

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
- Kifejezés, kiértékelés (más néven egyszerűsítés, redukció), érték

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. „tisza” 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.polym1.org>
- Standard ML of New Jersey (sml): <http://cm.bell-labs.com/cm/cs/what/smlnj>

TÍPUS ÉS FÜGGVÉNY

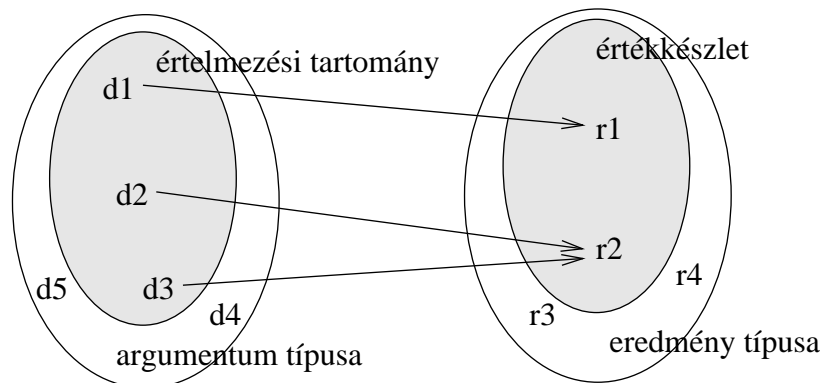
A típus és a függvény fogalma

- A típus fogalma
 - ◊ A típus értékek egy halmaza (pl. egész típus ~ az egész számok egy (rész)halmaza)
 - ◊ Jelölése az ún. *típuselméletben*: $\alpha, \beta \dots$
 - ◊ Típusállandó (azaz konkrét típus) jelölése az SML-ben: `int, real, char, string`...
 - ◊ Típusváltozó jelölése az SML-ben $\alpha, \beta \dots$ 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: $\alpha \rightarrow \beta$, 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* \rightarrow *valós*)
 - * `round` (a típusa: *valós* \rightarrow *egész*)
 - * `o` (függvénykompozíció; a típusa: $((\beta \rightarrow \gamma) * (\alpha \rightarrow \beta)) \rightarrow (\alpha \rightarrow \gamma)$)
 - ◊ Példák függvényalkalmazásra
 - * `round 5.4 = 5`, azaz egy *egész* típusú érték az eredménye ennek a függvényalkalmazásnak
 - * `round o sin` (a típusa: *valós* \rightarrow *egész*)
 - * `(round o 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 f -et alkalmazzuk 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 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 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 (\lfloor , $\backslash t$, $\backslash n$ stb.). Nulla db formázó karakter elegendő pl. $a(e)$ 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. $+$, $*$ (mindkettő infix), $real$, $round$ (mindkettő prefix)
 - ◊ Könyvtári függvények, pl. $Math.sin$, $Math.cos$, $Math.pi$ (0 argumentumú!)
 - ◊ Felhasználó által definiálható függvények, pl. $terulet$, $/\$, $head$

FÜGGVÉNYEK AZ SML-BEN

SML-példa: Egyszeres Hamming-távolságú ciklikus kód

- A függvényt *táblázattal* adjuk meg:

00	01	fn	00	=>	01
01	11		01	=>	11
11	10		11	=>	10
10	00		10	=>	00
- Változatok („klózek”): 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 *algoritmussal* adjuk meg (nem táblázattal)
 - ◊ n nem lehetne változó,
 - ◊ túl sok változatosat 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 incMod = fn n => fn i => (i + 1) mod n`
 - ◊ `val kovKod = fn 00 => 01 | 01 => 11 | 11 => 10 | 10 => 00`
- Szintaktikus édesítőszerrel
 - ◊ `fun incMod n i = (i + 1) mod n`
 - ◊ `fun kovKod 00 = 01`
 | `kovKod 01 = 11`
 | `kovKod 11 = 10`
 | `kovKod 10 = 00`
- Alkalmazásuk argumentumra
 - ◊ `incMod 128 111`
 - ◊ `kovKod 01`

Fejkomment

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

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

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 `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 `A ::`-ot négyespontnak vagy *cons*-nak olvassuk (vö. *constructor*, ami a függvény hagyományos neve a λ -kalkulusban és egyes funkcionális nyelvekben).

LISTÁK

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	azonos	SML	Prolog	különböző
<code>[]</code>	<code>[]</code>	azonos	<code>(x::xs)</code>	<code>[X Xs]</code>	különböző
<code>[1,2,3]</code>	<code>[1,2,3]</code>	azonos	<code>(x::y::ys)</code>	<code>[X,Y Ys]</code>	különböző

• Minták

A `[]`, *nil* konstruktorállandóval és a `::` konstruktoroperátorral felépített kifejezések, valamint a `[x1, x2, ..., xn]` listajelölés mintában is alkalmazhatók.

Lista: fej (hd), fark (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;
```

LOKÁLIS KIFEJEZÉS

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 kifejezést* használunk, ha ismétlődő részkifejezéseket *csak egyszer* akarunk kiszámítani, vagy akkor, ha bizonyos értékeket a program többi része elől *el akarunk rejtetni*.

- 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

Listák: 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: $\text{map } f [x_1, x_2, \dots, x_n] = [f x_1, f x_2, \dots, f x_n]$

- 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.

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 fark transzformálásával kapott lista elé fűzve valóban a várt eredményt kapjuk.
- ◊ A kiértékelés 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).

PROGRAMHELYESSÉG

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: gyűjtjük ki a kisbetűket egy karakterlistából!


```
List.filter Char.isLower (explode "VaLtOgAtVa") =
      ["a", "t", "g", "t", "a"];
```
- Általában, ha


```
p x1 = true, p x2 = false, p x3 = true, ..., p x2k+1 = true.
      akkor
      filter p [x1, x2, x3, ..., x2k+1] = [x1, x3, ..., x2k+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 (\rightarrow 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, annyiféle; 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 *polimorf* 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 édesítőszert!
 - `if b then e1 else e2` \equiv `(fn true => e1 | false => e2) b`
 - `e1 andalso e2` \equiv `(fn true => e2 | false => false) e1`
 - `e1 orelse e2` \equiv `(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` \equiv `e1 andalso e2`
 - `if e1 then true else e2` \equiv `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` *nehéz kiértékelésű* megfelelői:

<pre>(* && (a, b) = a / b && : bool * bool -> bool *) fun op&& (a, b) = a andalso b; infix 2 &&;</pre>	<pre>(* (a, b) = a b : bool * bool -> bool *) fun op (a, b) = a orelse b; infix 1 ;</pre>
--	---

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


```

(* 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 változatok, `List.take`, ill. `List.drop`, ha az `xs` listára alkalmazzuk, $i < 0$ vagy $i > length\ xs$ esetén `Subscript` néven kivételt jelez.

HALMAZMŰVELETEK

Halmazműveletek: „benne van-e?” (isMem) és „ha új, tedd bele” (newMem)

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

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

infix isMem
```

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

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

newMem, ha a sorrendtől eltekintünk, halmazt hoz létre.

Halmazműveletek: „listából halmaz” (setof)

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

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

- 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: „unió” (union) és „metszet” (inter)

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

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

- Két halmaz metszete

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

Halmazműveletek: „részhalmaza-e” (isSubset) és „egyenlők-e” (isSetEq)

- Részhalmaza-e egy halmaz egy másiknak?

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

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 : 'a list * 'a list -> bool
   isSetEq(xs, ys) = az xs elemeiből álló halmaz egyenlő-e
                    az ys elemeiből álló halmazzal
*)
fun isSetEq (xs, ys) = (xs isSubset ys) andalso (ys isSubset xs)
```

Halmazműveletek: „halmaz hatványhalmaza” (powerSet)

- Az S halmaz hatványhalmaza összes részhalmazának a halmaza, az S -t és a $\{\}$ -t is beleértve.
- S hatványhalmaza úgy állítható elő, hogy kivesszük S -ből az x elemet, majd *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.
- Miközben a fenti elvet rekurzív módon alkalmazzuk, tehát felsorolhatjuk az $S - \{x\}$ stb. részhalmazait, gyűjtjük a *már kiválasztott* elemeket. Egy-egy rekurzív lépésben a gyűjtő vagy változatlan (T), vagy kiegészül az x elemmel ($T \cup \{x\}$).
- A `pws` függvényben a `base` argumentumban gyűjtjük a halmaz *már kiválasztott* elemeit; kezdetben üres.
- $\text{pws}(xs, \text{base}) = \{S \cup \text{base} \mid S \subseteq xs\}$, azaz $xs \cup \text{base}$ azon részhalmazainak a listája, amelyek teljes egészében tartalmazzák a `base` halmazt.

KIFEJEZÉSEK KIÉRTÉKELÉSE

Halmazműveletek: „halmaz hatványhalmaza” (powerSet)

- Ezzel a `pws` függvény:

```
(* pws : 'a list * 'a list -> 'a list list
   pws(xs, base) = mindazon halmazok listája, amelyek előállnak xs egy
                   részhalmazának és a base halmaznak az uniójaként
*)
fun pws ([], base) = [base]
  | pws (x::xs, base) = pws(xs, base) @ pws(x, x::base)
```

- `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)` rekurzív módon `base`-ben gyűjti az x elemeket, vagyis előállítja az összes olyan halmazt, amelyben x benne van.
- `powerSet`-nek már csak megfelelő módon hívnia kell `pws`-t:

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

Mohó kiértékelés: faktoriális kiszámítása rekurzióval

- A faktoriális matematikai definíciója és megvalósítása SML-ben

$$\text{fac } 0 = 1; \text{ fac } n = n * \text{fac } (n - 1), n > 0$$

```
(* 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 (egy-egy triviális lépéseket elhagyunk).
 $\text{fac } 4 \rightarrow 4 * \text{fac } (4-1) \rightarrow 4 * \text{fac } 3 \rightarrow 4 * (3 * \text{fac } (3-1)) \rightarrow$
 $\rightarrow 4 * (3 * \text{fac } (2)) \rightarrow \dots \rightarrow 4 * (3 * (2 * (1 * 1))) \rightarrow \dots \rightarrow 24$
- 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 klózek 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          faci mint fac lokális függvénye:
   fac n = n!              fun fac n =
   PRE n >= 0              let fun faci 0 p = p
*)                          | faci n p = faci (n-1) (n*p)
fun fac n = faci n 1        in
                             faci n 1
                             end
```

ÖSSZETETT ADATTÍPUSOK

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 jobbrekurziót *terminális rekurzió*nak is nevezik (angolul: *tail* vagy *terminal recursion*).
- A jó fordítóprogram felismeri a jobbrekurziót, és hatékony tárgykódot állít elő: az argumentumokat frissíthető lokális változóknak tárolja, a rekurziót iterációval helyettesíti.
- A jobbrekurzív függvények feldolgozása tehát *iteratív*vá tehető.
- A jobbrekurzív függvényeket ezért *iteratív* függvényeknek is nevezik.

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 édesítőszers. Pl.


```
(2, 1.0) ≡ {1 = 2, 2 = 1.0} ≡ {2 = 1.0, 1 = 2} ≠ {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 egészségszám-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) ≡ {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.

Példa: Fibonacci-számok

- Ennes lehet függvény argumentuma és eredménye, összetett adat eleme stb.
- A Fibonacci-számok definíciója: $F'_0 = 0; F'_1 = 1; F'_n = F'_{n-2} + F'_{n-1}, n > 1$.
- Naív megvalósítása:

```
fun fib 0 = 0 | fib 1 = 1 | fib n = fib(n-2) + fib(n-1)
```

- Megvalósítása iterációval:

```
local
  (* iterfib(n, (prev, curr)) = a (prev, curr) Fibonacci-szám párt 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))
in
  (* fib n = az n-edik Fibonacci-szám
     fib : int -> int
  *)
  fun fib 0 = 0
    | fib n = iterfib(n, (0, 1))
end
```

HALMAZMŰVELETEK

Halmazműveletek: hatványhalmaz hatékonyabban

- pws rossz hatékonyságú, mert kétféle ágazó rekurziót használ. Pl. egy 19 egész számból álló lista hatványhalmazának előállítását nem lehet kivárni. Írjunk hatékonyabb változatot.
- Az insAll segédfüggvény egy elemet szűr be egy listából álló lista minden eleme elé.

```
(* insAll : 'a * 'a list list * 'a list list -> 'a list list
   insAll(x, yss, zss) = az yss lista ys elemeinek zss elé fűzött
                       listája, amelyben minden ys elem elé x van beszúrva *)
fun insAll (x, [], zss) = zss
  | insAll (x, ys::yss, zss) = insAll(x, yss, x::ys::zss)
```

- powerSet insAll-t használó rekurzív változata

```
fun powerSet [] = [[]]
  | powerSet (x::xs) = let val pws = powerSet xs
                       in pws @ insAll(x, pws, [])
                       end
```

- powerSet insAll-t használó iteratív változata

```
fun powerSet [] = [[]]
  | powerSet (x::xs) = let val pws = powerSet xs
                       in insAll(x, pws, pws)
                       end
```

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 0WxXXXX

- Füzerállandó: 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 füzerállandó.

Példák: # "a" # "\n" # "\^Z" # "\255" # "\""
 Ellenpéldák: # "a" # c # ""

SML-szintaxis: név

- Alfabetikus: kis- és nagybetűk, számjegyek, percjelek (') és aláhúzás-jelek (_) olyan sorozata, amely betűvel vagy perccel 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 : :: :> _ | => -> #
```

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).
 \uxxxx Az xxxx kódú karakter (x hexadecimális számjegy).
 \" Idézőjel (").
 \\ Hátrátört-vonal (\).
 \f...f\ Figyelmen kívül hagyott sorozat. f...f nulla vagy több formázókaraktert (szóköz, HT, LF, VT, FF, CR) jelent.

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	értékkadás	
	o	('b -> 'c) * ('a -> 'b) -> ('a -> 'c)	a két függvény kompozíciója	
0	before	'a * 'b -> 'a	a bal oldali argumentum	

LISTÁK

Listák 5-67

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.

Listák összefűzése (append) és megfordítása (nrev)

- Két lista összefűzése (append, infix változatban @)

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

Az xs-t először az elemeire bontjuk, majd hátulról visszafelé haladva fűzzük az elemeket az ys-hez, ugyanis a listákat csak előlről 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)

$$\text{nrev}[x_1, x_2, \dots, x_m] = \text{nrev}[x_2, \dots, x_m] @ [x_1] = \text{nrev}[\dots, x_m] @ [x_2] @ [x_1] = \dots = [x_m, \dots, x_1]$$

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]
```

Deklaratív programozás. BME VIK, 2002. tavaszi félév

Funkcionális programozás. 5. előadás

Listák 5-68

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 *párna 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

$$\text{foldr op}\oplus e [x_1, x_2, \dots, x_n] = (x_1 \oplus (x_2 \oplus \dots \oplus (x_n \oplus e) \dots))$$

$$\text{foldr op}\oplus e [] = e$$

$$\text{foldl op}\oplus e [x_1, x_2, \dots, x_n] = (x_n \oplus \dots \oplus (x_2 \oplus (x_1 \oplus e)) \dots)$$

$$\text{foldl op}\oplus e [] = e$$
- Asszociatív műveleteknél foldr és foldl eredménye azonos.

Deklaratív programozás. BME VIK, 2002. tavaszi félév

Funkcionális programozás. 5. előadás

Deklaratív programozás. BME VIK, 2002. tavaszi félév

Funkcionális programozás. 5. előadás

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 valólista 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-lel vagy foldr-rel. Kétooperandusú műveletként olyan segédfüggvényt (inc) alkalmazunk, amelyik *nem használja* az első paraméterét.

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

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

Lista: foldr és foldl definíciója

- $\text{foldr } \text{op} \oplus e [x_1, x_2, \dots, x_n] = (x_1 \oplus (x_2 \oplus \dots \oplus (x_n \oplus e) \dots))$
 $\text{foldr } \text{op} \oplus e [] = e$
 (* foldr f e xs = az xs elemeire jobbról balra haladva alkalmazott, kétooperandusú, 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;
- $\text{foldl } \text{op} \oplus e [x_1, x_2, \dots, x_n] = (x_n \oplus \dots \oplus (x_2 \oplus (x_1 \oplus e)) \dots)$
 $\text{foldl } \text{op} \oplus e [] = e$
 (* foldl f e xs = az xs elemeire balról jobbra haladva alkalmazott, kétooperandusú, 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;

Újabb példák foldr és foldl alkalmazására

- Egy lista elemeit egy másik lista elé fűzi foldr és foldl, ha kétooperandusú 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 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.
 $\text{foldr } \text{op} \oplus e [x_1, x_2, \dots, x_n] = (x_1 \oplus (x_2 \oplus \dots \oplus (x_n \oplus e) \dots))$
 $\text{foldl } \text{op} \oplus e [x_1, x_2, \dots, x_n] = (x_n \oplus \dots \oplus (x_2 \oplus (x_1 \oplus e)) \dots)$
- Nevezük foldL-nek a listában *balról jobbra* haladó, alábbi specifikációjú függvényt. Vegyük észre, hogy \oplus *bal oldali* egységelemet vár.
 $\text{foldL } \text{op} \oplus e [x_1, x_2, \dots, x_n] =$
 $(\dots ((e \oplus x_1) \oplus x_2) \oplus \dots \oplus x_n)$
- foldL olyan kétargumentumú függvényt vár, amelynek az „egységelem” (valójában: a részeredmény) az *első* argumentuma: $f : 'a * 'b -> 'a$.
 $\text{foldL } f e [x_1, x_2, \dots, x_n] =$
 (* foldL : ('a * 'b -> 'a) -> 'a -> 'b list -> 'a
 foldL f e xs = az xs elemeire balról jobbra haladva alkalmazott, kétooperandusú, 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 *kisebbitendő*, 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 *kisebbitendő*, 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);
```

ABSZTRAKCIÓ, ADATTÍPUSOK

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.

Gyenge és erős absztrakció, adattípusok

- Gyenge absztrakció: a név szinonima, az adatszerkezet részei továbbra is hozzáférhetőek.
- 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.

RACIONÁLIS SZÁMOK

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 egészsből *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}
```

Példa: racionális számok

- A racionális számokat *rekordként* ábrázolhatjuk; 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 – 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 elemeken 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)
```

Deklaratív programozás. BME VIK, 2002. tavaszi félév

(funkcionális programozás)6. előadás

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
  gcd m = m legnagyobb közös osztója 120-szal
*)
val gcd120 = gcd 120;

gcd120 45;
gcd120 48;
gcd120 ~96;
gcd120 630;
```

Deklaratív programozás. BME VIK, 2002. tavaszi félév

(funkcionális programozás)6. előadás

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)
```

Deklaratív programozás. BME VIK, 2002. tavaszi félév

(funkcionális programozás)6. előadás

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: *konstruktor*) 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 (főnemes) 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 nevet 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 felsorolása 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:

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

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 felsorolós 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 *felsorolós típus* létrehozni a datatype deklarációval. Pl.

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

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

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

A felsorolós 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 *típuskonstruktor* 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:

- a kétargumentumú `disun` típusoperátort,
- az `In1` : `'a -> ('a, 'b) disun` és
- 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.)

- 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 lekötöni.

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
```

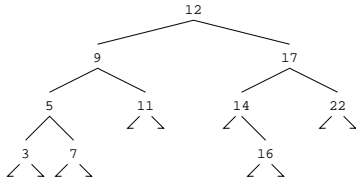
BINÁRIS FÁK

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:

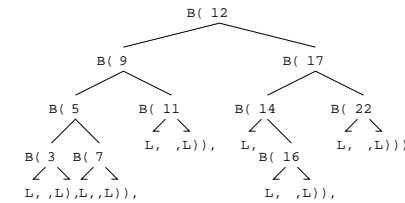


- Az 'a tree' adattípus L és B adatkonstruktoraival ez a fa pl. a következő lapon látható módon írható le.

Bináris fák datatype deklarációval (folyt.)

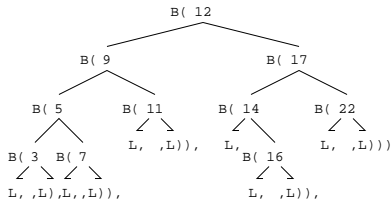
```
B(B(B(B(L, 3, L),
      5,
      B(L, 7, L)
    ),
    9,
    B(L, 11, L)
  ),
  12,
  B(B(L,
      14,
      B(L, 16, L)
    ),
    17,
    B(L, 22, L)
  )
)
```

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 átláthatóbb, 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.
 - ◊ kezdhethetjük 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 ágaknak (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
```