FUNKCIONÁLIS PROGRAMOZÁS

A funkcionális programozás néhány jellemzője

Funkcionális, más néven applikatív programozás

- Funkcionális = függvényalapú, függvényközpontú
- Applikatív = (függvényeket argumentumokra) alkalmazó
- Függvényjel (más néven operátor)
- Argumentum (más néven paraméter vagy operandus)
- Deklaráció
- Kifejezés, érték, kiértékelés (más néven egyszerűsítés, redukció)

Fő különbségek az imperatív és a funkcionális programozás között

- Változó: frissíthető, ill. nem frissíthető (vö. értékadás, ill. kötés)
- Ismétlés: ciklus, ill. rekurzió (vö. helyesség bizonyítása teljes indukcióval)
- Függvények: függvényeljárás, ill. „tiszta” függvény (vö. mellékhatás, ill. mellékhatás-mentesség)
- Állapotváltozások sorozata, ill. időtlenség, emlékezet nélküliség

Az SML néhány jellemzője

A Standard Meta Language (SML) néhány jellemzője

- Rekurzív típus: lineáris (lista) és nemlineáris (pl. fa)
- Magasabb rendű függvény (argumentuma és/vagy eredménye függvény)
- „Erősen típusos” *(strong typing;* ellenőrzés fordításkor)
- Típuslevezetés *(type derivation)*
- Polimorfizmus (többszörös terheléses és paraméteres)
- Modularitás (szignatúra, struktúra, funktor)
- Absztrakt típus

SML-megvalósítások

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

TÍPUS ÉS FÜGGVÉNY

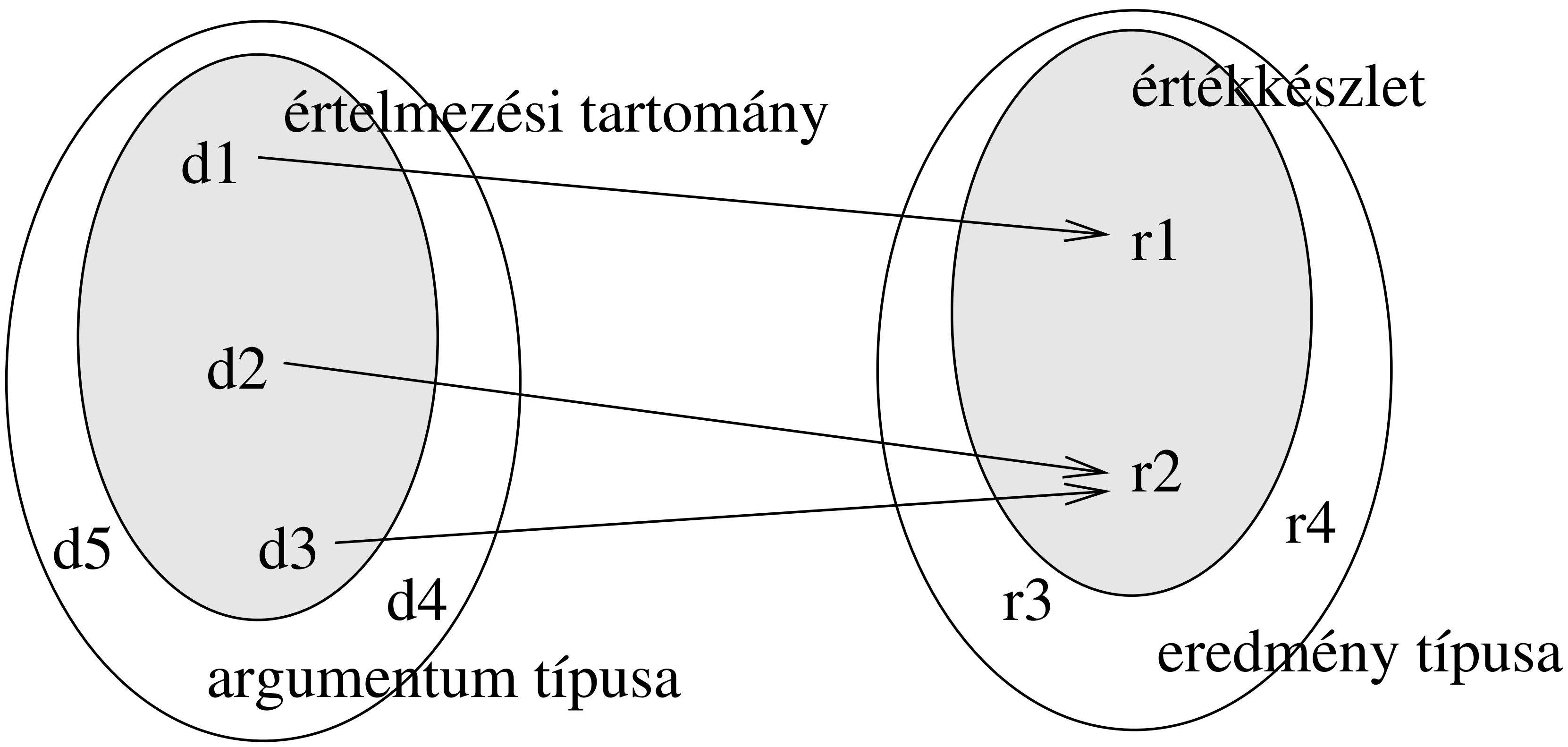
A típus és a függvény fogalma

- A típus fogalma
  - ◇ A típus értékek egy halmaza (pl. egész típus ~ az egész számok egy (rész)halmaza)
  - ◇ Tetszőleges típus jelölése az ún. *típuselméletben:* α, β ...
  - ◇ Típusállandó (azaz konkrét típus) jelölése az SML-ben: `int`, `real`, `char`, `string` ...
  - ◇ Típusváltozó jelölése az SML-ben α, β ... helyett: `'a`, `'b` ...
- A függvény fogalma
  - ◇ A függvény valamely D halmaznak valamely R halmazba való olyan *egyértelmű* leképzése, amelyet a (d, r) rendezett párok halmaza ad meg, 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
  - ◇ A függvény értelmezési tartománya $\subseteq$ az argumentum típusa
  - ◇ A függvény értékkészlete $\subseteq$ az 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 α az argumentum, a β az eredmény típusát jelöli
  - ◇ A függvény maga is: érték. *Függvényérték.*
  - ◇ Fontos: a függvényérték *nem* a függvény egy *alkalmazásának* az eredménye!
  - ◇ Példák függvényértékre
    - sin (a típusa: *valós* $\to$ *valós*)
    - round (a típusa: *valós* $\to$ *egész*)
    - $\circ$ (függvénykompozíció; a típusa: $((\beta \to \gamma) * (\alpha \to \beta)) \to (\alpha \to \gamma))$
  - ◇ Példák függvényalkalmazásra
    - $\mathrm{round}\ 5.4 = 5$, azaz egy *egész* típusú érték az eredménye ennek a függvényalkalmazásnak
    - $\mathrm{round} \circ \mathrm{sin}$ (a típusa: *valós* $\to$ *egész*)
    - $(\mathrm{round} \circ \mathrm{sin})1.0 = 1$ (a típusa: *egész*)

A függvény mint leképzés



Függvények osztályozása, ill. alkalmazása

- Függvények osztályozása
  - Parciális függvény: értelmezési tartomány $\subset$ argumentum típusa (figyelem: 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 $f : \alpha \rightarrow \beta$ injektív függvény inverze: $f^{-1} : \beta \rightarrow \alpha$
  - 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 az f-et alkalmazzuk az e-re.
  - Általánosabban: az $f e$ kifejezésben az e tetszőleges olyan kifejezés, amelynek az értéke az f értelmezési tartományába esik.
  - Még általánosabban: az $f e$ kifejezésben az f függvényértéket adó tetszőleges kifejezés, e pedig tetszőleges olyan kifejezés, amelynek 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,*
     - ◇ pl. $f[1, 2, 3]$ az f függvény alkalmazását jelenti az $[1, 2, 3]$ *listára.*
  2. A függvényt több egymás utáni lépésben alkalmazzuk az argumentumokra, pl. $f\ 1\ 2 \equiv (f\ 1)\ 2$ 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 $f\ 1\ 2$ függvényalkalmazás (vég)eredményét.
- Az $f\ 1\ 2$ esetben az f függvényt *részlegesen alkalmazható* függvénynek nevezzük.
- A programozó szabadon dönthet, hogy a függvényt részlegesen alkalmazható vagy pl. egy párra alkalmazható formában írja meg. A különbség *csak* a szintaxisban van. A részlegesen alkalmazható változat, mint látni fogjuk, rugalmasabban használható.
- Infix jelölés: $x \oplus y \equiv$ a $\oplus$ 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.
- FONTOS! 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. `+`, `*` (mindkettő infix), `real`, `round` (mindkettő prefix)
  - ◇ Könyvtári függvények, pl. `Math.sin`, `Math.cos`, `Math.pi` (0 argumentumú!)
  - ◇ Felhasználó által definiálható függvények, pl. `terulet`, `/\`, `head`

SML-példa: Egyszeres Hamming-távolságú ciklikus kód

- A függvényt pl. *táblázattal* adhatjuk 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 általában *algoritmussal* adjuk meg (nem táblázattal), egyébként
  - ◇ az argumentum 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 n => fn i => (i + 1) mod n`
- A függvény néhány alkalmazása:
  - ◇ `(fn n => (fn i => (i + 1) mod n)) 128 111`
  - ◇ `(fn n => (fn i => (i + 1) mod n)) 4 ~7`
  - ◇ `(fn n => (fn i => (i + 1) mod n)) 128 6.0` – hiba!

Értékdeklaráció SML-ben: függvényérték deklarálása

- Név kötése függvényértékhez
  - `val kovKod = fn 00 => 01 | 01 => 11 | 11 => 10 | 10 => 00`
  - `val incMod = fn n => fn i => (i + 1) mod n`
- Szintaktikus édesítőszerrel
  - ```
    fun kovKod 00 = 01
      | kovKod 01 = 11
      | kovKod 11 = 10
      | kovKod 10 = 00
    ```
  - `fun incMod n i = (i + 1) mod n`
- Alkalmazásuk argumentumra
  - `kovKod 01`
  - `incMod 128 111`
    ```

Fejkomment

Írjunk *fejkommentet* minden (függvény)érték-deklarációhoz!

- ```
  (* kovKod cc = az egyszeres Hamming-távolságú, kétbites,
                 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
  ```
- ```
  (* incMod n i = (i+1) modulo n szerint
     PRE: n > i >= 0
  *)
  fun incMod n i = (i+1) mod n
  ```

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 (szintaktikus édesítőszer).
  - ◇ A `::`-ot négyespontnak vagy *cons*-nak olvassuk (vö. *constructor*, ami a függvény hagyományos neve a λ-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::ys)`	`[X,Y|Ys]`	különböző

- Minták

A `[]`, `nil` konstruktorállandóval és a `::` konstruktoroperátorral felépített kifejezések, 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
   hd xs = a nem-üres xs első eleme (az xs feje)
*)
fun hd (x :: _) = x;
```

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

```
(* tl : 'a list -> 'a list
   tl xs = a nem-üres xs első utáni elemeinek az eredetivel
           azonos sorrendű listája (az xs farka)
*)
fun tl (_ :: xs) = xs;
```

- `hd` és `tl` *parciális* függvények. Ha könyvtárbeli megfelelőiket (`List.hd`, `List.tl`) üres listára alkalmazzuk, `Empty` néven *kivételt* jeleznek.
- `_` az ún. *mindenesjel*: mindenre illeszkedő minta. Kifejezésben – pl. az egyenlőségjel jobb oldalán – nem használható.

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

- Egy lista hosszát adja eredményül a `length` függvény (vö. `List.length`).

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

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

```
(* isum : int list -> int
   isum ns = az ns egészlista elemeinek összege *)
fun isum []        = 0
  | isum (n :: ns) = n + isum ns;
```

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

```
(* rprod : real list -> real
   rprod xs = az xs valós lista elemeinek szorzata *)
fun rprod []        = 1.0
  | rprod (x :: xs) = x * rprod xs;
```

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:`
	`! Empty`
`List.tl [1, 2, 3];`	`> val it = [2, 3] : int list`
`List.tl [];`	`! Uncaught exception:`
	`! 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`

LOKÁLIS KIFEJEZÉS

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          ahol  d nemüres deklarációsorozat,
in e                 e nemüres kifejezés.
end
```

- Példa:

```
(* length : 'a list -> int
   length zs = a zs elemeinek száma *)
fun length zs =
      let
          (* len : 'a list -> int
             len zs = a zs elemeinek száma *)
          fun len []          = 0
            | len (_ :: zs) = 1 + len zs
      in
          len zs
      end;
```

LISTÁK

Lista: adott függvény alkalmazása lista minden elemére (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 [x_1, x_2, ..., x_n] = [f x_1, f x_2, ..., f x_n]

- `map` definíciója:

```
(* map : ('a -> 'b) -> 'a list -> 'b list
   map f xs = az xs f szerint transzformált elemeiből álló lista
*)
fun map f [] = []
  | map f (x :: xs) = f x :: map f xs;
```

- `map` típusa:

  `('a -> 'b) -> 'a list -> 'b list` = `('a -> 'b) -> ('a list -> 'b list)`

  Ugyanis a −> jobbra köt!

  Azaz ha `map`-et egy `'a -> 'b` típusú függvényre alkalmazzuk, akkor egy `'a list -> 'b list` típusú **függvényt** ad eredményül. E függvényt 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 „végtelen” a rekurzió).
- Bizonyítása hossz szerinti *strukturális indukcióval* lehetséges (visszavezethető a teljes indukcióra).

```
fun map f [] = []
  | map f (x :: xs) = f x :: map f xs;
```

  - 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

  $\texttt{p } x_1 = \texttt{true}, \texttt{p } x_2 = \texttt{false}, \texttt{p } x_3 = \texttt{true}, \ldots, \texttt{p } x_{2k+1} = \texttt{true}$,

  akkor

  $\texttt{filter p } [x_1, x_2, x_3, \ldots, x_{2k+1}] = [x_1, x_3, \ldots, x_{2k+1}]$.

Lista: `filter` (folyt.)

- `filter` definíciója

```
(*  filter : ('a -> bool) -> 'a list -> 'a list
    filter p zs = a zs p-t kielégítő elemeiből álló lista
*)
fun filter _ [] = []
  | filter p (x :: xs) =
      if p x then x :: filter p xs else filter p xs;
```

- `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 egy `('a list -> 'a list)` függvényt ad eredményül. Ezt egy `'a list` típusú listára alkalmazva egy `'a list` típusú listát kapunk.

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.

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 szintaktikus édesítőszer. Pl.

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

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

- Rekordot kettőnél több értékből is összeállíthatunk. Pl.

  `{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-tuple):

  `("Bea", 19, 3192144) : (string * int * int)`

  azaz

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

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

SML-SZINTAXIS

SML-szintaxis: különleges állandók

- 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: Idézőjelek (`"`) 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`$xxxx$	Az $xxxx$ 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 ún. 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 : :: :> _ | = => -> #
```

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	

ABSZTRAKCIÓ, ADATTÍPUSOK

Gyenge és erős absztrakció, adattípusok

- Gyenge absztrakció: a név szinonima, az adatszerkezet részei továbbra is hozzáférhetők.
- Erős absztrakció: a név új dolgot (entitást, objektumot, izét) jelöl, az adatszerkezet részeihez csak korlátok között lehet hozzáférni.
- `type`: gyenge absztrakció

  Pl. `type rat = {num : int, den : int}`

  ◇ Új nevet ad egy típuskifejezésnek (vö. értékdeklaráció).

  ◇ Segíti a programszöveg megértését.
- `abstype`: erős absztrakció

  ◇ Új típust hoz létre: név, műveletek, ábrázolás, jelölés.

  ◇ Van helyette jobb: `datatype` + modulok
- `datatype`: modulok nélkül gyenge, modulokkal erős absztrakció

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

  ◇ Új entitást hoz létre.

  ◇ Rekurzív és polimorf is lehet.

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

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

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

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

A `datatype` deklaráció (folyt.)

- Ha más lenne a változatok sorrendje, a `_::ps` minta nemcsak a `King`-re, a `Peer`-re és a `Peasant`-ra illeszkedne (ti. ezek helyett áll a példában), hanem a `Knight`-ra is.
- Az összes diszjunkt eset 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:

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

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

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

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

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

Polimorf adattípusok

- Láttuk, hogy a `list` *postfix* pozíciójú *típusoperátor,* nem típus: a `datatype` deklaráció az adatkonstruktorok mellett *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 : (string, 'a) disun list -> string
   concat d = a d diszjunkt unió In1 címkéjű
              elemeinek konkatenációja           *)
fun concat [] = ""
  | concat (In1 s :: ls) = s ^ concat ls
  | concat (In2 _ :: ls) = concat ls
```

Megkülönböztetett egyesítés (folyt.)

- Egy példa `concat` alkalmazására:

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

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