

KIFEJZÉSEK KIÉRTÉKELÉSE



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*.)

A mohó és a lusta kiértékelés összevetése

- Más paraméteradási eljárások
 - *név 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.

Mohó kiértékelés

- Emlékeztető: az f e 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(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 előtt sokszor nemcsak fölösleges, hanem káros is előre kiszámítani az argumentumokat, mert végtelen rekurzió vagy illegális művelet (indexhatár-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 *megkét-szerezi* 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

- *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 *gráfredukciónak*).

A lusta kiértékeléshez bonyolult 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 értéken.

Lusta kiértékelés

- Nézzük pl. `sq(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 veszünk a lusta kiértékeléssel.

Láttuk, hogy `fun fac i(0, p) = p | fac i(n, p) = fac i(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:

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

ÖSSZEGETT ADATTÍPUSOK



Ennes és típusa

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

Ennes és típusa (folyt.)

- Egy ennesben a tagok sorrendje meghatározó! Pl. (2, 1.0) : (int * real), de (1.0, 2) : (real * int). A két ennes különböző!

- Ennes lehet függvény argumentuma és eredménye, összetett adat eleme stb.
Példa: Fibonacci-számok iterációval.

A definíció: $F_0 = 0; F_1 = 1; F_n = F_{n-2} + F_{n-1}, n > 1$.

```
(* iterfib(n, (prev, curr)) = a (prev, curr) Fibonacci-számpárt 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));

(* fib n = az n-edik Fibonacci-szám
   fib : int -> int *)
fun fib 0 = 0
  | fib n = iterfib(n, (0, 1));
```

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 *adatkonstruktor*a (röviden: *konstruktor*a) 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 (király) csak egy van, ezért definiálhattuk konstruktorállandóként.
- A Peer-t (főnembst) nemesi címe (string), birtokának neve (string) és sorszámja (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 nevét 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` mint a 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, végrehajítását is.
- A bizonyítás nem okoz gondot, ha a függvény harmadik sorát (`sirs (_::ps) = sirs ps`) *feltételes egyetlennek* 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 felsorolásos típus datatype deklarációval

- Gyakori, hogy egy név csak néhány különböző értéket vehet fel (azaz a név által felvehető értékek halmaza kis számosságú), ilyen esetben érdemes *felsorolásos típust* létrehozni a datatype deklarációval. Pl.

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

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

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

A felsorolásos típus datatype deklarációval (folyt.)

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

```
(* Lady 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 deklaráálhatnánk, 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:
 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 *alakszat* osztálynak *téglalap*, *háromszög* vagy *kör* nevű leszármozottai 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;
```

Megkülönböztetett egyesítés (folyt.)

- Egy példa `concat` alkalmazására:
 - `concat [In1 "Ű!", In2 King, In1 "Skócia"];`
> `val it = "Ű! Skócia : string`
- Az `In1` konstruktorfüggvény típusa `'a -> ('a, 'b) disun`, ezért a `string` típusú `"Ű!"` 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 "Ű!", In2 King, In1 "Skócia"]` kifejezésben mindkét alaptípust lekötjük, ezért ennek a listának a típusa: `((string, person) disun) list`.
- Az `[In2 "Ű", In2 King, In1 "Skócia"]` kifejezés kiértékelése hibajelzést eredményez, mert a `'b` típusváltozót nem lehet ugyanabban a kifejezésben egyszer így, másszor úgy lekötni.

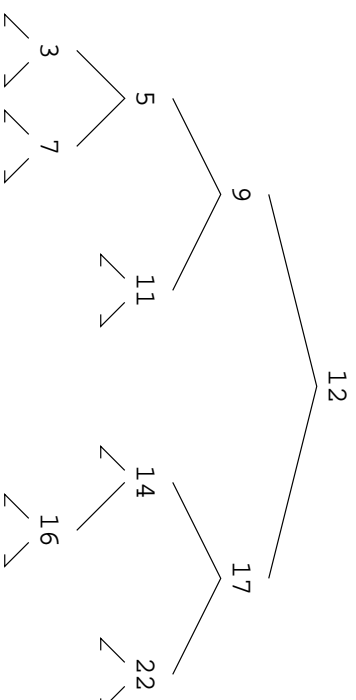
BINÁRIS FÁK



Bináris fák datatype deklarációval

- 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;
```
- Tekintsük például az alábbi fát:



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

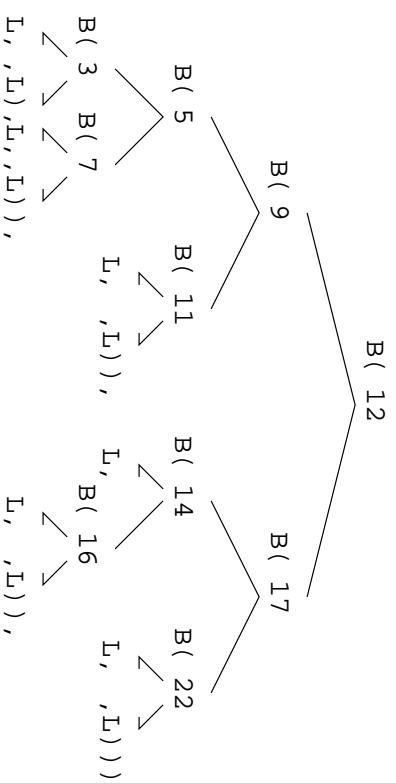
Bináris fák datatype deklarációval (folyt.)

```

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)
)
);

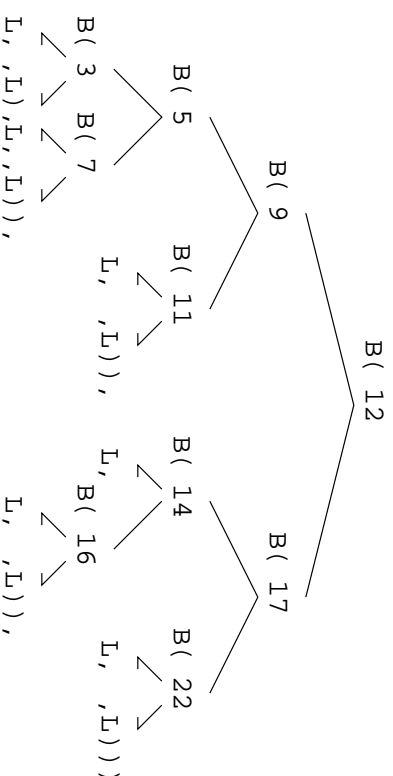
```

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 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 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 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 ágoknak (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 fákön

- 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 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:

```
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;
```