.

```
(* fac : int -> int
   fac n = n!
   PRE n >= 0
*)
fun fac n = faci n 1;
```

## Faktoriális kiszámítása jobbrekurzióval (folyt.)

- `fac` nem rekurzív, ezért csak `faci` kiértékelését vizsgáljuk (egyes triviális lépéseket összevonunk).

  A függvény: `fun faci 0 p = p | faci n p = faci (n-1) (n*p)`

  `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` *akkumulátor* gyűjti a részeredményt, ezért `faci` tárigénye állandó.
- A kiértékelés *iteratív*.
- A jó fordítóprogram felismeri a jobbrekurziót, és hatékony tárgykódot állít elő: az argumentumokat frissíthető lokális változókban tárolja, a rekurziót iterációval helyettesíti.
- A jobbrekurziót *terminális rekurziónak* is nevezik (angolul: *tail* vagy *terminal* recursion).
- `foldl` jobbrekurzív, `e` argumentuma akkumulátorként viselkedik.





## Lokális kifejezés

- *Lokális kifejezést* használunk, ha ismétlődő részkifejezéseket *csak egyszer* akarunk kiszámítani, vagy akkor, ha bizonyos értékeket a program többi része elől *el akarunk rejteni.*
- Szintaxisa: `let` *d* `in` *e* `end`, ahol
  - *d* nemüres deklarációsorozat,
  - *e* nemüres kifejezés.
- Példa:

```
fun fac n =
  let
      fun faci 0 p = p
        | faci n p = faci (n-1) (n*p)
  in
      faci n 1
  end
```





## Lokális deklaráció

- *Lokális deklarációt* használunk olyan értékek bevezetésére, amelyeket a program többi része elől *el akarunk rejteni.*
- Szintaxisa: `local` *d1* `in` *d2* `end`, ahol
  - *d1* és *d2* nemüres deklarációsorozatok.
- Példa:

```
local
    fun faci 0 p = p
      | faci n p = faci (n-1) (n*p)
in
    fun fac n = faci n 1
end
```



# LOGIKAI MŰVELETEK

## Logikai műveletek

- Típusnév: `bool`, adatkonstruktorok: `false`, `true`, beépített függvény: `not`.
- ***Lusta kiértékelésű*** beépített operátorok
  - Három argumentumú: `if b then e1 else e2`.
    Nem értékeli ki az `e2`-t, ha a `b` igaz, ill. az `e1`-et, ha a `b` hamis.
  - Két argumentumúak:
    `e1 andalso e2` : nem értékeli ki az `e2`-t, ha az `e1` hamis.
    `e1 orelse e2` : nem értékeli ki az `e2`-t, ha az `e1` igaz.
- Mind a három csupán szintaktikai édesítőszer!
  - `if b then e1 else e2` ≡ `(fn true => e1 | false => e2) b`
  - `e1 andalso e2` ≡ `(fn true => e2 | false => false) e1`
  - `e1 orelse e2` ≡ `(fn true => true | false => e2) e1`
  - `fun ifThenElse b = (fn true => e1 | false => e2) b; ifThenElse true;`
- Tipikus hiba: `if` *exp* `then true else false` !!!





## Logikai műveletek (folyt.)

- Nyilvánvaló: `andalso` és `orelse` kifejezhető `if-then-else`-szel is.
  - `if e1 then e2 else false` ≡ `e1 andalso e2`
  - `if e1 then true else e2` ≡ `e1 orelse e2`
- Használjuk az `andalso`-t és az `orelse`-t az `if-then-else` helyett, ahol csak lehet: olvashatóbb lesz a program.
- Lusta kiértékelésű függvényt a programozó nem definiálhat az SML-ben. Az SML, mielőtt egy függvényt alkalmazna az (egyszerű vagy összetett) argumentumára, kiértékeli.
- Az `andalso` és az `orelse` ***mohó kiértékelésű*** megfelelői:

```
(* && (a, b) = a /\ b
   && : bool * bool -> bool
*)
fun op&& (a, b) = a andalso b;
infix 2 &&;
```

```
(* || (a, b) = a \/ b
   || : bool * bool -> bool
*)
fun op|| (a, b) = a orelse b;
infix 1 ||;
```



# LISTÁK



## Listák összefűzése és megfordítása

- Listák összefűzése és megfordítása beépített függvényekkel: `@`, `rev` és `revAppend` (`List` könyvtár).
  - `@` a `fun append (xs, ys) = foldr op:: ys xs` beépített megfelelője: infix, 5-ös precedenciájú, jobbra köt, típusa `'a list * 'a list -> 'a list`.
  - `revAppend` a `fun revApp (xs, ys) = foldl op:: ys xs` beépített megfelelője: prefix, típusa `'a list * 'a list -> 'a list`.
  - `rev` a `fun rev xs = foldl op:: [] xs` beépített megfelelője: prefix, típusa `'a list -> 'a list` (vö. `revApp`).
- Az $[m, n)$ tartományba eső egészek listája: a kézenfekvő megoldás

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

## Listák összefűzése és megfordítása

- Az $[m, n)$ tartományba eső egészek listája: jobbrekurzív megoldás

```
fun upto m n =
      let (* az up számára az n állandó érték,
             ezért nem kell argumentumként átadni *)
          fun up zs m = if m < n then up (m::zs) (m+1) else rev zs
      in  up [] m
      end;
```

- Az $[m, n)$ tartományba eső egészek listája: hatékony jobbrekurzív megoldás

```
fun upto m n =
      let (* hátulról visszafelé haladva építjük föl a listát,
             ezért a végén nem kell megfordítani *)
          fun up zs n = if m < n  then up (n-1::zs) (n-1) else zs
      in up [] n
      end;
```





## Lista legnagyobb elemének megkeresése

- Egy egészlista legnagyobb elemének kiválasztásához szükségünk van az `Int.max` függvényre.
  - Üres listának nincs legnagyobb eleme,
  - egyelemű listában az egyetlen elem a legnagyobb,
  - legalább kételemű lista legnagyobb elemét úgy kapjuk, hogy az első elem és a maradéklista elemeinek legnagyobbika közül kiválasztjuk a legnagyobbat.

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

- `max` egy változata egészekre

```
fun max (n, m) = if n > m then n else m
```





## Lista legnagyobb elemének megkeresése (folyt.)

- Hogyan tehető polimorffá a `maxl` függvényt? Magasabbrendű, ún. `generikus` függvényként definiáljuk: *argumentumként* kapja azt a többszörösen terhelhető függvényt, amely két érték közül a nagyobbikat kiválasztja.

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

- `max` mindig ugyanaz, mégis újra és újra átadjuk argumentumként a rekurzív ágban. Javítja a hatékonyságot, ha *lokális kifejezést* használunk. (Lokális deklaráció használata most nem segítene. Miért nem?)

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





## Lista (folyt.)

- Változatok `max`-ra

```
(* charMax : char * char -> char *)
fun charMax (n, m) = if ord n > ord m then n else m;

(* pairMax : ((int * real) * (int * real)) -> (int * real)
fun pairMax (n as (n1 : int, n2 : real), m as (m1, m2)) =
      if n1 > m1 orelse n1 = m1 andalso n2 >= m2 then n else m;
```

- `concat xss` = az `xss`-beli listákat egy listába fűzi. Könyvtári változata: `List.concat`.

```
(*  concat : 'a list list -> 'a list *)
fun concat xss = foldr op@ [] xss;
```

- `ListPair.zip` két lista páronkénti elemeiből álló párok listáját, `ListPair.unzip` párok listájából két listát ad eredményül.

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

- Legyen $xs = [x_0, x_1, \ldots, x_{i-1}, x_i, x_{i+1}, \ldots, x_{n-1}]$, akkor
  take(xs, i) $= [x_0, x_1, \ldots, x_{i-1}]$ és drop(xs, i) $= [x_i, x_{i+1}, \ldots, x_{n-1}]$.

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

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

- Könyvtári változatuk, List.take és List.drop $i < 0$ vagy $i >$ length xs esetén Subscript kivételt jelez.





## Halmazműveletek

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

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

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

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

newMem, ha a sorrendtől eltekintünk, halmazt hoz létre.





## Halmazműveletek (folyt.)

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

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

- Szerencsésebb a halmazokat a megszokott halmazműveletekkel kezelni. 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, $pS$).





## Halmazműveletek (folyt.)

- Listaként kezeljük a halmazokat, később hatékonyabb ábrázolást választhatunk, pl. rendezett listát vagy bináris fát.
- Két halmaz uniója

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

- Két halmaz metszete

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

## Halmazműveletek (folyt.)

- Részhalmaza-e egy halmaz egy másiknak?

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

- Két halmaz egyenlősége

  A listák egyenlőségvizsgálata beépített művelet az SML-ben. Halmazokra mégsem használható, mert pl. `[3, 4]` és `[4, 3, 4]` listaként ugyan különböznek, de halmazként egyenlők.

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





## Halmazműveletek (folyt.)

- Halmaz hatványhalmaza

  A hatványhalmaz egy halmaz *összes* részhalmazának a halmaza, az eredeti halmazt és az üres halmazt is beleértve.

  Jelöljük $S$-sel az eredeti halmazt. $S$ hatványhalmazát úgy állíthatjuk elő, hogy $S$-ből kiveszünk egy $x$ elemet, és azután ***rekurzív módon*** előállítjuk az $S - \{x\}$ hatványhalmazát.

  Ha tetszőleges $T$ halmazra $T \subseteq S - \{x\}$, akkor $T \subseteq S$ és $T \cup \{x\} \subseteq S$, így mind $T$, mind $T \cup \{x\}$ eleme $S$ hatványhalmazának.

  A `pws` függvényben a `base` argumentum gyűjti a hatványhalmaz elemeit; kezdetben üresnek kell lennie.

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





## Halmazműveletek (folyt.)

- Halmaz hatványhalmaza (folyt.)

  A `pws(xs, base) @ pws(xs, x::base)` kifejezésben `pws(xs, base)` valósítja meg az $S - \{x\}$ rekurzív hívást (hiszen `x::xs` felel meg $S$-nek), azaz állítja elő az összes olyan halmazt, amelyekben `x` nincs benne.

  `pws(xs, x::base)` ugyancsak rekurzív módon `base`-ben gyűjti az `x` elemeket, vagyis előállítja az összes olyan halmazt, amelyben `x` benne van.

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



# KIFEJEZÉSEK KIÉRTÉKELÉSE

## Sztatikus és dinamikus kötés, mohó és lusta kiértékelés

- *Sztatikus kötés:* a formális paraméter összes előfordulását *fordítási időben* helyettesítjük az argumentummal (aktuális paraméterrel) a függvény (eljárás) törzsében.
- *Dinamikus kötés:* a formális paraméter összes előfordulását *futási időben* helyettesítjük az argumentummal (aktuális paraméterrel) a függvény (eljárás) törzsében.

  Kérdés, hogy az aktuális paraméterként átadott kifejezést az értelmező mikor értékeli ki: a behelyettesítés *előtt* vagy *után*.
- *Mohó kiértékelés:* a behelyettesítés *előtt* kiértékeljük az *összes* argumentumot (más megnevezések: *érték szerinti* paraméterátadás, *eager evaluation, call-by-value).*
- *Lusta kiértékelés:* a behelyettesítés *után* csak azt az argumentumot értékeljük ki, amelyikre szükség van, és csak akkor, amikor *szükség van* rá (más megnevezések: *szükség szerinti* paraméterátadás, *lazy evaluation, call-by-need.)*





## A mohó és a lusta kiértékelés összevetése

- Más paraméterátadási eljárások
  - *név szerinti* paraméterátadás *(call-by-name*, Algol).
  - *hivatkozás szerinti* paraméterátadás *(call-by-reference,* Pascal, C stb.)
- Nézzünk két egyszerű függvényt!

```
(* sq : int -> int          (* zero : int -> int
   sq x = x négyzete *)        zero x = az x-től függetlenül mindig 0 *)
fun sq x = x * x;           fun zero x = 0;
```

Az `sq` függvény argumentumát *lusta kiértékelés* esetén *kétszer* számítjuk ki.

A `zero` függvény argumentumát *mohó kiértékelés* esetén *feleslegesen* számítjuk ki, mert nem használjuk semmire.





## Mohó kiértékelés

- Emlékeztető: az $f\ e$ értékét úgy számítjuk ki, hogy először az $f$ függvényértéket adó kifejezés, majd az $e$ kifejezés értékét határozzuk meg, és ezután helyettesítjük az $f$ törzsében a formális paraméter minden előfordulását az $e$ értékével.

```
fun sq x = x * x;          fun zero x = 0;
```

- Nézzük `sq(sq(sq 2))` egyszerűsítését! (Az egyszerűsítés eredménye tovább már nem egyszerűsíthető, ún. *kanonikus* kifejezés.)

  `sq` három alkalmazásából csak a harmadiknak kanonikus kifejezés az argumentuma.

  `sq(sq(sq 2))` → `sq(sq(2*2))` → `sq(sq 4)` → `sq(4*4)` → `sq 16` → `16*16` → `256`

  Az utolsó lépés kivételével `zero(sq(sq(sq 2)))` egyszerűsítési lépései ugyanezek, pedig az eredmény nyilvánvalóan 0!

  Mohó kiértékelés mellett a számítógépet feleslegesen dolgoztatjuk!





## Név szerinti paraméterátadás

- Egy függvény alkalmazása előtt sokszor nemcsak fölösleges, hanem káros is előre kiszámítani az argumentumokat, mert végtelen rekurzió vagy illegális művelet (indexhatár-túllépés, 0-val való osztás stb.) lehet az „eredménye”.
- Az Algol *név szerinti* paraméterátadása a formális paraméter összes előfordulását az argumentumként átadott *teljes* (nem kanonikus) *kifejezéssel* helyettesíti a függvény törzsében.

  Ezért `zero(sq(sq(sq 2)))` *név szerinti* paraméterátadás esetén *azonnal,* az argumentum kiértékelése nélkül 0-t ad eredményül!

```
fun sq x = x * x;          fun zero x = 0;
```

A *név szerinti* paraméterátadás sem mindig kedvező: pl. `sq(sq(sq 2))` esetén `sq` mindegyik alkalmazása *megkétszerezi* az argumentumok számát. Aligha ezt akarjuk!

`sq(sq(sq 2))` → `sq(sq 2) * sq(sq 2)` → `(sq 2 * sq 2) * sq(sq 2)` → `((2*2) * sq 2) * sq(sq 2)` → ··· → `(4*(2*2) * sq(sq 2))` → ···

## Lusta kiértékelés

- ***Lusta kiértékelés*** esetén minden argumentumot csak egyszer kell kiértékelni: akkor, amikor ***először*** van rá szükség. Az argumentum összes elôfordulását egy ***rejtett hivatkozással*** helyettesítjük (mivel ***el van rejtve*** a programozó elöl, biztonságos): amikor a számítógép az argumentumot először kiértékeli, a kapott értéket elrakja, és később az összes olyan helyen, ahol szükség lesz rá, felhasználja.
- A függvényeket és argumentumaikat ***irányított gráffal*** ábrázolják: a gráf egy részének kiértékelésekor a gráfot az eredményül kapott értékkel frissítik a számítógépben (ezért nevezik ***gráfredukciónak***).

  A lusta kiértékeléshez bonyolult nyilvántartást kell vezetni (időigényes!).
- A lusta kiértékelés működési elvének megértéséhez irányított gráf helyett most $x = [E]$ -vel jelöljük, hogy az $x$ ***összes előfordulása*** osztozik az $E$ értéken.





## Lusta kiértékelés

- Nézzük pl. `sq(sq(sq 2))` lusta kiértékelését!

  ```
  fun sq x = x * x;          fun zero x = 0;
  ```

  ($x = [E]$ jelentése: az $x$ ***összes előfordulása*** osztozik az $E$ értéken.)

  `sq(sq(sq 2))` $\rightarrow$ `x * x` *`[x = sq(sq 2)]`* $\rightarrow$ `x * x` *`[x = y * y] [y = sq 2]`* $\rightarrow$
  `x * x` *`[x = y * y] [y = 2 * 2]`* $\rightarrow$ `x * x` *`[x = y * y] [y = 4]`* $\rightarrow$
  `x * x` *`[x = 4 * 4]`* $\rightarrow$ `x * x` *`[x = 16]`* $\rightarrow$ `16 * 16` $\rightarrow$ `256`
- Gyakran nyerünk, de néha vesztünk a lusta kiértékeléssel.

  Láttuk, hogy `fun faci(0, p) = p | faci(n, p) = faci(n-1, n*p)` ***mohó kiértékelés*** esetén hatékonyabb `fac`-nál, mert az `n*p` szorzást azonnal végrehajtja. ***Lusta kiértékelés*** esetén az `n`-et azonnal kiszámítaná (szükség van `n` értékére az implicit `n = 0` vizsgálathoz), a `p` kiértékelését azonban a szorzások akkumulálásával késleltetné:

  `faci(4, 1)` $\rightarrow$ `faci(4-1, 4*1)` $\rightarrow$ `faci(3-1, 3*(4*1))` $\rightarrow$
  `faci(2-1, 2*(3*(4*1)))` $\cdots \rightarrow$ `24`



# ÖSSZETETT ADATTÍPUSOK



## Ennes és típusa

- Két különböző típusú értékből rekordot vagy párt képezhetünk. Pl.
  `{x = 2, y = 1.0} : {x : int, y : real}` és `(2, 1.0) : (int * real)`.
- A pár is csak szintaktikai édesítőszer. Pl.
  `(2, 1.0) = {1 = 2, 2 = 1.0} = {2 = 1.0, 1 = 2}`, de `(2, 1.0)` és `{1 = 1.0, 2 = 2}` különböző típusúak. Az `1` és a `2` ***mezőnevek*** (vö. szintaxis).
- Rekordot kettőnél több értékből is összeállíthatunk. Pl.
  `{nev= "Bea", tel= 3192144, kor= 19} : {kor :int, nev :string, tel :int}`.
  Egy hasonló rekord egészszám-mezőnevekkel:
  `{1 = "Bea", 3 = 3192144, 2 = 19}: {1 : string, 2 : int, 3 : int.`
  Az ***utóbbi*** azonos az alábbi ***ennessel*** (n-es, n-tuple):
  `("Bea", 19, 3192144) : (string * int * int)`,
  azaz `(string * int * int)` $\equiv$ `{1 = string, 2 = int, 3 = int}`.
- Egy rekordban a tagok sorrendje közömbös, az értékeket a mezőnév azonosítja.

## Ennes és típusa (folyt.)

- Egy ennesben a tagok sorrendje meghatározó! Pl. `(2, 1.0) : (int * real)`, de `(1.0, 2) : (real * int)`. A két ennes különböző!
- Ennes lehet függvény argumentuma és eredménye, összetett adat eleme stb. Példa: Fibonacci-számok iterációval.
  A definíció: $F_0 = 0; F_1 = 1; F_n = F_{n-2} + F_{n-1}, n > 1$.

```
(* iterfib(n, (prev, curr)) = a (prev, curr) 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));

(* fib n = az n-edik Fibonacci-szám
   fib : int -> int *)
fun fib 0 = 0
  | fib n = iterfib(n, (0, 1));
```



# FELHASZNÁLÓI ADATTÍPUSOK



## A `datatype` deklaráció

- `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: *konstruktora)* 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
```





## A `datatype` deklaráció (folyt.)

- `King` (király) csak egy van, ezért definiálhattuk konstruktorállandóként.
- A `Peer`-t (főnemest) 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.

## A datatype deklaráció (folyt.)

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

```
(* title p = p megszólítása
   title : person -> string *)
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` nevét összegyűjti a `person` típusú személyek egy listájából (a változatok sorrendje *fontos* az `_` miatt!):

```
(* sirs ps = az összes Knight nevének listája
   sirs : person list -> string list *)
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 fölsorolása 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 egyenletnek* tekintjük:

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





## 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 (p, r)= igaz, ha p magasabb rangú r-nél
   superior : person * person -> bool *)
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;
```





## A felsorolásos 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 *felsorolásos típust* létrehozni a `datatype` deklarációval. Pl.

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

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

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

## A felsorolásos típus `datatype` deklarációval (folyt.)

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

```
(* lady p = p főnemes hitvesének rangja
   lady : degree -> string *)
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 b = b negáltja
   Not : Bool -> Bool *)
fun Not True = False | Not False = True;
```





## Polimorf adattípusok

- Láttuk, hogy a `list` *postfix* pozíciójú *típusoperátor,* nem típus: a `datatype` deklaráció az adatkonstruktorok mellett *típuskonstruktort* 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:*

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

- A *hatospontot* közvetlenül a típusdeklarációban is definiálhatjuk:

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





## 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.)

- A megkülönböztetett egyesítés lehetővé teszi, hogy különböző típusokat használjunk ott, ahol egyébként csak egyetlen típust használhatnánk (vö. objektum-orientált programozás, ahol pl. egy *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 d = a d diszjunkt unió In1 címkéjű elemeinek konkatenációja
   concat : (string, 'a) disun list -> string *)
fun concat [] = ""
  | concat (In1 s :: ls) = s ^ concat ls
  | concat (In2 _ :: ls) = concat ls;
```

## Megkülönböztetett egyesítés (folyt.)

- Egy példa `concat` alkalmazására:

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

- Az `In1` konstruktorfüggvény típusa `'a -> ('a, 'b) disun`, ezért a `string` típusú `"Ő!"` 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 "Ő!", In2 King, In1 "Skócia"]` kifejezésben mindkét alaptípust lekötjük, ezért ennek a listának a típusa: `((string, person) disun) list`.
- Az `[In2 "Ő", In2 King, In1 "Skócia"]` kifejezés kiértékelése hibajelzést eredményez, mert a `'b` típusváltozót nem lehet ugyanabban a kifejezésben egyszer így, másszor úgy lekötni.



# BINÁRIS FÁK

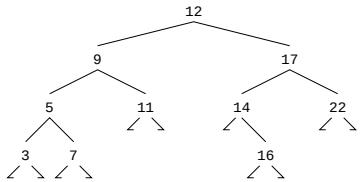


## Bináris fák `datatype` deklarációval

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

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

- Tekintsük például az alábbi fát:




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

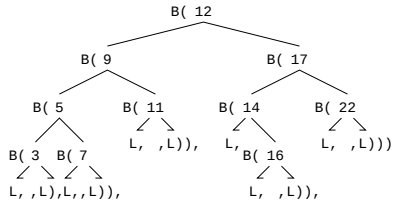




## Bináris fák `datatype` deklarációval (folyt.)

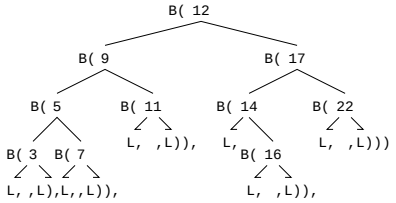
```
B(B(B(B(L,3,L),
       5,
       B(L,7,L)
      ),
     9,
     B(L,11,L)
    ),
   12,
   B(B(L,
       14,
       B(L,16,L)
      ),
     17,
     B(L,22,L)
    )
);
```

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



## Bináris fák datatype deklarációval (folyt.)

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




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





## Bináris fák datatype deklarációval (folyt.)

- Másféle fastruktúrákat is deklarálhatunk, pl.
  - kezdhetjük az 'a típusú értékkel, majd folytathatjuk előbb a bal, azután a jobb részfa megadásával,
  - felhasználhatjuk a levelet is értékek tárolására,
  - az értéket nem tároló üres csonkokat pedig E-vel jelölhetjük.
- A leírtak szerinti bináris fát hoz létre a következő deklaráció:

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

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

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





## Egyszerű műveletek bináris fákon

- nodes egy fa csomópontjait számlálja meg. Legyen

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

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

- nodes akkumulátort használó változata (nodesa):

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





## Egyszerű műveletek bináris fákon (folyt.)

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

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

- depth akkumulátort használó változata (deptha):

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

# KIÍRÁS, NYOMKÖVETÉS



## Kiírás

- `{TextIO.}print : string -> unit`

  `print s` = kiírja az `s` értékét a standard kimenetre, és azonnal kiüríti a puffert.

- `{Meta.}printVal : 'a -> 'a`

  `printVal e` = kiírja az `e` kifejezés értékét a standard kimenetre pontosan úgy, ahogyan az SML értelmező írja ki a „legfelső szinten”, és azonnal kiüríti a puffert. Eredményül visszaadja az `e` kifejezés értékét. ***Csak interaktív módban használható.***

- Példák:

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

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

*Megjegyzés.* A kapcsos zárójelek – `{` és `}` – között opcionálisan megadható modulnév áll. Például `{TextIO.}print` azt jelenti, hogy a függvény a `TextIO` modulban van definiálva, de az SML-értelmező a `print` nevet rövid alakban is felismeri.





## Kiírás (folyt.)

- `printVal`-lal tetszőleges típusú érték íratható ki. További példák:

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

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

- datatype t = L | B of t * t;
> New type names: =t
  datatype t = (t,{con B : t * t -> t, con L : t})
  con B = fn : t * t -> t
  con L = L : t
- val fa = B(B(B(L,B(L,B(L,B(B(L,L),L)))),L),B(L,L));
> val fa = B(B(B(L, B(L, B(L, B(B(L, L), L)))), L), B(L, L)) : t
- printVal fa;
B(B(B(L, B(L, B(L, B(B(L, L), L)))), L), B(L, L))> val it = B(B(B(L, B(L, B(L, B(
```





## Kiírás (folyt.)

- Az utolsó példában a kiírt sor túl hosszú lett, jó lenne eltörni a `>` jel előtt. Hogyan írathatunk ki egy újsor-jelet úgy, hogy az eredmény a `fa` érték maradjon? Például így, de ez elég körülményes:

```
- let val res = printVal fa;
      val _ = print "\n"
  in
     res
  end;
B(B(B(L, B(L, B(L, B(B(L, L), L)))), L), B(L, L))
> val it = B(B(B(L, B(L, B(L, B(B(L, L), L)))), L), B(L, L)) : t
```

- A `before` operátort az ilyen és hasonló dolgok kezelésére találták ki.

## Szekvenciális kifejezés (before)

- Az `x before y` kifejezés az ún. ***szekvenciális kifejezés*** egy változata.

  `{General.}before : 'a * 'b -> 'a`

  `x before y` = először az `x`-et, majd az `y`-t értékeli ki, eredménye az `x` értéke. Precedenciaszintje 0.
- Példa `before` használatára:

```
- printVal fa before print "\n";
B(B(B(L, B(L, B(L, B(B(L, L), L)))), L), B(L, L))
> val it = B(B(B(L, B(L, B(L, B(B(L, L), L)))), L), B(L, L)) : t
```

- Az `x before y`-hoz hasonló a `(x; y)` szekvenciális kifejezés, amely azonban az ***utolsó*** részkifejezésének az értékét adja eredményül.

```
- (print "A fa változó értéke =\n"; printVal fa before print "\n");
A fa változó értéke =
B(B(B(L, B(L, B(L, B(B(L, L), L)))), L), B(L, L))
> val it = B(B(B(L, B(L, B(L, B(B(L, L), L)))), L), B(L, L)) : t
```





## Szekvenciális kifejezés (;)

- Az `(x; y)` szekvenciális kifejezés, akárcsak az `x before y`, szintaktikai édesítőszer. Az `(x; y)` helyett írhatjuk, hogy:

```
let val _ = x in y end;
```





## Kiírás (folyt.)

- Hosszú lista, ill. egymásba skatulyázott adatszerkezetek esetén `printVal` (és maga az SML-értelmező is) alapesetben csak az első 200 listaelemet, ill. legfeljebb 20 szintet ír ki. A hosszat 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:

```
- printVal [1,2,3,4,5,6,7,8,9,10] before print "\n";
[1, 2, 3, 4, 5, 6, 7, ...]
> val it = [1, 2, 3, 4, 5, 6, 7, ...] : int list
- 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
```





## Kiírás (folyt.)

- Különböző típusú egyszerű értékeket alakítanak át füzérré a `toString` függvények:

```
Char.toString : char -> string
Int.toString  : int -> string
Real.toString : real -> string
Bool.toString : bool -> string
Word.toString : word -> string
```

## Nyomkövetés

- Az MOSML-ben nyomkövetés csak a program szövegébe beírt kiíró függvényekkel lehetséges.
- Példa: a `length` függvény két változatának kiértékelése
- A `length` „naív” változata

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

- A `length` „naív” változata kiíró függvényekkel

```
fun length ((_ : int) :: xs) =
        printVal(1 + (print " & "; printVal(length(printVal xs))
                      before print " $ "
                     )
                 )
        before print " #\n"
  | length []      = (print " * "; printVal 0 before print " %\n");
```





## Nyomkövetés (folyt.)

- A `length` iteratív változata

```
fun lengthi xs = let fun len (i, _::xs) = len(i+1, xs)
                       | len (i, [])    = i
                 in len(0, xs)
                 end;
```

- A `length` iteratív változata kiíró függvényekkel

```
fun lengthi xs =
    let fun len (i, (_ : int) :: xs) =
              len((print " "; printVal((printVal i before print " $ ") + 1)),
                  (print " & "; printVal xs)
                 )
                  before print "#\n"
          | len (i, []) = (print " * "; printVal i before print " %\n")
    in len(0, xs)
    end;
```





## Nyomkövetés

- `length` és egy alkalmazása

```
fun length ((_ : int) :: xs) =
        printVal(1 + (print " & "; printVal(length(printVal xs))
                      before print " $ "
                     )
                 )
        before print " #\n"
  | length []      = (print " * "; printVal 0 before print " %\n");

length [1,2,3];
 & [2, 3] & [3] & [] * 0 %
0 $ 1 #
1 $ 2 #
2 $ 3 #
```





## Nyomkövetés (folyt.)

- `lengthi` és egy alkalmazása

```
fun lengthi xs =
    let fun len (i, (_ : int) :: xs) =
              len((print " "; printVal((printVal i before print " $ ") + 1)),
                  (print " & "; printVal xs)
                 )
                  before print "#\n"
          | len (i, []) = (print " * "; printVal i before print " %\n")
    in len(0, xs)
    end;

lengthi [1,2,3];
 0 $ 1 & [2, 3] 1 $ 2 & [3] 2 $ 3 & [] * 3 %
#
#
#
```

## Nyomkövetés (folyt.)

- `length` és `lengthi` kiértékelésének összehasonlítása

```
length [1,2,3];                    | lengthi [1,2,3];
                                   |
 & [2, 3] & [3] & [] * 0 %         |  0 $ 1 & [2, 3] 1 $ 2 & [3] 2 $ 3 & [] * 3 %
0 $ 1 #                            | #
1 $ 2 #                            | #
2 $ 3 #                            | #
```

- További példák a `22fp.sml` állományban
  - `nodes` és akkumulátort használó `nodesa` változata
  - `depth` és akkumulátort használó `deptha` változata



# KIVÉTELKEZELÉS



## Kivételkezelés

- A leggyakoribb belső kivételek (többek között ld. a `General` könyvtárat)

| *Megnevezés* | *Művelet, amely a kivételt kiválthatja* |
|---|---|
| `Bind` | Értékdeklarációban a jobb oldali kifejezés nem illeszkedik a bal oldali mintára. |
| `Chr` | `chr pred succ` |
| `Div` | `/ div mod` |
| `Domain` | Az érték kilóg az értelmezési tartományból. |
| `Empty` | `hd tl last` |
| `Fail` | `compile load loadOne` |
| `Interrupt` | Megszakítás ctrl/c-vel. |
| `Io` | Ki/beviteli hiba. `Io of {function : string, name : string, cause : exn }` |
| `Match` | Mintaillesztési hiba `case` és `handle` kifejezésben, vagy függvényalkalmazásban. |
| `Option` | Hiba egy `Option` könyvtárbeli függvény alkalmazásakor. |
| `Ord` | Pl. `NJ93.ord ""` váltja ki; elavult. |
| `Overflow` | `~ + - * / div mod abs ceil floor round trunc` |
| `Size` | `^ array concat fromList implode tabulate translate vector` |
| `Subscript` | `copy drop extract nth sub substring take update` |





## Kivételkezelés (folyt.)

- Kivételt az `exception` kulcsszóval deklarálunk, a `raise` kulcsszóval jelzünk, a `handle` kulcsszóval bevezetett kifejezésben kezelünk.
- A kivételeket leggyakrabban hibák jelzésére használjuk.
- A kivételkonstruktor lehet állandó vagy függvény.
- A kivételkonstruktorállandó, ill. a kivételkonstruktorfüggvény típusa: `exn`.
- Az `exn` speciális típus:
  - a kivételkonstruktorok halmaza *bővíthető*,
  - az `exn` típust tartalmazó ún. *kivételcsomag* minden típussal kompatíbilis:

```
- fun // {den = 0, ...}   = raise Domain
     | // {num = n, den = d} = (real n) / (real d);
> val // = fn : {den : int, num : int} -> real
```

pedig

```
- Domain;
> val it = Domain : exn
```

## Kivételkezelés (folyt.)

- A `raise` kulcsszó olyan ***kivételcsomagot*** hoz létre, amelyben `exn` típusú érték is van.
- 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` ***kivételcsomagot*** eredményez, akkor az SML-futtatórendszer megpróbálja a `P1 ··· Pn` mintákra illeszteni.
  - Ha az első illeszkedő minta a `Pi` (`i = 1, 2, ..., n`) , akkor a kivételkezelő eredménye az `Ei` kifejezés eredménye.
  - Ha egyetlen minta sem illeszthető a kivételcsomagra, akkor a kivételkezelő továbbpasszolja a kivételcsomagot az előző hivási szintre.



# BINÁRIS FÁK



## Egyszerű műveletek bináris fákon (folyt.)

- `fulltree` $n$ mélységű ***teljes bináris fát*** épít, és a fa csomópontjait 1-től $2^n - 1$-ig beszámozza. Egy teljes bináris fában minden csomópontból pontosan két él indul ki, és minden levelének ugyanaz a szintje.

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

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

```
(* reflect =
   reflect : 'a tree -> 'a tree *)
fun reflect (N(v,t1,t2)) = N(v, reflect t2, reflect t1)
  | reflect L = L;
```



# KIÍRÁS, NYOMKÖVETÉS

# Nyomkövetés: nodes (akkumulátort nem használ)

```
(* tab : string -> string
   tab i = a sorok behúzásához használandó i füzér szóközökkel kiegészítve
*)
fun tab i = i ^ "    ";

fun nodes f =
      let (* nodes0 i f = a csomópontok száma f-ben; i a behúzáshoz használt füzér
             nodes0 : string -> 'a tree -> int *)
          fun nodes0 i (N(a, t1, t2)) =
                (print("\n" ^ i ^ "<"); printVal a : int; print "> ";
                 printVal(1 +
                          nodes0 (tab i) (printVal t2 before print " *") +
                          nodes0 (tab i) (printVal t1 before print " %")
                          before print "$ "
                         )
                 before print(" #\n" ^ i)
                )
            | nodes0 i L = (print("\n" ^ i); 0)
      in
          nodes0 "" f
      end;
```





# Nyomkövetés: nodesa (akkumulátort használ)

```
fun nodesa f =
      let (* nodes0 i (f, n) = n + a csomópontok száma f-ben;
                               i a behúzáshoz használt füzér
             nodes0 : string -> 'a tree * int -> int
          *)
          fun nodes0 i (N(a, t1, t2), n) =
                (print("\n" ^ i ^ "<"); printVal a : int; print "> ";
                 nodes0 (tab i) (printVal t1 before print(" %\n" ^ (tab i)),
                                 nodes0 (tab i) (printVal t2 before print(" *\n" ^
                                                                          (tab i)),
                                                 printVal(n+1) before print " $"
                                                )
                                )
                 before print(" #" ^ i)
                )
            | nodes0 i (L, n) = (* (print("\n" ^ i); n) *) n
      in
          nodes0 "" (f, 0)
      end;
```





# nodes és nodesa alkalmazása hét csomópontból álló teljes fára

```
f7 = N(1, N(2, N(4, L, L), N(5, L, L)), N(3, N(6, L, L), N(7, L, L))) : int tree
```

```
- nodes f7;                          | - nodesa f7;
                                     |
<1> N(3, N(6, L, L), N(7, L, L)) * | <1> N(2, N(4, L, L), N(5, L, L)) %
    <3> N(7, L, L) *                |     N(3, N(6, L, L), N(7, L, L)) *
        <7> L *                     |     1 $
            L %                     |     <3> N(6, L, L) %
            $ 1 #                   |         N(7, L, L) *
        N(6, L, L) %                |         2 $
        <6> L *                     |         <7> L %
            L %                     |             L *
            $ 1 #                   |             3 $ #
        $ 3 #                       |         <6> L %
    N(2, N(4, L, L), N(5, L, L)) % |             L *
                                    |             4 $ #         #
```

Folytatása a következő lapon.





# nodes és nodesa alkalmazása . . . (folyt.)

```
f7 = N(1, N(2, N(4, L, L), N(5, L, L)), N(3, N(6, L, L), N(7, L, L))) : int tree
```

```
 (nodes f7)                         |  (nodesa f7)
                                    |
    <2> N(5, L, L) *                |     <2> N(4, L, L) %
        <5> L *                     |         N(5, L, L) *
            L %                     |         5 $
            $ 1 #                   |         <5> L %
        N(4, L, L) %                |             L *
        <4> L *                     |             6 $ #
            L %                     |         <4> L %
            $ 1 #                   |             L *
        $ 3 #                       |             7 $ #         #     #
    $ 7 #                           |
                                    |
> val it = 7 : int                  | > val it = 7 : int
```

# Nyomkövetés: depth (akkumulátort nem használ)

```
fun depth f =
      let (* depth0 i f = az f fa mélysége; i a behúzáshoz használt füzér
             depth0 : string -> 'a tree -> int
          *)
          fun depth0 i (N(a : int, t1, t2)) =
                (print("\n" ^ i ^ "<"); printVal a : int; print "> ";
                 printVal(1 +
                          Int.max(depth0 (tab i) (printVal t2 before print " *"),
                                  depth0 (tab i) (printVal t1 before print " %")
                                 )
                         )
                 before print(" #\n" ^ i))
            | depth0 i L = (print( "\n" ^ i) ; 0)
      in
          depth0 "" f
      end;
```

- *Megjegyzés:* Az itt alkalmazott `nodes`, `nodesa`, `depth` és `deptha` függvények nyomkövetés nélküli változatát az előző előadásokon ismertettük.





# Nyomkövetés: deptha (akkumulátort használ)

```
fun deptha f =
      let (* depth0 i (f, d) = d + az f fa mélysége; i a behúzáshoz használt füzér
             depth0 : string -> 'a tree * int -> int *)
          fun depth0 i (N(a : int, t1, t2), d) =
                (print("\n" ^ i ^ "<"); printVal a : int; print "> ";
                 printVal(Int.max(depth0 (tab i) (printVal t2 before print(" *\n" ^
                                                                          (tab i)),
                                                  printVal(d+1) before print " $ "
                                                 ),
                                  depth0 (tab i) (printVal t1 before print(" %\n" ^
                                                                          (tab i)),
                                                  printVal(d+1) before print " & "
                                                 )
                                 )
                         )
                 before print(" #\n" ^ i)
                )
            | depth0 i (L, d) = (print( "\n" ^ i) ; d);
      in  depth0 "" (f, 0)
      end;
```





# depth és deptha alkalmazása hét csomópontból álló teljes fára

```
f7 = N(1, N(2, N(4, L, L), N(5, L, L)), N(3, N(6, L, L), N(7, L, L))) : int tree
```

```
- depth f7;

<1> N(3, N(6, L, L), N(7, L, L)) *
    <3> N(7, L, L) *
        <7> L *
            L %
            1 #
        N(6, L, L) %
        <6> L *
            L %
            1 #
        2 #
    N(2, N(4, L, L), N(5, L, L)) %
```

```
- deptha f7;

<1> N(3, N(6, L, L), N(7, L, L)) *
    1 $
    <3> N(7, L, L) *
        2 $
        <7> L *
            3 $
            L %
            3 &
            3 #
        N(6, L, L) %
        2 &
        <6> L *
            3 $
            L %
            3 &
            3 #
        3 #
    N(2, N(4, L, L), N(5, L, L)) %
    1 &
```

- Folytatása a következő lapon.





# depth és deptha alkalmazása hét csomópontból álló teljes fára

```
f7 = N(1, N(2, N(4, L, L), N(5, L, L)), N(3, N(6, L, L), N(7, L, L))) : int tree
```

```
 (depth f7)

    <2> N(5, L, L) *
        <5> L *
            L %
            1 #
        N(4, L, L) %
        <4> L *
            L %
            1 #
        2 #
    3 #

> val it = 3 : int
```

```
 (deptha f7)

    <2> N(5, L, L) *
        2 $
        <5> L *
            3 $
            L %
            3 &
            3 #
        N(4, L, L) %
        2 &
        <4> L *
            3 $
            L %
            3 &
            3 #
        3 #
    3 #

> val it = 3 : int
```

# BINÁRIS FÁK



## Lista előállítása bináris fa elemeiből

- `preorder`, `inorder` és `postorder` *bináris fából listát* állít elő. A három függvény abban különbözik egymástól, hogy az egy csomópontból az ott tárolt értéket mikor veszik ki, és milyen sorrendben járják be a bal, ill. a jobb részfát.
  - `preorder` először az értéket veszi ki, majd bejárja a bal, és azután a jobb részfát.
  - `inorder` először bejárja a bal részfát, majd kiveszi az értéket, és végül bejárja a jobb részfát.
  - `postorder` először bejárja a bal, majd a jobb részfát, és utoljára veszi ki az értéket.
- A következő megvalósítások egyszerűek, érthetőek, de nem elég hatékonyak a `@` operátor használata miatt.





## Lista előállítása bináris fa elemeiből (folyt.)

- Akkumulátor nem használó változatok
  - ```
    (* preorder f = az f fa elemeinek preorder sorrendű listája
       preorder : 'a tree -> 'a list *)
    fun preorder (N(v,t1,t2)) = v :: preorder t1 @ preorder t2
      | preorder L = [];
    ```
  - ```
    (* inorder f = az f fa elemeinek inorder sorrendű listája
       inorder : 'a tree -> 'a list *)
    fun inorder (N(v,t1,t2)) = inorder t1 @ (v :: inorder t2)
      | inorder L = [];
    ```
  - ```
    (* postorder f = az f fa elemeinek postorder sorrendű listája
       postorder : 'a tree -> 'a list *)
    fun postorder (N(v,t1,t2)) = postorder t1 @ postorder t2 @ [v]
      | postorder L = [];
    ```
- Az akkumulátort használó változatok nehezebben érthetőek, de *hatékonyabbak*.





## Lista előállítása bináris fa elemeiből (folyt.)

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

## Bináris fa előállítása lista elemeiből: balPreorder

- Listát *kiegyensúlyozott* (balanced) *bináris fává* alakítanak a következő függvények: balPreorder, balInorder és balPostorder; a különbség közöttük most is a bejárási sorrendben van.

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

- A hatékonyságot kisebb mértékben rontja, hogy List.take és List.drop egymástól függetlenül *kétszer* mennek végig a lista első felén.





## Bináris fa előállítása lista elemeiből: take'ndrop

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

```
(* take'ndrop(xs, k) = olyan pár, amelynek első tagja xs első k db
                       eleme, második tagja pedig xs maradéka
   take'ndrop : 'a list * int -> 'a list * 'a list
*)
fun take'ndrop (xs, k) =
      let fun td (xs, 0, ts) = (rev ts, xs)
            | td (x::xs, k, ts) = td(xs, k-1, x::ts)
            | td ([], _, ts) = (rev ts, [])
      in
          td(xs, k, [])
      end;
```

- take'ndrop felhasználása, nevezetesen az eredményül átadott pár miatt módosítani kell balpreorder felépítésén.





## Bináris fa előállítása lista elemeiből: balPreorder, újra

- Ez volt:

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

- Ez lett:

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





## Bináris fa előállítása lista elemeiből

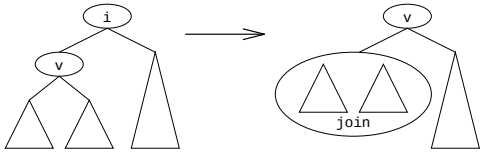
```
(* balInorder xs = az xs lista elemeiből álló, inorder bejárású,
                   kiegyensúlyozott fa
    balInorder: 'a list -> 'a tree
*)
fun balInorder (xxs as x::xs) =
      let val k = length xxs div 2
          val ys = List.drop(xxs, k)
      in N(hd ys, balInorder(List.take(xxs, k)), balInorder(tl ys))
      end
  | balInorder [] = L;
```

```
(* balPostorder xs = az xs lista elemeiből álló, postorder
                     bejárású, kiegyensúlyozott fa
    balPostorder: 'a list -> 'a tree
*)
fun balPostorder xs = balPreorder(rev xs);
```

- balInorder take'ndrop-pal való definiálását meghagyjuk gyakorló feladatnak.

## Elem törlése bináris fából

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




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





## Elem törlése bináris fából (folyt.)

- A `remove` rendezetlen bináris fából törli az `i` értékű elem *összes* előfordulását.
- A `join`-nal egyesítjük a törlés hatására létrejövő két részfát: a bal részfát lebontja, és közben az elemeit egyesével berakja a jobb részfába.

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

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



# LISTÁK HASZNÁLATA



## A „jó” számok” előállítása SML-függvénnyel

- „Jó” számok: keressük azokat a számokat, amelyek négyzete háromjegyű, és a szám fordítottjával kezdődnek (vö. Prolog-előadások).

```
(* joSzamok i = azoknak az i és 100 közötti kétjegyű számoknak a listája,
          amelyek négyzete háromjegyű, és a szám fordítottjával kezdődnek
   joSzamok : int -> int list
*)
fun joSzamok i =
       if i < 100
       then if i * i div 10 = i mod 10 * 10 + i div 10
            then i :: joSzamok (i+1)
            else joSzamok (i+1)
       else [];

joSzamok 10;
```

- Írjunk általánosabb megoldást: emeljük ki a szám jó voltának és a felső határ elérésének a vizsgálatát!

## A „jó” számok” előállítása SML-függvénnyel (folyt.)

- A `jsz` és a `lim` segédfügvények

```
(* jsz1 i = igaz, ha a kétjegyű i négyzete háromjegyű és a
            fordítottjával kezdődik
   jsz1 : int -> bool *)
fun jsz1 i = i * i div 10 = i mod 10 * 10 + i div 10;

(* jsz2 i = igaz, ha a háromjegyű i egyes és százas helyiértékű
            jegyei egyenlők
   jsz2 : int -> bool *)
fun jsz2 i = i > 100 andalso
             (i mod 10, i div 100) = (i div 100, i mod 10);

(* lim x i = igaz, ha i kisebb x-nél
   lim : int -> int -> bool *)
fun lim x i = i < x;
```





## A „jó” számok” előállítása SML-függvénnyel (folyt.)

- `joSzamok` egy szokásos megvalósítása

```
(* joSzamok lim f i = a (lim max)-ot és az f-et kielégítő i egészek
                      listája, ahol (lim max) a felső határ elérését,
                      f pedig i jó szám voltát vizsgálja
   joSzamok : (int -> bool) -> (int -> bool) -> int -> int list
*)
fun joSzamok lim f i =
       if lim i
       then if f i
            then i :: joSzamok lim f (i+1)
            else joSzamok lim f (i+1)
       else [];

joSzamok (lim 100) jsz1 10;
joSzamok (lim 300) jsz2 10;
```





## A „jó” számok” előállítása SML-függvénnyel (folyt.)

- `joSzamok` jobbrekurzív változata

```
(* joSzamok lim f i = a (lim max)-ot és az f-et kielégítő i egészek
                      listája, ahol (lim max) a felső határ elérését,
                      f pedig i jó szám voltát vizsgálja
   joSzamok : (int -> bool) -> (int -> bool) -> int -> int list *)
fun joSzamok lim f i =
      let fun jSz i zs =
                if lim i
                then jSz (i+1) (if f i then i :: zs else zs)
                else rev zs
      in
          jSz i []
      end;

joSzamok (lim 100) jsz1 10;
joSzamok (lim 300) jsz2 10;
```



# LEÁLLÁSI FELTÉTEL KEZELÉSE

## Leállási feltétel kezelése

- Háromféle megoldást mutatunk be:
  - igazságérték – `true`, `false` – visszaadásával,
  - az `'a option` típus alkalmazásával,
  - kivételkezeléssel.
- Példa: „jó” számok előállítása
- A következő érték előállítására és a felső határ elérésének vizsgálatára *speciális* függvényt írunk, háromféle változatban:
  `kov x i` jelzi, hogy `i` kisebb-e az `x` felső határnál, és ha igen, az `i` után következő értéket adja eredményül, egyébként az eredmény tetszőleges.





## A `kov` függvény háromféle változatban

- Igazságérték visszaadásával:

```
(* kov11 x i = (i+1, true), ha i < x (felső határ), egyébként (i, false)
   kov11 : int -> int -> int * bool *)
fun kov11 x i = if i < x then (i+1, true) else (i, false);
```

- `int option` alkalmazásával:

```
(* kov21 x i = SOME(i+1), ha i < x (felső határ), egyébként NONE
   kov21 : int -> int -> int option *)
fun kov21 x i = if i < x then SOME(i+1) else NONE;
```

- Kivételjelzéssel:

```
exception Limit;
(* kov31 x i = i+1, ha i < x (felső határ), egyébként a Limit kivétel
   kov31 : int -> int -> int *)
fun kov31 x i = if i < x then i+1 else raise Limit;
```





## Leállási feltétel kezelése igazságértékkel

- `kov : 'a -> 'a -> 'a * bool` `(* nxt = kov x *)`

```
(* findAll1 nxt f i = az f i összes megoldásának listája a felső határ elérését
         vizsgáló és a következő értéket eredményező nxt függvény segítségével
   findAll1 : ('a -> 'a * bool) -> ('a -> bool) -> 'a -> 'a list *)
fun findAll1 nxt f i =
      let fun fAll f z zs =
                let val (j, b) = nxt z
                in
                    if b then fAll f j (if f z then z::zs else zs)
                    else rev zs
                end
      in
          fAll f i []
      end;

findAll1 (kov11 100) jsz1 10;
findAll1 (kov11 300) jsz2 100;
```





## Leállási feltétel kezelése az `'a option` típus alkalmazásával

- `kov : 'a -> 'a -> 'a option` `(* nxt = kov x *)`

```
(* findAll2 nxt f i = az f i összes megoldásának listája a felső határ elérését
         vizsgáló és a következő értéket eredményező nxt függvény segítségével
   findAll2 : ('a -> 'a option) -> ('a -> bool) -> 'a -> 'a list
*)
fun findAll2 nxt f i =
      let fun fAll f z zs =
                case nxt z of
                    SOME j => fAll f j (if f z then z::zs else zs)
                  | NONE   => rev zs
      in
          fAll f i []
      end;

findAll2 (kov21 100) jsz1 10;
findAll2 (kov21 300) jsz2 100;
```

## Leállási feltétel kezelése kivételkezeléssel

- `kov : 'a -> 'a -> 'a (* nxt = kov x *)`

```
(* findAll3 nxt f i = az f i összes megoldásának listája a felső határ elérését
          vizsgáló és a következő értéket eredményező nxt függvény segítségével
   findAll3 : ('a -> 'a) -> ('a -> bool) -> 'a -> 'a list
*)
fun findAll3 nxt f i =
      let fun fAll f z zs = fAll f (nxt z) (if f z then z::zs else zs)
                            handle Limit => rev zs
      in
          fAll f i []
      end;

findAll3 (kov31 100) jsz1 10;
findAll3 (kov31 300) jsz2 100;
```



# LISTÁK RENDEZÉSE



## Listák rendezése

- `inssort` (beszúró rendezés),
- `quicksort` (gyorsrendezés),
- `tmsort` (felülről lefelé haladó összefésülő rendezés),
- `bmsort` (alulról felfelé haladó összefésülő rendezés),
- `smsort` (simarendezés).





## Beszúró rendezés

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

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

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

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

- Példa `inssort` alkalmazására:

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

# LISTÁK RENDEZÉSE



## Beszúró rendezés, generikus változat

- Az `ins` függvényt generikussá tesszük:

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





## Beszúró 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 összehasonlítjuk).

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





## Beszúró 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 `insort`-tal és `insort2`-vel:

```
inssort op<= [4.24, 4.1, 5.67, 1.12, 4.1, 0.33, 8.0];
inssort2 op>= [4, 4, 5, 1, 0, 8];
inssort op< (explode "qwerty");
```

- 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];
inssortLr op>= [4.24, 4.1, 5.67, 1.12, 4.1, 0.33, 8.0];
```

## A futási idők összehasonlítása

- 2000 elemet tartalmazó, véletlenszerűen előállított, illetve eredetileg éppen fordított sorrendű listák rendezéséhez szükséges futási időt mérünk.
  - Véletlen eloszlású egészlistát állít elő a `Random` könyvtárbeli `rangelist` függvény:

```
val xs2000R = Random.rangelist (1, 100000) (2000, 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 xs2000N = 1 -- 2000;
```





## A futási idők összehasonlítása (folyt.)

- A futási időt az alábbi függvénnyel mérjük meg:

```
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
          "Int sort with " ^ sortFn ^ ", " ^ cmpFn ^
          ", length = " ^ Int.toString(length xs) ^ " (" ^
          kind ^ "), time = " ^ Time.fmt 2 tim ^ " sec\n"
      end;

val t1N = futIdo (inssort, "inssort") (op>=, "op>=") (xs2000N, "increasing");
val t2N = futIdo (inssort2, "inssort2") (op>=, "op>=") (xs2000N, "increasing");
val t1R = futIdo (inssort, "inssort") (op>=, "op>=") (xs2000R, "random");
val t2R = futIdo (inssort2, "inssort2") (op>=, "op>=") (xs2000R, "random");
```





## A futási idők összehasonlítása (folyt.)

- A 2000 elemű, fordított sorrendű lista rendezése az akkumulátort nem használó `inssort`-változatokkal több mint 5 s-ig, az akkumulátort használó változatokkal csak 0.01 s-ig tart (linux, 233 MHz-es Pentium).

```
Int sort with inssort, op>=, length = 2000 (increasing), time = 5.18 sec
Int sort with inssort2, op>=, length = 2000 (increasing), time = 0.01 sec
Int sort with inssortRi, op>=, length = 2000 (increasing), time = 5.14 sec
Int sort with inssortLi, op>=, length = 2000 (increasing), time = 0.01 sec
```

- Eltűnik a különbség, ha ugyanolyan hosszú, de véletlenszerűen előállított listákat rendezünk.

```
Int sort with inssort, op>=, length = 2000 (random), time = 2.39 sec
Int sort with inssort2, op>=, length = 2000 (random), time = 2.26 sec
Int sort with inssortRi, op>=, length = 2000 (random), time = 2.40 sec
Int sort with inssortLi, op>=, length = 2000 (random), time = 2.24 sec
```





## Gyorsrendezés, akkumulátor használata nélkül

```
(* quicksort1 cmp xs = az xs elemeinek cmp szerint rendezett listája
   quicksort1 : ('a * 'a -> order) -> 'a list -> 'a list *)
fun quicksort1 cmp xs =
      let (* qs : 'a list -> 'a list
             qs ys = az ys elemeinek cmp szerint rendezett listája *)
          fun qs (m::ys) =
                let (* partition : 'a list *  'a list * ' list -> 'a list
                       partition (xs, ls, rs) = ... *)
                    fun partition (x::xs, ls, rs) =
                          if cmp(x, m) = LESS then partition(xs, x::ls, rs)
                                             else partition(xs, ls, x::rs)
                      | partition ([], ls, rs) = qs ls @ (m::qs rs)
                in
                    partition (ys, [], [])
                end
            | qs [] = []
      in
         qs xs
      end;
```

## Gyorsrendezés, akkumulátor használatával

```
(* quicksort2 cmp xs = az xs elemeinek cmp szerint rendezett listája
   quicksort2 : ('a * 'a -> order) -> 'a list -> 'a list *)
fun quicksort2 cmp xs =
      let (* qs : 'a list -> 'a list
             qs ys = az ys elemeinek cmp szerint rendezett listája  *)
          fun qs (m::ys) zs =
                let (* partition : 'a list * a' list * 'a list -> 'a list
                       partition (xs, ls, rs) = ... *)
                    fun partition (x::xs, ls, rs) =
                          if cmp(x, m) = LESS then partition(xs, x::ls, rs)
                                             else partition(xs, ls, x::rs)
                      | partition ([], ls, rs) = qs ls (m :: qs rs zs)
                in
                    partition (ys, [], [])
                end
            | qs [] zs = zs
      in
          qs xs []
      end;
```





## A futási idők összehasonlítása

```
val t1 = futIdo (inssort2, "inssort2") (op>=, "op>=") (xs2000R, "random");
                                                  (* ~ 2 M összehasonlítás! *)
val t3 = futIdo (quicksort2, "quicksort2")
                (Int.compare, "Int.compare") (xs20000R, "random");
val t4 = futIdo (Listsort.sort, "Listsort.sort")
                (Int.compare, "Int.compare") (xs20000R, "random");
                                                  (* ~ 300 E összehasonlítás *)

Int sort with inssort2, op>=, length = 2000 (random), time = 2.30 sec

Int sort with quicksort1, Int.compare, length = 20000 (random), time = 2.18 sec
Int sort with quicksort2, Int.compare, length = 20000 (random), time = 1.72 sec
Int sort with Listsort.sort, Int.compare, length = 20000 (random), time = 1.76 sec

Int sort with quicksort2, Int.compare, length = 200000 (random), time = 27.13 sec
Int sort with quicksort1, Int.compare, length = 200000 (random), time = 32.59 sec

val t7 = futIdo (Listsort.sort, "Listsort.sort") (Int.compare, "Int.compare")
         (Random.rangelist (1, 100000) (200000, Random.newgen()), "random");
! Uncaught exception:
! Out_of_memory
```





## Összefésülő rendezések

- Az összefésülő rendezéshez kell egy olyan függvény, amely két listát növekvő sorrendben egyesít:

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

- Hatékonyságromlást okoz, hogy a részeredményeket a veremben tároljuk. Iteratív megoldás esetén meg kell fordítani az eredménylistát.





## Fölülről lefelé haladó összefésülő rendezés

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

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

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

## Alulról fölfelé haladó összefésülő rendezés

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

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





## Alulról fölfelé haladó összefésülő rendezés (folyt.)

- `bmsort` a `sorting` segédfüggvényt használja, amelynek
  - első argumentuma a rendezendő lista,
  - második argumentuma a már rendezett részlistákat gyűjti,
  - harmadik argumentuma az adott lépésben összefuttatandó elem sorszáma.

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





## Alulról fölfelé haladó összefésülő rendezés (folyt.)

- Ha a rendezendő lista (`xs`) még nem fogyott el, soron következő eleméből `sorting` egyelemű listát (`[x]`) képez, és ezt a már rendezett részlisták listája (`lss`) elé fűzve meghívja a `mergepairs` segédfüggvényt. `mergepairs` az argumentumként átadott lista két azonos hosszúságú bal oldali részlistáját fűzi egybe, feltéve persze, hogy vannak ilyenek. `k` az éppen átadott elem sorszáma. Ha a rendezendő lista kiürült, `sorting` a kétszintű lista egyetlen elemét, a rendezett listát adja eredményül.

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





## Alulról fölfelé haladó összefésülő rendezés (folyt.)

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

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

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

## Alulról fölfelé haladó összefésülő rendezés (folyt.)

- A legrosszabb esetben $O(n \cdot log\, n)$ lépésre van szükség.
- A függvények működését egy példán is bemutatjuk. A kezdőhívás legyen

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

- Amíg `sorting` első argumentuma a nem üres (`x::xs`) lista, `sorting` saját magát hívja meg. A rekurzív hívás
  - első argumentuma a lépésenként egyre rövidülő `xs` lista,
  - második argumentuma a `mergepairs([x]::lss, k+1)` függvényalkalmazás eredménye, ahol kezdetben `lss = []`,
  - harmadik argumentuma (`k+1`) a már feldolgozott listaelemek száma.

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





## Alulról fölfelé haladó összefésülő rendezés (folyt.)

- A következő táblázatos elrendezés
  - `mergepairs` mindkét argumentumát,
  - a rekurzív `sorting` hívás itt `j`-vel jelölt 3. argumentumát, `k+1`-et, és
  - bináris számként `k`-t mutatja lépésről lépésre.
- A `sorting` függvény hívja `mergepairs`-t azokban a sorokban, amelyekben a `j` új értéket vesz föl, a többi helyen `mergepairs` hívása rekurzív.
- Ne feledjük, hogy `mergepairs`-nek listák listája az első argumentuma!
- A táblázat utolsó oszlopa a vonatkozó magyarázatra hivatkozik.
- Vegyük észre, hogy kapcsolat van az `llss` első eleme utáni listaelemek hossza és a `k` bitjei között! Ha `k` valamelyik bitje 1, akkor (balról jobbra haladva) az `llss` megfelelő listaelemének a hossza az adott bit helyiértékével egyenlő. A 0 értékű biteknek megfelelő listaelemek „hiányoznak” `llss`-ből.

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





## Alulról fölfelé haladó összefésülő rendezés (folyt.)

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

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

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

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

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

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





## Simarendezés

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

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

- `nextrun` eredménye egy pár, ennek
  - első tagja a futam (egy növekvő számsorozat),
  - a második tagja pedig a rendezendő lista maradéka.

## Simarendezés (folyt.)

- A futam csökkenő sorrendben bővül, kilépéskor a futamot meg kell fordítani. `smsorting` a futamokat ismételten előállítja és összefuttatja:

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

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

- A simarendezés egy változata `sort` néven megtalálható a `Listsort` könyvtárban.





## A futási idők összehasonlítása

```
fun futIdo2 (sort, sortFn) (xs, kind) =
      let val starttime = Timer.startCPUTimer()
          val zs = sort xs
          val {usr=tim,...} = Timer.checkCPUTimer starttime
      in  "Int sort with " ^ sortFn ^ ", length = " ^ Int.toString(length xs) ^
          " (" ^kind ^ "), time = " ^ Time.fmt 2 tim ^ " sec\n"
      end;

val t101 = futIdo2 (tmsort, "tmsort")
             ((Random.rangelist (1, 100000) (100000, Random.newgen())), "random");
val t102 = futIdo2 (bmsort, "bmsort")
             ((Random.rangelist (1, 100000) (100000, Random.newgen())), "random");
val t103 = futIdo2 (smsort, "smsort")
             ((Random.rangelist (1, 100000) (100000, Random.newgen())), "random");

Int sort with tmsort, length = 100000 (random), time = 10.96 sec
Int sort with bmsort, length = 100000 (random), time = 7.69 sec
Int sort with smsort, length = 100000 (random), time = 7.70 sec
Int sort with quicksort2, Int.compare, length = 100000 (random), time = 11.98 sec
Int sort with Listsort.sort, Int.compare, length = 100000 (random), time = 14.17 sec
```



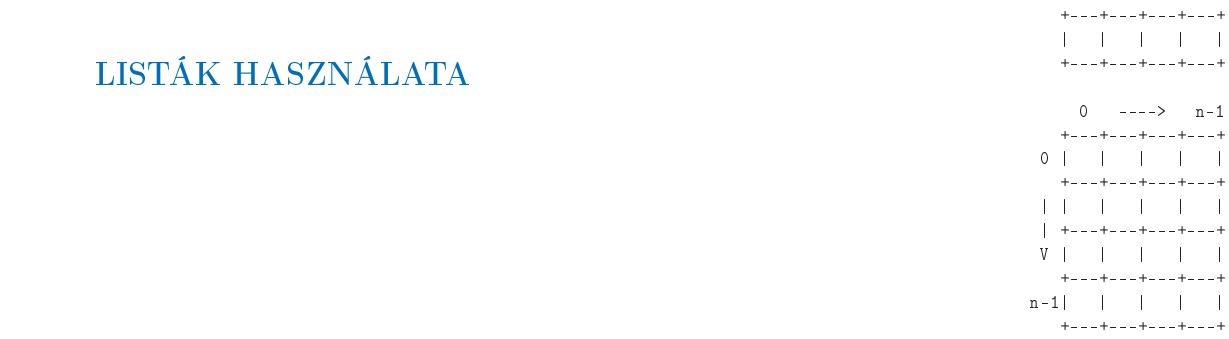
# LISTÁK HASZNÁLATA



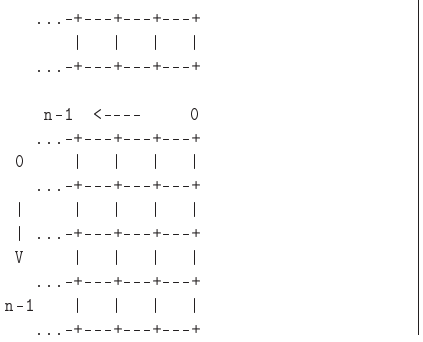
## $n$ vezér a sakktáblán

- Hányféleképpen rakható $n$ vezér a sakktáblára úgy, hogy ne üssék egymást?

A vezéreket tartalmazó mezők sorának számát az egyes oszlopokon belül egy `n` hosszú sorvektor adott oszlophoz rendelt mezőjébe írt `<= s < n` szám adja meg. Példa `n = 4` esetén:




A sorvektort (egy egyre bővülő) listával valósítjuk meg. Egy listához balról könnyű új elemeket fűzni, a táblát és a vezérek helyzetét leíró listát hossztengelye mentén tükrözzük.



## $n$ vezér a sakktáblán (folyt.)

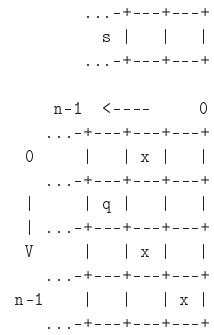
- Azt, hogy az új vezért üti-e a már táblára rakott másik vezér, a sorvektor vizsgálatával dönthetjük el, amely tehát azt adja meg, hogy a listaelemek indexe által meghatározott oszlopban és a listaelem értéke által meghatározott sorban vezér van.
  1. Az új vezér sorának száma, azaz az új listalem értéke nem fordulhat elő a lista már felépített részében.
  2. Az új vezér átlós irányban sem lehet egy vonalban más vezérrel a táblán. Ez azt jelenti, hogy ha a sorvektort jelentő lista elejére az `s` sorindexet akarjuk rakni, akkor az `i`-edik elemének az értéke, ha van ilyen eleme, nem lehet `s-(i+1)`, ill. `s+(i+1)`.
  3. A következő példa segít megvilágítani az esetet.





## $n$ vezér a sakktáblán (folyt.)

- Ha a 2-es oszlopba és az `s=1`-es sorba akarjuk lerakni az új vezért, akkor az `x`-szel jelölt mezőket kell megvizsgálnunk. Az eddig létrehozott listának (sorvektornak) két eleme van, ahol a lista fejének az indexe 0. A listafej értéke nem lehet s-1, sem s+1. A lista rekurzív algoritmussal dolgozható fel.








## $n$ vezér a sakktáblán (folyt.)

- „Ütésben van”-vizsgálat

```
(* utesbenVan : int list -> bool
   utesbenVan zs = igaz, ha a (hd zs) vezér nincs ütésben
   egyetlen (tl zs)-beli vezérrel sem
*)
fun utesbenVan [] = false
  | utesbenVan (z::zs) =
      let 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;
```





## $n$ vezér a sakktáblán (folyt.)

- Egy megoldás előállítása

```
exception Zsakutca;

(* vezerek0 : int -> int list
   vezerek0 n = a feladvány egy megoldása n vezér esetén
*)
fun vezerek0 n =
      let
          fun vez0 z zs =
                if z = 0 andalso utesbenVan zs orelse z = n then
                                                    raise Zsakutca
                else if length zs = n then rev zs
                else vez0 0 (z::zs) handle Zsakutca => vez0 (z+1) zs
      in
          vez0 0 []
      end;
```

## $n$ vezér a sakktáblán (folyt.)

- Több megoldás előállítása visszalépéssel

```
(* vezerek : int -> int list list
   vezerek n = a feladvány összes megoldásának listája n vezér esetén
*)
fun vezerek n =
      let
          fun vez0 z zs =
                if z = 0 andalso utesbenVan zs orelse z = n
                                                 then raise Zsakutca
                else if length zs = n then [rev zs]
                else (vez0 0 (z::zs) handle Zsakutca => []) @
                     (vez0 (z+1) zs handle Zsakutca => [])
      in
          vez0 0 []
      end;
```





## $n$ vezér a sakktáblán (folyt.)

- Több megoldás előállítása listák listájával

```
fun vezerek n =
      let fun vez0 z zs =
                if z = 0 andalso utesbenVan zs orelse z = n then []
                else if length zs = n then [rev zs]
                else vez0 0 (z::zs) @ vez0 (z+1) zs
      in  vez0 0 [] end;
```

- Több megoldás előállítása listák listájával, akkumulátor alkalmazásával

```
fun vezerek n =
      let fun vez0 z zs ws =
                if z = 0 andalso utesbenVan zs orelse z = n then ws
                else if length zs = n then rev zs :: ws
                else vez0 0 (z::zs) (vez0 (z+1) zs ws)
      in  vez0 0 [] [] end;
```



# BINÁRIS FÁK



## Bináris keresőfák

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

```
exception Bsearch of string;
```

- Szebb lenne, ha *generikus függvényeket* írnánk; ezt gyakorló feladatnak hagyjuk.

## Bináris keresőfák: blookup

- A `blookup` függvény adott kulcshoz tartozó értéket ad vissza egy rendezett bináris fából:

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





## Bináris keresőfák: binsert

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

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





## Bináris keresőfák: bupdate

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

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



# LUSTA LISTÁK

## Lusta lista

- Olyan lista, amelynek a farka függvény, ezáltal késleltetjük a kiértékelését.
- Ily módon *végtelen listákat* hozhatunk létre.
- A lusta listának hátrányai, veszélyei is vannak, pl.
  - egy lusta lista bármely részét megjeleníthetjük, de sohasem az egészet;
  - két lusta lista elemeiből páronként képezhetünk egy harmadikat, de nem számíthatjuk ki egy lusta lista elemeinek az összegét, nem kereshetjük meg benne a legkisebbet, nem fordíthatjuk meg az elemek sorrendjét;
  - úgy kell rekurziót definiálnunk, hogy nincs alapeset;
  - egy program befejeződése helyett csak azt igazolhatjuk, hogy az eredmény tetszőleges véges része véges idő alatt előáll.
- A lusta listát sorozatnak *(sequence)* nevezzük, és a `seq` típusoperátort használjuk a létrehozására.

```
datatype 'a seq = Nil | Cons of 'a * (unit -> 'a seq)
```





## Lusta lista (folyt.)

- Egy sorozat fejét adja eredményül a `head` függvény; abortál, ha üres sorozatra alkalmazzuk.

```
(* head : 'a seq -> 'a *)
fun head (Cons(x, _)) = x;
```

- Egy sorozat farkát adja eredményül a `tail` függvény; abortál, ha üres sorozatra alkalmazzák.

```
(* tail : 'a seq -> 'a seq *)
fun tail (Cons(_, xf)) = xf();
```

A sorozat farka `unit -> 'a seq` típusú ***függvény,*** erre illesztjük az `xf` mintát `tail` fejében; `tail` törzsében `xf`-et a `()` argumentumra kell alkalmazni.





## Lusta lista (folyt.)

- `consq(x, xq)` `x`-et berakja az `xq` sorozatba:

```
(* consq : 'a * 'a seq -> 'a seq *)
fun consq (x, xq) = Cons(x, fn () => xq);
```

  - Ha a `consq` függvényt alkalmazzuk, mondjuk, az `(x, E)` argumentumra, az SML a `consq(x, E)` kifejezést ***nem lustán*** értékeli ki, hiszen alapvetően mohó kiértékelésű.
  - Ha `E` kiértékelésének eredményét `xq`-val jelöljük, akkor `consq(x, E)` kiértékelése a fenti definíció szerint `Cons(x, fn () => xq)`-t eredményez.
  - A `consq`-beli `fn () => xq` függvény nem késlelteti a farok (a példában `E`) kiértékelését `consq` alkalmazásakor.
  - A lusta kiértékelés érdekében a híváskor is a `Cons(x, fn () => E)` alakot kell használnunk, `consq(x,E)` nem jó.
  - Az explicit `fn () => E` késlelteti a kiértékelést, és ezzel ***szükség szerinti hivatkozást*** valósít meg.





## Lusta lista (folyt.)

- Példaként a korábban megismert `from` és `take` függvények lusta változatait mutatjuk be.
- A `fromq k` sorozat egészek `k`-tól induló végtelen sorozata.

```
(* fromq : int -> int seq *)
fun fromq k = Cons(k, fn () => fromq(k+1));
```

- `takeq(xq, n)` az `xq` sorozat első `n` eleméből képzett listát adja vissza:

```
(* takeq : 'a seq * int -> 'a list *)
fun takeq (xq, 0) = []
  | takeq (Cons(x, xf), n) = x :: takeq(xf(), n-1)
  | takeq (Nil, n) = [];
```

- Az `'a seq` típus nem egészen lusta kiértékelésű: egy nemüres sorozat fejét a rendszer mindig feldolgozza.

## Egyszerű függvények lusta listákra

- A kiszámíthatóság érdekében egy függvény eredményének tetszőleges véges része az argumentum véges részétől függhet csak.
- Amikor az eredményre szükség van, akkor ez az igény váltja ki az argumentum feldolgozását.
- Első példánkban egészeket egyesével emelünk négyzetre. Amikor szükség van rá, az eredmény farka (egy függvény) alkalmazza a `squareq` függvényt az argumentum farkára.

```
(* squareq : int seq -> int seq *)
fun squareq Nil: int seq = Nil
  | squareq (Cons (x, xf)) = Cons(x * x, fn () => squareq(xf()));
```

- Két lusta lista hasonlóan adható össze.

```
(* addq : (int seq * int seq) -> int seq *)
fun addq (Cons (x, xf), Cons(y, yf)) = Cons(x+y, fn () => addq(xf(), yf()))
  | addq _: int seq = Nil;
```





## Egyszerű függvények lusta listákra (folyt.)

- Az `appendq` függvény addig nem nyúl `yq`-hoz, amíg `xq` ki nem ürül – vagyis csak akkor nyúl hozzá, ha `xq` véges. Véges sorozatot `consq`-val készíthetünk.

```
(* appendq : 'a seq * 'a seq -> 'a seq *)
fun appendq (Cons (x, xf), yq) = Cons(x, fn () => appendq (xf(), yq))
  | appendq (Nil, yq) = yq;
```

- Most érthetjük meg, hogy miért kellett a típusdefinícióban a `Nil` konstruktorállandót definiálni.





## Magasabb rendű függvények lusta listákra

- A `map` lusta változata:

```
(* mapq : ('a -> 'b) -> 'a seq -> 'b seq *)
fun mapq f (Cons (x, xf)) = Cons(f x, fn () => mapq f (xf()))
  | mapq f Nil = Nil;
```

- A `filter` lusta változata:

```
(* filterq : ('a -> bool) -> 'a seq -> 'a seq *)
fun filterq p (Cons (x, xf)) = if p x
                               then Cons(x, fn () => filterq p (xf()))
                               else filterq p (xf())
  | filterq p Nil = Nil;
```

- `squareq` a korábban látottnál sokkal egyszerűbben definiálható `mapq`-val:

```
val squareq = mapq (fn i => i * i);
```





## Magasabb rendű függvények lusta listákra (folyt.)

- Olyan számsorozatot állítunk elő, amelyben 50-nél nagyobb, 7-esre végződő egészek vannak:

```
filterq (fn n => n mod 10 = 7) (fromq 50);
```

- Az `iterateq` függvény – a `fromq` egy általánosítása – a következő sorozatot állítja elő (vö. `repeat`-tel): $[x, f(x), f(f(x)), \cdots, f^k(x), \cdots]$.

```
(* iterateq : ('a -> 'a) -> 'a -> 'a seq *)
fun iterateq f x = Cons(x, fn () => iterateq f (f x));
```

- `fromq`-t `iterateq`-val így definiálhatjuk:

```
(* fromq : int -> int seq *)
val fromq = iterateq (fn i => i+1);
```

## Álvéletlen számok

- Hagyományos álvéletlenszám-generátorok: olyan eljárások, amelyek egy *frissíthető változóban* tárolják a *seed* (mag) értéket – ebből állítják elő egy következő hívásnál a következő álvéletlen számot.
- Lusta listaként megvalósítva: a következő álvéletlen szám csak szükség esetén áll elő.

```
(* randseq : int -> real seq *)
local val a = 16807.0 and m = 2147483647.0
      (* nextrandom : real -> real
      *)
      fun nextrandom seed =
            let val t = a * seed
            in t - real(floor(t/m)) * m
            end
in
   fun randseq s = mapq (secr op/ m) (iterateq nextrandom (real s))
end;
```





## Álvéletlen számok (folyt.)

- Ha a `nextrandom`-ot 1.0 és 21474836467.0 közötti `seed`-re alkalmazzuk, ugyanebbe a tartományba eső más értéket állít elő az `a * seed mod m` művelettel. (A valós számokat a túlcsordulás elkerülésére használjuk.)
- A lusta lista előállítására `iterateq`-t `nextrandom`-ra és `seed` valós számmá alakított kezdőértékére alkalmazzuk. `mapq` gondoskodik arról, hogy a lusta listában minden értéket elosszunk `m`-mel, és így `randseq` 0.0-nál nem kisebb és 1.0-nél kisebb értékeket adjon eredményül. Látható, hogy a lusta lista a megvalósítás részleteit szépen elrejti a felhasználó elől.
- Az előállított álvéletlen-számok 0.0-nál nem kisebb és 1.0-nél kisebb valós számok; `mapq`-val alakíthatjuk át őket 0 és 1 közötti egészekké:

```
mapq (floor o secl 10.0 op*) (randseq 1);
```





## Prímszámok előállítása *eratosztenészi szitával*

- Az algoritmus:
  1. Vegyük az egészek 2-vel kezdődő sorozatát: (2, 3, 4, 5, 6, 7, ...).
  2. Töröljük az összes 2-vel osztható számot: (3, 5, 7, 9, 11, ...).
  3. Töröljük az összes 3-mal osztható számot: (5, 7, 11, 13, 17, 19, ...).
  4. Töröljük az összes 5-tel osztható számot: (7, 11, 13, 17, 19, ...).
  5. Töröljük az összes ...
- A sorozat első eleme mindig a következő prím. A sorozatban azok a számok maradnak benne, amelyek az eddig előállított prímekkel nem oszthatók.

```
(* sift : int -> int seq -> int seq *)
fun sift p = filterq (fn n => n mod p <> 0);
```

- A `sift` a `p` argumentum többszöröseit törli egy lusta listából.





## Prímszámok előállítása *eratosztenészi szitával* (folyt.)

- A `sieve`-nek már csak ismételten alkalmaznia kell `sift`-et a megfelelő lusta listára. Mivel ez a lusta lista sohasem üres, nem kell az üres lusta listára illeszkedő változatot írnunk.

```
(* sieve : int seq -> int seq *)
fun sieve (Cons (p, nf)) = Cons(p, fn () => sieve(sift p (nf())))
  | sieve Nil = Nil;
```

## Négyzetgyökvonás Newton-Raphson módszerrel

- `nextapprox` $x_k$-ból $x_{k+1}$-et számítja ki az $x_{k+1} = \frac{\frac{a}{x_k}+x_k}{2}$ képlet alapján.

```
(* nextapprox : real -> real -> real *)
fun nextapprox a x = (a/x + x)/2.0;
```

- A befejeződés megállapítására egyszerű tesztet írunk:

```
(* within : real -> real seq -> real *)
fun within (eps: real) (Cons (x, xf)) =
      let val Cons (y, yf) = xf()
      in
         if abs (x-y) <= eps then y else within eps (Cons (y, yf))
      end;
```

A `(Cons (y, yf))` és az `xf()` lusta lista ugyanaz: az `else`-ágban azért használjuk az elsőt, mert `xf()` meghívása költségesebb.





## Négyzetgyökvonás Newton-Raphson módszerrel (folyt.)

- Ezzel

```
(* qroot : real -> real *)
fun qroot a = within 1E~6 (iterateq (nextapprox a) 1.0);
```

- A példában világosan különválasztjuk a leállásvizsgálatot (termination test) a következő jelölt előállításától.

  Most az abszolút különbséget ($|\, x - y \,| < \varepsilon$) teszteljük, de vizsgálhatnánk pl. a relatív különbséget ($|\, \frac{x}{y} - 1 \,| < \varepsilon$) vagy az $\frac{|x-y|}{\frac{|x|+|y|}{2}+1} < \varepsilon$ feltételt.

  A feladat többi része független attól, hogy milyen leállásvizsgálatot alkalmazunk, és így is kell megfogalmazni a megoldást.





## Négyzetgyökvonás Newton-Raphson módszerrel (folyt.)

- Írjunk függvényt a következő jelölt előállítására, és rejtsük el a részleteket:

```
(* approxq : real -> real seq *)
fun approxq a =
      let (* nextapprox : real -> real
          *)
      fun nextapprox x = (a/x + x) / 2.0
      in iterateq nextapprox 1.0
      end;
```

- Ezzel `qroot` egy „tisztább” változata:

```
(* qroot : real -> real *)
val qroot = within 1E~6 o approxq;
```





## Keresztszorzatokból álló lista

- Legyen `xq` és `yq` egy-egy sorozat. Képezzünk új sorozatot az $(x_i, y_j)$ párokból, ahol $x_i \in$ `xq` és $y_j \in$ `yq`!
- Először hagyományos listákra oldjuk meg a feladatot `map` és `pair` alkalmazásával.
- `xs` és `ys` egy-egy lista. Képezzünk listát az $(x_i, y_j)$ párokból, ahol $x_i \in$ `xs` és $y_j \in$ `ys`!
- `map`-et, `pair`-t és `List.concat`-ot alkalmazva juthatunk el a keresett függvényhez.

```
(* pair : 'a -> 'b -> ('a * 'b) *)
fun pair x y = (x, y);
```

- A `pair`-t a `map`-pel az `ys` lista elemeire alkalmazva olyan párokból álló listát kapunk eredményül, amelyben a párok első tagja a rögzített `x` érték, a második tagja pedig az `ys` egy-egy eleme.

```
map (pair x) ys
```

## Keresztszorzatokból álló lista (folyt.)

- Hogyan érhetjük el, hogy az `x` végigfusson az `xs` lista összes elemén? Az eddig szabad `x`-et kössük le egy függvény argumentumaként:

  ```
  fn x => map (pair x) ys
  ```

  majd alkalmazzuk újból a `map`-et erre a függvényre és `xs`-re:

  ```
  map (fn x => map (pair x) ys) xs
  ```

- Listák listáját kapjuk eredményül, mert a belső `map` már listát adott vissza, amelynek minden eleméből újabb listát képeztünk a külső `map`-pel. `List.concat` elvégzi a szükséges simítást:

  ```
  (* pairs : 'a list -> 'b list -> ('a * 'b) list *)
  fun pairs xs ys = flat (map (fn x => map (pair x) ys) xs);
  ```





## Keresztszorzatokból álló lusta lista

- A `pairss`-hez hasonlóan állíthatjuk elő párok lusta listájának lusta listáját:

  ```
  (* pairqq : 'a seq -> 'b seq -> ('a * 'b) seq seq *)
  fun pairqq xq yq = mapq (fn x => mapq (pair x) yq) xq;
  ```

- Az eredmény véges része kiíratható `takeqq`-val, amely a bal felső saroktól számított első `m` sorból és `n` oszlopból álló téglalapot jeleníti meg az `xqq` lusta listából:

  ```
  (* 'a takeqq : (int * int) * 'a seq seq -> 'a list list *)
  fun takeqq ((m, n), xqq) = map (secl n takeq) (takeq(m, xqq));
  ```

- Példa: olyan lusta lista, amelyben a párok első tagja az egymás után következő egészek 30-tól kezdve, második tagja pedig a prímszámok 2-től kezdve:

  ```
  - pairqq (fromq 30) primes;
  > val it = Cons (Cons ((30, 2), fn), fn): (int * int) seq seq
  ```





## Keresztszorzatokból álló lusta lista (folyt.)

- ```
  - takeqq((3, 5), it);
  > val it = [[(30, 2), \ldots,  (30, 11)],
              [(31, 2), \ldots,  (31, 11)],
              [(32, 2), \ldots,  (32, 11)]] : (int * int) list list
  ```

- Ha ki akarjuk simítani a lusta listát, egy `List.concat`-hoz hasonló, lusta listákra alkalmazható függvénnyel nem megyünk semmire: ha `xq` végtelen, `appendq (xq, yq) = xq`. Azonban két lusta lista elemei páronként egymásba ékelhetők:

  ```
  (* interleaveq : 'a seq * 'a seq -> 'a seq *)
  fun interleaveq (Nil, yq) = yq
    | interleaveq (Cons (x, xf), yq) = Cons(x, fn () => interleaveq(yq, xf()));
  ```

- `interleaveq` a rekurzív hívásban váltogatja a két lusta listát.

- ```
  - takeq(10, interleaveq(fromq 0, fromq 50));
  > val it = [0, 50, 1, 51, 2, 52, 3, 53, 4, 54] : int list
  ```





## Keresztszorzatokból álló lusta lista (folyt.)

- `enumerate`: lusta listák lusta listájából egyetlen lusta listát állít elő. Legyen a kétszeres mélységű lusta lista feje `xq` és a farka `xqf`; alkalmazzuk `enumerate`-et rekurzívan `xqf`-re, majd az eredményt ékeljük `xq`-ba:

  ```
  (* enumerate : 'a seq seq -> 'a seq *)
  fun enumerate Nil = Nil
    | enumerate (Cons (xq, xqf)) = interleaveq (xq, enumerate(xqf()));
  ```

  Ez a „megoldás” nem jó, mert a „ végtelen” lusta lista miatt a rekurzió nem ér véget: az SML-ben, amely alapvetően mohó kiértékelésű, a rekurzív hívást késleltetni kell.

- Több esetet kell megkülönböztetnünk:

  ```
  (* 'a enumerate : 'a seq seq -> 'a seq *)
  fun enumerate Nil = Nil
    | enumerate (Cons (Nil, xqf)) = enumerate (xqf())
    | enumerate (Cons (Cons (x, xf), xqf)) =
                  Cons(x, fn () => interleaveq(enumerate(xqf()),xf()));
  ```

# Keresztszorzatokból álló lusta lista (folyt.)

- Ha a bemenő lusta lista üres, készen vagyunk. Ha nem üres, meg kell vizsgálni a lusta lista fejét: ha ez üres, akkor folytatni kell a rekurzív hívást, ha nem üres, akkor az explicit `fn () => ...` függvénydefinícióval *késleltetni kell* a rekurziót.
- Példa: pozitív egészekből álló párok egy lusta listáját!

```
- val posintqq = pairqq (fromq 1) (fromq 1);
> val posintqq = Cons (Cons ((1, 1), fn), fn):(int * int) seq seq
- takeq(15, enumerate posintqq);
> val it = [(1,1), (2,1), (1,2), (3,1), (1,3), (2,2),
            (1,4), (4,1), (1,5), (2,3), (1,6), (3,2),
            (1,7), (2,4), (1,8)] : (int * int) list
```