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: definíciók, adat- és típuskonstruktorok

- 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` *adatkonstruktorállandó*.
  - A `nil` helyett általában a `[]` jelet használjuk (szintaktikus édesítőszer).
  - A `nil` típusa: `'a list`.
  - Az `'a` *típusváltozót* jelöl, a `list`-et *típuskonstruktornak* nevezzük.
  - A `::` *adatkonstruktorfüggvény* új listát hoz létre egy elemből és egy (esetleg üres) listából.
  - A `::` típusa `'a * 'a list -> 'a list`, infix pozíciójú, 5-ös precedenciájú, jobbra köt. Infix pozíciója miatt *adatkonstruktoroperátornak* is nevezzük.
  - A `::`-ot *négyespontnak* vagy *cons*-nak olvassuk (vö. *constructor,* ami ennek az adatkonstruktorfüggvénynek a hagyományos neve a λ-kalkulusban és egyes funkcionális nyelvekben).

Lista: jelölések, minták

- Példák
  - Lista létrehozása adatkonstruktorokkal

```
[]              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` adatkonstruktorállandóval és a `::` adatkonstruktoroperátorral felépített kifejezések, valamint a `[x1, x2, ..., xn]` listajelölés mintában is alkalmazhatók.

Lista: fej (hd), farok (tl)

- A nemüres lista első eleme a lista *feje*.

```
(* hd : 'a list -> 'a
   hd xs = a nemüres xs első eleme (az xs feje)
*)
fun hd (x :: _) = x;
```

- A nemüres lista első utáni elemeiből áll a lista *farka*.

```
(* tl : 'a list -> 'a list
   tl xs = a nemüres 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 _ (aláhúzás) az ún. *mindenesjel*, azaz a mindenre illeszkedő minta. Figyelem: a mindenesjel kifejezésben – pl. 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 lista 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`

`map`: adott függvény alkalmazása egy lista minden elemére

- 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` polimorf függvény!):

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

- `map` típusa (mivel a `->` típusoperátor jobbra köt!):

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

- A `map` egy ún. *részlegesen alkalmazható*, magasabbrendű függvény: ha egy `'a -> 'b` típusú függvényre alkalmazzuk, akkor egy `'a list -> 'b list` típusú **függvényt** ad eredményül. A kapott függvényt egy `'a list` típusú listára alkalmazva egy `'b list` típusú listát kapunk.

- `map` – teljes nevén `List.map` – belső függvény az SML-ben.

A program helyességének (informális) 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 *alapeset* kezeléséről, ui. van nem rekurzív ág).

Néhány belső, ill. könyvtári függvény

- `explode : string -> char list` – a füzér karaktereiből álló lista
  pl. `explode "abc" = [#"a", #"b", #"c"]`
- `implode : char list -> string` – a karakterlista elemeiből álló füzér
  pl. `implode [#"a", #"b", #"c"] = "abc"`
- `map`-nek más változatai is vannak, amelyek egyéb összetett adatokra alkalmazhatók. Például
  - `String.map : (char -> char) -> string -> string`
  - `Vector.map : ('a -> 'b) -> 'a vector -> 'b vector`
- A `Char` könyvtárban sok hasznos ún. *tesztelő* függvény található, például:
  - `Char.isLower : char -> bool` – igaz az angol ábécé kisbetűire
  - `Char.isSpace : char -> bool` – igaz a hat formázó karakterre
  - `Char.isAlpha : char -> bool` – igaz az angol ábécé betűire
  - `Char.isAlphaNum : char -> bool` – igaz az angol ábécé betűire és a számjegyekre
  - `Char.isAscii : char -> bool` – igaz a 128-nál kisebb ascii-kódú karakterekre

  pl. `Char.isSpace #"\t" = true; Char.isAlphaNum #"!" = false`

`filter`: adott predikátumot kielégítő elemek kiválogatása

- 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_{2k+1} `= true`, akkor `filter p [`x_1`,` x_2`,` x_3`, ...,` x_{2k+1}`] = [`x_1`,` x_3`, ...,` x_{2k+1}`]`.
- `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. A kapott függvényt egy `'a list` típusú listára alkalmazva egy `'a list` típusú listát kapunk.

Lista legnagyobb elemének megkeresése

- Üres listának nincs legnagyobb eleme,
- egyelemű lista egyetlen eleme a „legnagyobb”,
- legalább kételemű lista legnagyobb eleme
  - az első elem és a maradéklista elemeinek legnagyobbika közül a nagyobb

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

  - az első két elem legnagyobbika és a maradéklista elemei közül a legnagyobb.

```
fun maxl' [] = raise Empty
  | maxl' [n] = n
  | maxl' (n::m::ns) = maxl'(Int.max(n,m)::ns)
```

    - `maxl`-lel szemben itt a klózok sorrendje közömbös (a minták diszjunktak).
    - `maxl'` *jobbrekurzív,* tárigénye konstans.

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 *`let`-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
```

Változatok `max`-ra

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;
  ```
  ```

Listák összefűzése (append) és megfordítása (nrev)

- Két lista összefűzése (append, infix változatban @)

$$[x_1,\ldots,x_m]@[y_1,\ldots,y_n] = [x_1,\ldots,x_{m-1}]@(x_m::[y_1,\ldots,y_n]) = \ldots = [x_1,\ldots,x_m,y_1,\ldots,y_n]$$

Az xs-t először az elemeire bontjuk, majd hátulról visszafelé haladva fűzzük az elemeket az ys-hez, ugyanis a listákat csak elölről tudjuk építeni. A lépések száma $O(n)$.

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

- Lista naív megfordítása (nrev)

$$\texttt{nrev}[x_1,x_2,\ldots,x_m] = \texttt{nrev}[x_2,\ldots,x_m]@[x_1] = \texttt{nrev}[\ldots,x_m]@[x_2]@[x_1] = \ldots = [x_m,\ldots,x_1]$$

A lista elejéről levett elemet egyelemű listaként tudjuk a végéhez fűzni. A lépések száma $O(n^2)$.

```
(* nrev : 'a list -> 'a list
   nrev xs = xs megfordítva *)
fun nrev [] = []
  | nrev (x::xs) = (nrev xs) @ [x]
```

Listák megfordítása: példa `nrev` alkalmazására

- Egy példa `nrev` egyszerűsítésére

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

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

A `::` és a `@` jobbra kötnek, precedenciaszintjük 5.

`nrev([1,2,3,4])` $\rightarrow$ `nrev([2,3,4])@[1]` $\rightarrow$ `nrev([3,4])@[2]@[1]`
$\rightarrow$ `nrev([4])@[3]@[2]@[1]` $\rightarrow$ `nrev([])@[4]@[3]@[2]@[1]`
$\rightarrow$ `[]@[4]@[3]@[2]@[1]` $\rightarrow$ `[4]@[3]@[2]@[1]`
$\rightarrow$ `4::[]@[3]@[2]@[1]` $\rightarrow$ `4::[3]@[2]@[1])`
$\rightarrow$ `[4,3]@[2]@[1])` $\rightarrow$ `4::([3]@[2])@[1])`
$\rightarrow$ `[]@[4]@(3::[2,1]` $\rightarrow$ `[]@[4]@[3,2,1]` $\rightarrow$ ...

`nrev` rossz hatékonyságú: a lépések száma $O(n^2)$.

Listák összefűzése (`revApp`) és megfordítása (`rev`)

- Egy lista elemeinek egy másik lista elé fűzése fordított sorrendben (`revApp`)

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

`revApp` lépésszáma arányos a lista hosszával. Segítségével `rev` hatékonyan:

```
(* rev : 'a list -> 'a list
   rev xs = xs megfordítva
*)
fun rev xs = revApp (xs, [])
```

Egy 1000 elemű listát `rev` 1000 lépésben, `nrev` $\frac{1000 \cdot 1001}{2} = 500500$ lépésben fordít meg. Hatalmas a nyereség!

- `append` – `@` néven, infix operátorként – és `rev` beépített függvények, `List.revApp` pedig `List.revAppend` néven könyvtári függvény az SML-ben.

ÖSSZETETT ADATTÍPUSOK

Rekord és ennes

- 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.