

Sztatikus és dinamikus kötés, mohó és lusta kiértékelés

KIFEJJEZÉSEK KIÉRTÉKELÉSE

A mohó és a lusta kiértékelés összevetése

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

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

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

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

Sztatikus és dinamikus kötés, mohó és lusta kiértékelés

- *Sztatikus kötés*: a formális paraméter összes előfordulását *fordítási időben* helyettesítjük az argumentummal (aktuális paraméterrel) a függvény (eljárás) törzsében.
- *Dinamikus kötés*: a formális paraméter összes előfordulását *futási időben* helyettesítjük az argumentummal (aktuális paraméterrel) a függvény (eljárás) törzsében.
- Kérdés, hogy az aktuális paraméterként átadott kifejezést az értelmező mikor értékeli ki: a behelyettesítés *előtt* vagy *után*.
- *Mohó kiértékelés*: a behelyettesítés *előtt* kiértékeljük az *összes* argumentumot (más megnevezések: *érték szerinti* paraméterátadás, *eager evaluation*, *call-by-value*).
- *Lusta kiértékelés*: a behelyettesítés *után* csak azt az argumentumot értékeljük ki, amelyikre szükség van, és csak akkor, amikor *szükség van* rá (más megnevezések: *szükség szerinti* paraméterátadás, *lazy evaluation*, *call-by-need*.)

Mohó kiértékelés

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

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

- Nézzük `sq(sq 2)` egyszerűsítését! (Az egyszerűsítés eredménye tovább már nem egyszerűsíthető, ún. *kanonikus* kifejezés.)
sq három alkalmazásából csak a harmadiknak kanonikus kifejezés az argumentuma.

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

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

Mohó kiértékelés mellett a számítógépet feleslegesen dolgoztatjuk!

Név szerinti paraméterátadás

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

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

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

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

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

Lusta kiértékelés

- Nézünk pl. sq(sq(2)) lusta kiértékelését!

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

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

```
sq(sq(sq(2))) → x * x [x = sq(sq(2))] → x * x [x = y * y] [y = sq(2)] →
x * x [x = y * y] [y = 2 * 2] → x * x [x = y * y] [y = 4] →
x * x [x = 4 * 4] → x * x [x = 16] → 16 * 16 → 256
```

- Gyakran nyerünk, de néha veszítünk a lusta kiértékeléssel.

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

```
fací(4, 1) → fací(4-1, 4*1) → fací(3-1, 3*(4*1)) →
fací(2-1, 2*(3*(4*1))) ... → 24
```

Lusta kiértékelés

- Lusta kiértékelés* esetén minden argumentumot csak egyszer kell kiértékelni: akkor, amikor *először* van rá szükség. Az argumentum összes előfordulását egy *rejtett hivatkozással* helyettesítjük (mivel *el van rejtve* a programozó elől, biztonságos): amikor a számítógép az argumentumot először kiértékeli, a kapott értéket elrakja, és később az összes olyan helyen, ahol szükség lesz rá, felhasználja.

- A függvényeket és argumentumaikat *irányított gráffal* ábrázolják: a gráf egy részének kiértékelésekor a gráfot az eredményül kapott értékkel frissítik a számítógépben (ezért nevezik *gráfredakciónak*).

A lusta kiértékeléshez bizonyult nyilvántartást kell vezetni (időigényes!).

- A lusta kiértékelés működési elvének megértéséhez irányított gráf helyett most $x = [E]$ -vel jelöljük, hogy az x összes előfordulása osztozik az E értékén.

ÖSSZETETT ADATTÍPUSOK

Emmes és típusa

- Két különböző típusú értékből rekordot vagy párt képezhetünk. Pl.
 $\{x = 2, y = 1.0\} : \{x : \text{int}, y : \text{real}\}$ és $(2, 1.0) : (\text{int} * \text{real})$.
- A pár is csak szintaktikai édesítőszerszer. Pl.
 $(2, 1.0) = \{f_1 = 2, 2 = 1.0\} = \{f_2 = 1.0, 1 = 2\}$, de $(2, 1.0)$ és $\{f_1 = 1.0, 2 = 2\}$ különböző típusúak. Az 1 és a 2 *mezőnevek* (vö. szintaxis).
- Rekordot kétfőnél több értékből is összeállíthatunk. Pl.
 $\{nev = \text{"Bea"}, tel = 3192144, kor = 19\} : \{kor : \text{int}, nev : \text{string}, tel : \text{int}\}$.
 Egy hasonló rekord egészszám-mezőnevekkel:
 $\{f_1 = \text{"Bea"}, 3 = 3192144, 2 = 19\} : \{f_1 : \text{string}, 2 : \text{int}, 3 : \text{int}$.
 Az *utóbbi azonos az alábbi ennessel* (n-es, n-tuple):
 $(\text{"Bea"}, 19, 3192144) : (\text{string} * \text{int} * \text{int})$,
 azaz $(\text{string} * \text{int} * \text{int}) \equiv \{f_1 = \text{string}, 2 = \text{int}, 3 = \text{int}\}$.
- Egy rekordban a tagok sorrendje közömbös, az értékeket a mezőnév azonosítja.

Deklaratív programozás, BMIE, 2001 tavaszi félév

14. előadás (funkcionális programozás)

Emmes és típusa (folyt.)

- Egy ennessen a tagok sorrendje meghatározó! Pl $(2, 1.0) : (\text{int} * \text{real})$, de $(1.0, 2) : (\text{real} * \text{int})$. A két emmes különböző!
- Emmes lehet függvény argumentuma és eredménye, összetett adat eleme stb.
 Példa: Fibonacci-számok iterációval.
 A definíció: $F_0 = 0; F_1 = 1; F_n = F_{n-2} + F_{n-1}, n > 1$.
 $(* \text{iterfib}(n, \text{prev}, \text{curr})) = a$ (prev, curr) Fibonacci-számpárt követő n-edik Fibonacci-szám (n > 0)
 $\text{iterfib} : \text{int} * (\text{int} * \text{int}) \rightarrow \text{int} *$
 $\text{fun iterfib } (1, (\text{prev}, \text{curr})) = \text{curr}$
 $\quad | \text{iterfib } (n, (\text{prev}, \text{curr})) = \text{iterfib}(n - 1, (\text{curr}, \text{prev} + \text{curr}));$
 $(* \text{fib } n = \text{az } n\text{-edik Fibonacci-szám}$
 $\text{fib} : \text{int} \rightarrow \text{int} *)$
 $\text{fun fib } 0 = 0$
 $\quad | \text{fib } n = \text{iterfib}(n, (0, 1));$

Deklaratív programozás, BMIE, 2001 tavaszi félév

14. előadás (funkcionális programozás)

FELHASZNÁLÓI ADATTÍPUSOK

Felhasználói adattípusok 14-12

A datatype deklaráció

- person néven új összetett típust hozunk létre:
 $\text{datatype person} = \text{King}$
 $\quad | \text{Peer of string} * \text{string} * \text{int}$
 $\quad | \text{Knight of string}$
 $\quad | \text{Peasant of string};$
- Az új típusnak négy *adatkonstruktor* (röviden: *konstruktor*) van: King, Peer, Knight és Peasant.
- King ún. *adatkonstruktorállandó*, a többi ún. *adatkonstruktorfüggvény*.
- Az adatkonstruktoroknak is van típusuk:
 $\text{King} : \text{person}$
 $\text{Peer} : \text{string} * \text{string} * \text{int} \rightarrow \text{person}$
 $\text{Knight} : \text{string} \rightarrow \text{person}$
 $\text{Peasant} : \text{string} \rightarrow \text{person}$

Deklaratív programozás, BMIE, 2001 tavaszi félév

14. előadás (funkcionális programozás)

A datatype deklaráció (folyt.)

- King (király) csak egy van, ezért definiálhatunk 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.

Deklaratív programozás, BMFE, 2001 tavaszi félév

14. előadás (funkcionális programozás)

Felhasználói adattípusok 14-15

A datatype deklaráció (folyt.)

- Ha más lenne a változatosok sorrendje, a `_: :ps` minta nemcsak a `King-re`, a `Peer-re` és a `Peasant-ra` illeszkedne (í.e. 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 Vs.p≠knight s.
```

Deklaratív programozás, BMFE, 2001 tavaszi félév

14. előadás (funkcionális programozás)

A datatype deklaráció (folyt.)

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


```
(* 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 `knights` nevét összegyűjti a `person` típusú személyek egy listájából (a változatosok 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;
```

Deklaratív programozás, BMFE, 2001 tavaszi félév

14. előadás (funkcionális programozás)

Felhasználói adattípusok 14-16

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;
```

Deklaratív programozás, BMFE, 2001 tavaszi félév

14. előadás (funkcionális programozás)

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;
```

Polimorf adattípusok

- Látnuk, hogy a list *posztfix pozíciójú típusoperátor*, nem típus: a datatype deklaráció az adatkonstruktorok mellett *típuskonstruktor*t is létrehoz.

- A belső 'a list típushoz hasonló 'a list listát és vele együtt a Nil és a Cons *adatokonstruktorokat* például így definiálhatjuk:

```
datatype 'a list = Nil | Cons of 'a * 'a list;
```

- A Cons *adatokonstruktorfüggvény* 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 ::: *adatokonstruktoroperátort*:

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

- A *hatospontot* közvetlenül a típusdeklarációban is definiálhatjuk:

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

A felsorolásos típus datatype deklarációval (folyt.)

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

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

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

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

Polimorf adattípusok: megkülönböztetett egyesítés

- Következő példánk két típus *megkülönböztetett egyesítése*, más néven diszjunkt uniója:

```
datatype ('a, 'b) disun = In1 of 'a | In2 of 'b;
```

- Itt három dolgot definiáltunk:

1. a kétargumentumú disun típusoperátort,
2. az In1 : 'a -> ('a, 'b) disun és
3. az In2 : 'b -> ('a, 'b) disun adatkonstruktorfüggvényeket.

- ('a, 'b) disun az 'a és 'b típusok megkülönböztetett egyesítése.

Megkülönböztetettnek nevezzük az egyesítést, mert később is bármikor meg tudjuk mondani, hogy egy ('a, 'b) disun típusú pár egyik vagy másik eleme melyik alaptípusból származik. Az új típusba tartozó értékek In1 x alakúak, ha x 'a típusú, és In2 y alakúak, ha y 'b típusú.

- Az In1 és In2 konstruktorfüggvények olyan *címkének* tekinthetők, amelyek az 'a típust megkülönböztetik a 'b típustól.

Megkülönböztetett egyesítés (folyt.)

- A megkülönböztetett egyesítés lehetővé teszi, hogy különböző típusokat használjunk ott, ahol egyébként csak egyetlen típust használhatnánk (vö. objektum-orientált programozás, ahol pl. egy *alakzat osztálynak téglalap, háromszög vagy kör* nevű leszármazottai lehetnek).
- Az SML-ben megkülönböztetett egyesítéssel tudunk létrehozni *különböző típusú elemekből* álló listát:

```
[In2 King, In1 "Skócia"] : ((string, person) disun) list
[In1 "zsarnok", In2 1040] : ((string, int) disun) list
```

- A lehetséges eseteket most is *mintaillesztéssel* elemezhetjük, pl.

```
(* concat d = a d diszjunkt unió In1 címkejű elemeknek konkatenációja
   concat : (string, 'a) disun list -> string *)
fun concat [] = ""
  | concat (In1 s :: ls) = s ^ concat ls
  | concat (In2 _ :: ls) = concat ls;
```

Megkülönböztetett egyesítés (folyt.)

- Egy példa `concat` alkalmazására:
 - `concat [In1 "Űj", In2 King, In1 "Skócia"];`
 - `> val it = "Űj Skócia : string"`
- Az `In1` konstruktorfüggvény típusa `'a -> ('a, 'b) disun`, ezért a `string` típusú `"Űj"` 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 "Űj", In2 King, In1 "Skócia"]` kifejezésben mindkét alap típust lekötjük, ezért ennek a listának a típusa: `((string, person) disun) list`.

- Az `[In2 "Űj", In2 King, In1 "Skócia"]` kifejezés kiértékelése hibajelzést eredményez, mert a `'b` típusváltozót nem lehet ugyanabban a kifejezésben egyszer `figy`, mászor `ügy` lekötöni.

BINÁRIS FÁK

Bináris fák datatype deklarációval

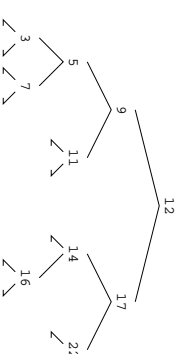
Bináris fák

14-24

- A listához hasonlóan rekurzív adatszerkezetet 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;
```

- Tekintjük például az alábbi fát:

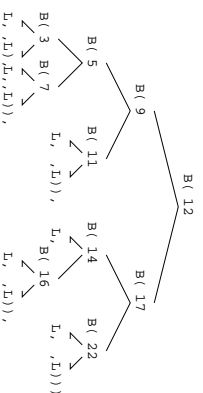


- Az `'a tree` adattípus `L` és `B` adatkonstruktoraival ez a *fa* pl. a következő lapon látható módon írható le.

Bináris fák datatype deklarációval (folyt.)

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

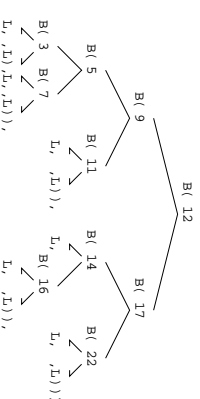
A bal oldali kifejezést elég nehéz átlátni. A faststruktúra szöveges leírását megkönnyíti, ha az ábrába beírjuk a megfelelő adatkonstruktorokat.



$)$;

Bináris fák datatype deklarációval (folyt.)

• A faststruktúra szöveges leírása átláthatóbb, ha az egyes részfákknak 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 faststruktúrákat is deklarálhatunk, pl.
 - kezdhetjük az 'a típusú értékek, majd folytathatjuk előbb a bal, azután a jobb részfa megadásával,
 - felhasználhatjuk a levelet is értékek tárolására,
 - az értéket nem tároló üres csomókat 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 ágknak (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 fákban

- 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:


```
fun nodes 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 lefeléhez vezető úton az élék számát (az út hosszát) az adott levél szintjének is nevezzük. A szintek közül a legnagyobbbat 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:

```
fun depth 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;
```