# LISTÁK



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

- `max` egy változata egészekre

```
(* max: int * int -> int
   max (n, m) = n és m közül a nagyobbik
*)
fun max (n, m) = if n > m then n else m
```

# POLIMORFIZMUS



## Polimorfizmus

- Nézzük az identitásfüggvényt: `fun id x = x`.
- Mi az `x` típusa? Tetszőleges, 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*.
- Nézzük az egyenlőségfüggvényt: `fun eq (x, y) = x = y`.
- Mi az `x` és `y` típusa? Tetszőleges, típusát szintén *típusváltozó* jelöli:
  `val ''a eq = fn : ''a * ''a -> bool`.
- A *két* percjellel kezdődő típusnév (pl. `''a`, olvasd *alfa*) az ún. *egyenlőségi típus változója*.

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).

# LISTÁK



## Lista legnagyobb elemének megkeresése (folyt.)

- Hogyan tehető *polimorffá* a `maxl` függvény? Úgy, hogy ún. *generikus* függvényként definiáljuk: *aktuális paraméterként* kapja azt a többszörösen terhelhető függvényt, amely két érték közül a nagyobbikat kiválasztja.

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

- `max` mindig ugyanaz, mégis újra és újra átadjuk argumentumként a rekurzív ágban. Javíthatja a hatékonyságot, ha *lokális kifejezést* használunk.

```
fun maxl max zs = let fun mxl [] = raise Empty
                        | mxl [y] = y
                        | mxl (y::ys) = max(y, mxl ys)
                  in
                      mxl zs
                  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 szintaktikus 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

# Lista (folyt.)

## Változatok `max`-ra

- ```
  (* charMax : char * char -> char
     charMax (a, b) = a és b közül a nagyobbik
  *)
  fun charMax (a, b) = if ord a > ord b then a else b;
  ```
  vagy egyszerűen (`ord` nélkül)
  ```
  fun charMax (a : char, b) = if a > b then a else b;
  ```
- ```
  (* pairMax : ((int * real) * (int * real)) -> (int * real)
     pairMax (n, m) = n és m közül lexikografikusan a nagyobbik
  *)
  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;
  ```
- ```
  (* stringMax : string * string -> string
     stringMax (s, t) = s és t közül a nagyobbik
  *)
  fun stringMax (s : string, t) = if s > t then s else t;
  ```





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

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

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