Deklaratív programozás

Hanák Péter
hanak@inf.bme.hu

Irányítástechnika és Informatika Tanszék

(OM Kutatás-fejlesztési Helyettes Államtitkárság)

Szeredi Péter
szeredi@iqsoft.hu

Számítástudományi és Információelméleti Tanszék

(IQSOFT Intelligens Software Rt.)

KÖVETELMÉNYEK, TUDNIVALÓK

Deklaratív programozás: tudnivalók

Honlap, levelezési lista

- Honlap: <http://dp.iit.bme.hu/~dp>
- Levlista: <http://www.iit.bme.hu/mailman/listinfo/dp-l>.
  A listatagoknak szóló levelet a <dp-l@www.iit.bme.hu> címre kell küldeni.
  Csak a feliratkozottak levele jut el moderátori jóváhagyás nélkül a listatagokhoz.

Jegyzet

- Szeredi Péter, Benkő Tamás: Deklaratív programozás. Bevezetés a logikai programozásba
- Hanák D. Péter: Deklaratív programozás. Bevezetés a funkcionális programozásba
- Ára kötetenként 600-800 Ft, terjedelemtől függően
- Elektronikus változata elérhető a honlapról (ps, pdf)
- Jegyzetrendelés: az ETS (Elektronikus TanárSegéd) rendszeren keresztül

Deklaratív programozás: tudnivalók (folyt.)

Fordító- és értelmezőprogramok

- SICStus Prolog (3.10, licenszköteles, aláírás ellenében jelszót adunk)
- Moscow SML (2.0, szabad szoftver)
- Mindkettő telepítve van a kempelen.inf.bme.hu-n
- Mindkettő letölthető a honlapról (linux, Win95/98/NT)
- Webes gyakorló felület az ETS-ben (ld. honlap)
- Kézikönyvek HTML-, ill. PDF-változatban
- Más programok: swiProlog, gnuProlog, poly/ML, smlnj
- emacs-szövegszerkesztő SML-, ill. Prolog-módban (linux, Win95/98/NT)

Deklaratív programozás: félévközi követelmények

Nagy házi feladat (NHF)

- Programozás mindkét nyelven (Prolog, SML)
- Mindenkinek önállóan kell kódolnia (programoznia)!
- Hatékony (időlimit!), jól dokumentált („kommentezett”) programok
- A két programhoz közös, 5–10 oldalas fejlesztői dokumentáció (TXT, TeX/LaTeX, HTML, PDF, PS; de nem DOC vagy RTF)
- Kiadás a 6. héten, a honlapon, letölthető keretprogrammal
- Beadás a 12. héten; elektronikus úton (ld. honlap)
- A beadáskor és a pontozáskor külön-külön tesztsorozatot használunk (nehézségben hasonlókat, de nem azonosakat)
- A minden tesztesetet hibátlanul megoldó programok *létraversenyen* vesznek részt (hatékonyság, gyorsaság plusz pontokért)

Deklaratív programozás: félévközi követelmények (folyt.)

Nagy házi feladat (folyt.)

- Nem kötelező, de *nagyon* ajánlott!
- Beadható csak az egyik nyelvből is
- A beadási határidőig többször is beadható, csak az utolsót értékeljük
- Pontozása mindkét nyelvből:
  - helyes és időkorláton belüli futás esetén a 10 teszteset mindegyikére 0,5-0,5 pont, összesen max. 5 pont, feltéve, hogy legalább 4 teszteset sikeres
  - a dokumentációra, a kód olvashatóságára, kommentezettségére max. 2,5 pont
  - tehát nyelvenként összesen max. 7,5 pont szerezhető
- A NHF súlya az osztályzatban: 15% (a 100 pontból 15)

Deklaratív programozás: félévközi követelmények (folyt.)

Kis házi feladatok (KHF)

- 2-3 feladat Prologból is, SML-ből is
- Beadás elektronikus úton (ld. honlap)
- Nem kötelező, de *nagyon* ajánlott
- Minden feladat jó megoldásáért 1-1 jutalompont jár

Gyakorló feladatok

- Nem kötelező, de a sikeres ZH-hoz, vizsgához *elengedhetetlen*!
- Gyakorlás az ETS rendszerben (lásd honlap)

Deklaratív programozás: félévközi követelmények (folyt.)

Nagyzárthelyi, pótzárthelyi (NZH, PZH, PPZH)

- A zárthelyi kötelező!
- Semmilyen jegyzet, segédlet nem használható
- A megtanulandó könyvtári függvények, ill. eljárások listáját előre megadjuk
- 40%-os szabály (nyelvenként a maximális részpontszám 40%-a kell az eredményességhez). Kivétel: a korábban aláírást szerzett hallgató zárthelyin szerzett pontszámát az alsó ponthatártól függetlenül beszámítjuk a félévvégi osztályzatba.
- Az NZH a 7., a PZH az utolsó oktatási hetekben lesz
- A PPZH-ra indokolt esetben, ismétlővizsga-jelleggel a vizsgaidőszak első három hetében egyetlen alkalommal adunk lehetőséget
- Az NZH anyaga az 1.-6. hét tananyaga
- A PZH, ill. a PPZH anyaga azonos az NZH anyagával
- A zárthelyi súlya az osztályzatban: 15% (a 100 pontból 15)
- Több zárthelyi megírása esetén a zárthelyikre kapott pontszámok közül a *legnagyobbat* vesszük figyelembe

Deklaratív programozás: vizsga

Vizsga

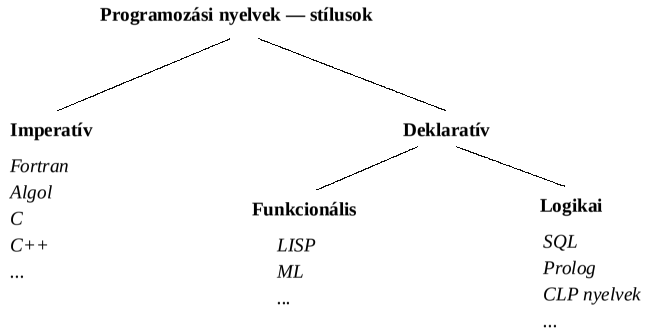
- Vizsgára az a hallgató bocsátható, aki aláírást szerzett a jelen félévben vagy a jelen félévet megelőző négy félévben
- A vizsga szóbeli, felkészülés írásban
- Prolog, SML: több kisebb feladat (programírás, -elemzés) kétszer 35 pontért
- A vizsgán szerezhető max. 70 ponthoz adjuk hozzá a **jelen** félévben félévközi munkával szerzett pontokat: ZH: max. 15 pont, NHF: max. 15 pont, továbbá a pluszpontokat (KHF, létraverseny)
- *Korábbi* félévben szerzett pontokat *nem* számítunk be!
- A vizsgán semmilyen jegyzet, segédlet nem használható, de lehet segítséget kérni
- A megtanulandó könyvtári függvények, ill. eljárások listáját előre megadjuk
- Ellenőrizzük a nagy házi feladat és a zárthelyi „hitelességét”
- 40%-os szabály (nyelvenként a max. részpontszám 40%-a kell az eredményességhez)
- Korábbi vizsgakérdések a honlapon találhatók

A félév időbeosztása

- Az előadások közelítő menetrendje
  - 1. előadás: A deklaratív és imperatív programozás összehasonlítása
  - 2.-7. előadás (1.-4. hét): Prolog I. rész (a nyelv alapjai)
  - 8.-13. előadás (4.-7. hét): SML I. rész (a nyelv alapjai)
  - 14.-19. előadás (7.-10. hét): Prolog II. rész
  - 20.-25. előadás (10.-13. hét): SML II. rész
  - 26.-27. előadás (14. hét): Kitekintés
- A félévközi (nem kötelező és kötelező) számonkérések közelítő menetrendje
  - A KHF-ek kiadása: a 2., 4., 7. és 9. héten
  - A KHF-ek beadása: a kiadásuk után két héttel
  - Az NHF kiadása: 6. hét
  - Az NHF beadása: 12. hét
  - Az NZH megírása a 7. héten, hétfőn 16.15-től kb. 18.00-ig
  - A PZH megírása a 13. vagy 14. héten, megállapodás szerinti időben
  - A PPZH megírása (indokolt esetben, ismétlővizsga-jelleggel) a vizsgaidőszak első három hetében kijelölt egyetlen alkalommal lehetséges

DEKLARATÍV ÉS IMPERATÍV PROGRAMOZÁS

Programozási nyelvek osztályozása



Imperatív és deklaratív programozási nyelvek

- Imperatív program
  - felszólító módú, utasításokból áll
  - változó: változtatható értékű memóriahely

```
int fakt(int n)          // C nyelvű példa:
  { int f=1;             // Legyen f értéke 1!
    while (n>1)          // Amíg n>1 ismételd ezt:
      f*=n--;            //    szorozd f-et n-nel, majd csökkentsd n-et!
    return f;            // Add vissza f végértékét mint n faktoriálisát!
  }                      //(fakt(4) = 1*4*3*2)
```

- Deklaratív program
  - kijelentő módú, egyenletekből, állításokból áll
  - változó: egy ismeretlen, de (előbb–utóbb) rögzített értékű mennyiség.
  - SML példa:

```
fun fakt 0 = 1           (* 0 faktoriálisa 1 *)
  | fakt n = n *         (* n faktoriálisa egyenlő n szorozva *)
             fakt (n-1)  (* n-1 faktoriálisával *)
                         (fakt(4) = 4*(3*(2*1)))
```

  - C példa: `int fakt(int n) {if (n<=1) return 1; else return n*fakt(n-1);}`

Deklaratív programozási nyelvek

- Alapvető jellemzők
  - A deklaratív programok **dekomponálhatók**: a program felbontható részekre, amelyek egymástól **függetlenül** megírhatók, tesztelhetők, verifikálhatók.
  - A deklaratív programokon könnyű következtetéseket végezni, pl. helyességüket bizonyítani.
- Tulajdonságok
  - Egy nyelvi elem értelme csak önmagától függ — állapotmentesség.
  - Hivatkozási átlátszóság (referential transparency) — pl. ha $f(x) = x^2$, akkor $f(a)$ **helyettesíthető** a^2-tel.
  - Egyszeres értékadás (single assignment) — párhuzamos végrehajthatóság.
- Jelmondat
  - MIT és nem HOGYAN (WHAT rather than HOW): a *megoldás módja* helyett inkább a megoldandó *feladat leírását* kell megadni
  - A gyakorlatban mindkét szemponttal foglalkozni kell — kettős szemantika:
    - deklaratív szemantika — MIT (milyen feladatot) old meg a program;
    - procedurális szemantika — HOGYAN oldja meg a program a feladatot.

Példa — családi kapcsolatok

- Adatok

  Egy gyerek–szülő kapcsolat, pl.

gyerek	szülő
Imre	István
Imre	Gizella
István	Géza
István	Sarolta
Gizella	Civakodó Henrik
Gizella	Burgundi Gizella

- A feladat:

  Definiálandó az unoka–nagyszülő kapcsolat, pl. keressük egy adott személy nagyszüleit.

A nagyszülő feladat — C nyelvű megoldás

```
/* Az adatbázis */
struct gysz {
  char *gyerek, *szulo;
} szulok[] = {
  "Imre",    "István",
  "Imre",    "Gizella",
  "István",  "Géza",
  "István",  "Sarolt",
  "Gizella", "Civakodó Henrik",
  "Gizella", "Burgundi Gizella",
  NULL,      NULL
};
```

```
/*  unoka nagyszüleinek kiiratása */
void nagyszuloi(char *unoka)
{
  struct gysz *mgysz = szulok;
  for (; mgysz->gyerek; ++mgysz)
    if(!strcmp(unoka, mgysz->gyerek))
    { struct gysz *mszn = szulok;
      for (; mszn->gyerek; ++mszn)
        if (!strcmp(mgysz->szulo,
                    mszn->gyerek))
          puts(mszn->szulo);
    }
}
```

A nagyszülő feladat — SML megoldás

- Az SML program:

```
(* szulei x = az x személy szüleinek listája *)
fun szulei "Imre"    = ["István", "Gizella"]
  | szulei "István"  = ["Géza", "Sarolt"]
  | szulei "Gizella" = ["Civakodó Henrik", "Burgundi Gizella"]
  | szulei _         = []     (* senki másnak nincs szülője *)
> val szulei = fn : string -> string list

(* nagyszulei g = g nagyszüleinek listája*)
fun nagyszulei g = List.concat (map szulei (szulei g));
> val nagyszulei = fn : string -> string list
```

- A függvény futtatása

```
- nagyszulei "Imre";
> val it = ["Géza", "Sarolt", "Civakodó Henrik", "Burgundi Gizella"]
           : string list
```


A nagyszülő feladat — SQL megoldás

```
SQL> create table szulok (gyerek char(30), szulo char(30));
(...)

SQL> create view nagyszulok as select fiatal.gyerek, oreg.szulo
  ->    from szulok as fiatal, szulok as oreg
  ->    where fiatal.szulo = oreg.gyerek;
View created.

SQL> select * from nagyszulok;
GYEREK                         SZULO
------------------------------ ------------------------------
Imre                           Civakodó Henrik
Imre                           Burgundi Gizella
Imre                           Géza
Imre                           Sarolt

SQL>
```


A nagyszülő feladat — Prolog megoldás

```
% szuloje(Gy, Sz):Gy szülője Sz.
szuloje('Imre', 'István').
szuloje('Imre', 'Gizella').
szuloje('István', 'Géza').
szuloje('István', 'Sarolt').
szuloje('Gizella',
            'Civakodó Henrik').
szuloje('Gizella',
            'Burgundi Gizella').

%  Gyerek nagyszülője Nagyszulo.
nagyszuloje(Gyerek, Nagyszulo) :-
    szuloje(Gyerek, Szulo),
    szuloje(Szulo, Nagyszulo).
```

```
% Kik Imre nagyszülei?
| ?- nagyszuloje('Imre', NSz).
NSz = 'Géza' ? ;
NSz = 'Sarolt' ? ;
NSz = 'Civakodó Henrik' ? ;
NSz = 'Burgundi Gizella' ? ;
no
% Kik Géza unokái?
| ?- nagyszuloje(U, 'Géza').
U = 'Imre' ? ;
no
```


A deklaratív és imperatív megoldások összehasonlítása

- A keresési feladat megoldása
  - C nyelven: ciklussal
  - SQL-ben: beépített adatbázis-kereséssel
  - SML-ben: magasabbrendű függvénybe rejtett rekurzióval
  - Prologban: beépített mintaillesztéses eljáráshívással
- Az összetett feltételek kezelése
  - C nyelven: skatulyázott ciklussal
  - SML-ben: függvények kompozíciójával
  - SQL-ben, Prologban: relációk konjunkciójának képzésével
- A funkcionális és logikai megoldásokról
  - az SML megoldás rendkívül tömör (magasabbrendű függvények)
  - a Prolog megoldás többirányú (több függvénykapcsolatnak felel meg)

Egy összetettebb példa: bináris fák bejárása

- A bináris fa adatstruktúra
  - vagy egy csomópont (`node`), amelynek két részfája van (`left`, `right`)
  - vagy egy levél (`leaf`), amely egy egészt tartalmaz
- Binárisfa-struktúrák különböző nyelveken

```
% Struktúra deklarációk C-ben
enum treetype Node, Leaf;
struct tree {
  enum treetype type;
  union {
    struct { struct tree *left;
             struct tree *right;
           } node;
    struct { int value;
           } leaf;
  } u;
};
```

```
% Adattípus-deklaráció SML-ben

datatype Tree =
         Node of Tree * Tree
       | Leaf of int

% Adattípus-komment Prologban

% :- type tree --->
%        node(tree, tree)
%      | leaf(int).
```


Bináris fák összegzése

- Egy bináris fa levélösszegének kiszámítása:
  - csomópont esetén a két részfa levélösszegének összege
  - levél esetén a levélben tárolt egész
- Binárisfa-összegzők különböző nyelveken

```
% C nyelvű (deklaratív!) függvény
int sum_tree(struct tree *tree)
{
  switch(tree->type) {
  case Leaf:
   return tree->u.leaf.value;
  case Node:
   return
    sum_tree(tree->u.node.left) +
    sum_tree(tree->u.node.right);
   }
}
```

```
% SML nyelvű függvény
fun sum_tree( Node(Left,Right) )
         = sum_tree Left +
           sum_tree Right
  | sum_tree( Leaf(Val) ) = Val

% Prolog eljárás (predikátum)
sum_tree(leaf(Value), S) :-
        S = Value.
sum_tree(node(Left,Right), S) :-
        sum_tree(Left, S1),
        sum_tree(Right, S2),
        S is S1+S2.
```


Bináris fák összegzése — SML példafutás

```
% mosml
Moscow ML version 2.00 (June 2000)
Enter ‘quit();’ to quit.
- use "treeSum.sml";
[opening file "treeSum.sml"]
> New type names: =Tree
  datatype Tree =
  (Tree,con Leaf : int -> Tree, con Node : Tree * Tree -> Tree)
  con Leaf = fn : int -> Tree
  con Node = fn : Tree * Tree -> Tree
  val sum_tree = fn : Tree -> int
[closing file "treeSum.sml"]
> val it = () : unit
- sum_tree( Node(Leaf(5),
                 Node(Leaf(3),Leaf(2))) ) ;
> val it = 10 : int
- quit();
%
```


Bináris fák összegzése — Prolog példafutás

```
% sicstus -f
SICStus 3.10.0 (x86-linux-glibc2.1): Tue Dec 17 15:12:52 CET 2002
Licensed to BUTE DP course
| ?- consult(tree).
consulting /home/szeredi/peldak/tree.pl...
consulted /home/szeredi/peldak/tree.pl in module user, 0 msec 704 bytes
yes
| ?- sum_tree(node(leaf(5),
                   node(leaf(3), leaf(2))), Sum).
Sum = 10 ? ;
no
| ?- sum_tree(Tree, 10).
Tree = leaf(10) ? ;
! Instantiation error in argument 2 of is/2
! goal:  _76 is _73+_74
| ?- halt.
%
```

A funkcionális programozásról dióhéjban

- Alapeszme
  - a program elemei értékek, speciálisan függvények
    - egy függvény egy kiszámítási szabályt ad meg
    - a program futása: kiértékelés (egyszerűsítés, redukció)
- A funkcionális programozás első megvalósítása: LISP
  - alapötlet: listák könnyű/hatékony feldolgozása
- A funkcionális programozás egy modern megvalósítása: SML
  - a függvények „teljes jogú” értékek
  - erős típusfogalom, típusok automatikus levezetése

SML — előnyök és hátrányok

- Miért jó?
  - nagyon tömör kód
  - függvények is értékek: futási időben létrehozhatók
  - mintaillesztés: adatstruktúrák könnyen, áttekinthetően kezelhetők
  - erős típusrendszer
- Mik a hátrányai?
  - megszokottól eltérő programozói stílus
- Hogyan tovább?
  - lusta kiértékelés (Haskell, Clean)
  - párhuzamos végrehajtás (Parallel Haskell, CAML — Concurrent ML)
  - típusrendszer bővítése öröklődéssel (Haskell, Clean, Objective CAML)

A logikai programozásról dióhéjban

- Alapeszme
  - A program elemei logikai állításoknak felelnek meg, pl.:
    `nagyszuloje(U, N) :- szuloje(U, Sz), szuloje(Sz, N).`
    matematikai formája:
    $\forall U \forall N \forall Sz(\texttt{nagyszuloje}(U,N) \leftarrow \texttt{szuloje}(U,Sz) \wedge \texttt{szuloje}(Sz,N))$
  - A program futása: dedukció (tételbizonyítási folyamat)
- A logikai programozás első megvalósítása: a Prolog nyelv
  - A logikai állítások egyszerűek, tekinthetők eljárásdefiníciónak is
  - A tételbizonyítási folyamat értelmezhető mint:
    mintaillesztéses eljáráshívás + visszalépéses keresés
  - Prolog = RDBMS + rekurzió + adatstruktúrák

Prolog — előnyök és hátrányok

- Miért jó?
  - tömör kód, többirányú eljárások
  - „automatikus” visszalépéses keresés, ciklusok kiváltása
  - „logikai” változó — meghatározatlan adatok kezelése
- Mik a hátrányai?
  - nehéz megtanulni (különösen „tapasztalt” programozóknak)
  - rögzített, rugalmatlan vezérlési mechanizmus
  - gyenge következtetési képesség
- Hogyan tovább?
  - CLP — korlát logikai programozás (constraint logic programming)
  - annotációk, típusok — Mercury
  - rugalmasabb vezérlés, párhuzamos végrehajtás — Aurora, Andorra, Oz

Deklaratív programozás — miért tanítjuk?

- Új, magasszintű programozási elemek
  - rekurzió
  - mintaillesztés
  - visszalépéses keresés
- Új gondolkodási stílus
  - dekomponálható programok: a programrészek (relációk, függvények) önálló jelentéssel bírnak
  - verifikálható programok: a kód és a jelentés összevethető
- Új alkalmazási területek
  - szimbolikus alkalmazások
  - következtetési módszerekre épülő megoldások
  - nagyfokú megbízhatóságot igénylő rendszerek

Egy példa: párbeszéd egy 50 soros Prolog programmal

```
| ?- párbeszéd.
|: Magyar legény vagyok én.
Felfogtam.
|: Ki vagyok én?
Magyar legény
|: Péter kicsoda?
Nem tudom.
|: Péter tanuló.
Felfogtam.
|: Péter jó tanuló.
Felfogtam.
|: Péter kicsoda?
tanuló
jó tanuló
|: Boldog vagyok.
Felfogtam.
|: Te egy Prolog program vagy.
Felfogtam.
|: Ki vagyok én?
Magyar legény
Boldog
|: Okos vagy.
Felfogtam.
|: Te vagy a világ közepe.
Felfogtam.
|: Ki vagy  te?
egy Prolog program
Okos
a világ közepe
|: Valóban?
Nem értem.
|: Unlak.
Én is.
```


BEVEZETÉS A LOGIKAI PROGRAMOZÁSBA

Bevezetés a Logikai Programozásba

- Az előadássorozat áttekintése
  - Bevezetés
  - A Prolog nyelv alapjai
  - Prolog programozási módszerek
  - A legfontosabb beépített eljárások
  - Fejlettebb nyelvi és rendszerelemek
  - Új irányzatok a logikai programozásban

A Prolog nyelv — logikai alapú bemutatás

- **Prolog** = **Pro**gramozás **log**ikában (**Pro**gramming in **log**ic) alapelve:
  - egy program logikai állításokból áll, a program futása következtetési folyamat.
- A logika nyelvének elemei: (rövid összefoglaló, vö. a Matematikai Logika c. tárgy anyagával)
  - Kifejezés (term): változókból és konstansokból függvények segítségével épül fel, pl $f(a, g(X))$, ahol f kétargumentumú, g egyargumentumú függvénynév, a konstansnév (azaz 0-argumentumú függvénynév) és X változónév.
  - Elemi állítás (atom): egy relációjel, megfelelő számú argumentummal ellátva, ahol az argumentumok kifejezések, pl. $osztja(X, X * Y)$.
  - Prolog konvenciók:
    - A változóneveket nagybetűvel vagy aláhúzásjellel kezdjük.
    - Kétargumentumú függvénykifejezéseket, állításokat infix alakban is írhatunk, pl. $X + 2 * Y \equiv +(X, *(2, Y))$, $X < X * 2 \equiv <(X, *(X, 2))$
    - A függvények (és konstansok) nevét kisbetűvel kezdjük, vagy aposztrofok közé tesszük. Speciális jelek ill. jelsorozatok is megengedettek függvény, konstans, vagy állítás neveként (pl. $+, *, <$).
  - Állítás (formula): elemi állításokból logikai összekötő jelekkel (pl. $\wedge$, $\vee$, $\neg$, $\rightarrow$) és kvantorok ($\forall$, $\exists$) alkalmazásával épül fel, pl. $\forall X(X < 0 \rightarrow \neg X < X * 2)$.

A logika nyelvének megszorítása

- A következtetési folyamat hatékonyabbá tételéhez érdemes a logikai nyelvet szűkíteni.
- Bevezetjük a klóz (clause) fogalmát. Egy klóz az alábbi alakú logikai állítás:

$$\forall X_1 \ldots X_j((F_1 \vee \ldots \vee F_n) \leftarrow (T_1 \wedge \ldots \wedge T_m))$$

  - az implikáció bal (következmény) oldala a klóz **feje**
  - az implikáció feltétele a klóz **törzse**, a törzsbeli konjunkció elemeit (rész)**cél**oknak is hívjuk
  - F_i és T_j elemi állítások, $n, m \geq 0$, azaz a fej és a törzs is lehet üres.
  - $X_1 \ldots X_j$: a klózban szereplő összes változó.
- A fentivel ekvivalens logikai alak (vö. $A \leftarrow B \equiv A \vee \neg B$):

$$\forall X_1, \ldots X_j(F_1 \vee \ldots \vee F_n \vee \neg T_1 \vee \ldots \vee \neg T_m)$$

- Klózok egyszerűsített írásmódja: $F_1, \ldots, F_n : -T_1, \ldots, T_m.$ Ha $m = 0$, a `:-` jelet elhagyjuk.
- Példák — vigyázat, ezek általános klózok, nem feltétlenül megengedettek Prologban!

```
ferfi(X), no(X) :- ember(X).     % Aki ember az férfi vagy nő.
:- ferfi(X), no(X).              % ≡ ∀ X ¬ (ferfi(X) ∧ no(X))
                                 % Nincs olyan dolog, ami férfi és nő is.
szereti(X, X) :- szent(X).       % Minden szentnek maga felé hajlik a keze.
szent('István').                 % István szent.
```


Definit klózok — a Prolog program építőelemei

- Általános klóz (ismétlés): $F_1, \ldots, F_n : -T_1, \ldots, T_m.$ $\forall \overline{X}(F_1 \vee \ldots \vee F_n \vee \neg T_1 \vee \ldots \vee \neg T_m)$
- Definit klóz (definite clause) vagy Horn klóz (Horn clause):
  - olyan klóz, amelynek fejében legfeljebb egy elemi állítás szerepel ($n \leq 1$).
- Horn klózok osztályozása
  - Ha $n = 1, m > 0$, akkor a klózt **szabály**nak hívjuk, pl.
    `nagyszuloje(U,N):-szuloje(U,Sz),szuloje(Sz,N).`
    logikai alak: $\forall UNSz(\texttt{nagyszuloje}(U, N) \leftarrow \texttt{szuloje}(U, Sz) \wedge \texttt{szuloje}(Sz, N))$
    ekvivalens alak: $\forall UN \quad (\texttt{nagyszuloje}(U, N) \leftarrow \exists Sz(\texttt{szuloje}(U, Sz) \wedge \texttt{szuloje}(Sz, N)))$
  - $n = 1, m = 0$ esetén a klóz **tényállítás**, pl.
    `szuloje('Imre', 'István').`
    logikai alakja változatlan.
  - $n = 0, m > 0$ esetén a klóz egy **célsorozat**, pl.
    `:- nagyszuloje('Imre', X).`
    logikai alak: $\forall X \neg \texttt{nagyszuloje}(\texttt{'Imre'}, X)$, azaz $\neg \exists X \texttt{nagyszuloje}(\texttt{'Imre'}, X)$
  - Ha $n = 0, m = 0$, akkor **üres klóz**ról beszélünk, jele: $\square$. Logikailag üres diszjunkció, azaz azonosan hamis.

A Prolog mint logikai nyelv

- Szintaxis:
  - Prolog program: szabályok és tényállítások halmaza. Példa:

```
szuloje('Imre', 'István').
(...)
szuloje('Gizella', 'Burgundi Gizella').
nagyszuloje(Gy, N) :- szuloje(Gy, Sz), szuloje(Sz, N).
```

  - Egy klóz fejének nevét és argumentumszámát együtt a klóz **funktor**ának hívjuk és `Név/Argszám` alakban írjuk.
  - Az azonos funktorú klózok alkotják egy **predikátum** (vagy eljárás) definícióját. A fenti példa a `szuloje/2` és `nagyszuloje/2` predikátumokat definiálja.
  - Programok javasolt formázása:
    - Az egy predikátumhoz tartozó klózok legyenek egymás mellett a programban, közéjük ne tegyünk üres sort. A predikátumokat válasszuk el üres sorokkal.
    - A klózfejet írjuk sor elejére, minden célt lehetőleg külön sorba, néhány szóközzel beljebb kezdve
  - Egy program futtatásához megadandó egy célsorozat. Példa:
    `:- nagyszuloje('Imre', N).`

A Prolog mint logikai nyelv

- Deklaratív szemantika
  - Segédfogalom: egy kifejezés/állítás **példánya**: belőle változók behelyettesítésével előálló kifejezés/állítás.
  - Egy célsorozat lefutása **sikeres**, ha a célsorozat törzsének egy példánya logikai **következménye** a programnak (a programbeli klózok konjunkciójának).
  - A futás eredménye a példányt előállító **behelyettesítés**.
  - Egy célsorozat többféleképpen is lefuthat sikeresen.
  - Egy célsorozat futása **sikertelen**, ha egyetlen példánya sem következménye a programnak.
- Példa:

```
szuloje('Imre', 'István').                          (sz1)
szuloje('Imre', 'Gizella').                         (sz2)
szuloje('István', 'Géza').                          (sz3)
szuloje('István', 'Saroltʼ).                        (sz4)
szuloje('Gizella', 'Civakodó Henrik').              (sz5)
szuloje('Gizella', 'Burgundi Gizella').             (sz6)
nagyszuloje(Gy, N) :- szuloje(Gy, Sz), szuloje(Sz, N).  (nsz)
:- nagyszuloje('Imre', N).                          (cel)
```

  - `(sz1)` + `(sz3)` + `(nsz)` következménye: `nagyszuloje('Imre', 'Géza')`, tehát `(cel)` sikeresen fut le az `N = 'Géza'` behelyettesítéssel.
  - Egy másik sikeres lefutás, pl. `(sz1)`+`(sz4)`+`(nsz)` alapján `N = 'Saroltʼ`.

Deklaratív szemantika

- Miért jó a deklaratív szemantika?
  - A program **dekomponálható**: külön-külön vizsgálhatjuk az egyes predikátumokat (sőt az egyes klózokat).
  - A program (informálisan) **verifikálható**: a predikátumok szándékolt jelentésének ismeretében eldönthető, hogy az egyes klózok igaz állításokat fogalmaznak-e meg.
  - Egy predikátum szándékolt jelentését nagyon fontos egy ún. **fejkomment**ben, azaz az argumentumok kapcsolatát leíró kijelentő mondatban megfogalmazni. Példák:
    - Fejkommentek: `% szuloje(Gy, Sz):` *Gy* szülője *Sz.*
      `% nagyszuloje(Gy, NSz):` *Gy* nagyszülője *NSz.*

      `nagyszuloje(Gy, N) :- szuloje(Gy, Sz), szuloje(Sz, N).`

      A klóz jelentése: Ha `Gy` szülője `Sz` és `Sz` szülője `N`, akkor `Gy` nagyszülője `N`. Ez megfelel elvárásainknak, **igaz állítás**ként elfogadható.
    - Fejkommentek: `% sum_tree(T, Sum):` A *T* fa levélösszege *Sum.*
      `% E is Kif:` A *Kif* aritmetikai kifejezés értéke *E.* `(is infix!)`

      `sum_tree(node(L,R), S) :- sum_tree(L, S1), sum_tree(R, S2), S is S1+S2.`

      A klóz jelentése: Ha az `L` fa levélösszege `S1`, az `R` fa levélösszege `S2`, és `S1+S2` értéke `S` akkor a `node(L,R)` fa levélösszege `S`. Ez is egy igaz állítás.

Deklaratív szemantika (folyt.)

- Miért nem elég a deklaratív szemantika?
  - A deklaratív szemantika egy általános következményfogalomra épít.
  - A következtetés szükségképpen többirányú, tehát kereséssel jár.
  - Végtelen keresési tér esetén a következtető is **végtelen ciklus**ba eshet.
  - Véges keresési tér esetén is lehet a keresés nagyon **rossz hatékonyságú**.
  - Egyes **beépített predikátumok** csak bizonyos feltételek mellett képesek működni. Pl. `S is S1+S2` hibát jelez, ha `S1` vagy `S2` ismeretlen mennyiség. Emiatt

    `sum_tree(node(L,R), S) :- S is S1+S2, sum_tree(L, S1), sum_tree(R, S2).`

    logikailag helyes, de működésképtelen.
- Ezek miatt fontos, hogy a Prolog programozó ismerje a Prolog pontos végrehajtási mechanizmusát is, azaz a nyelv **procedurális szemantikáját**.
- Jelszó: Gondolkodj és programozz deklaratívan, ellenőrizd a programot procedurálisan!

A Prolog procedurális szemantikája

- A Prolog végrehajtási mechanizmusa többféleképpen is leírható. Ez nem más mint:
  - Az ún. SLD rezolúciós tételbizonyítási módszer (nagyon tömören lásd alább); avagy
  - egy cél-redukción alapuló tételbizonyítási módszer (lásd a következő fóliákon); avagy
  - mintaillesztésen alapuló visszalépéses eljárásszervezés (részletesen lásd később).
- A Prologban alkalmazott rezolúciós tételbizonyítási módszerről:
  - SLD resolution: **L**inear resolution with a **S**election function for **D**efinite clauses.
  - A célsorozat **tagadja** a keresett dolgok létezését, pl. `'Imre'`-nek nincs nagyszülője: `:- nagyszuloje('Imre', N).` $\equiv \neg\exists N$ `nagyszuloje('Imre',`N`)`
  - A célsorozat és egy programklóz ún. rezolvenseként kapunk egy újabb célsorozatot.
  - A rezolúciós lépéseket addig ismételjük, amíg el nem jutunk az üres klózhoz (zsákutcák esetén visszalépést alkalmazva).
  - Ha ez sikerül, akkor ezzel **indirekt** módon beláttuk, hogy a célsorozat törzse következik a programból, hiszen a törzs negáltjából és a programból következik az azonosan hamis $\square$.
  - A rezolúciós bizonyítás konstruktív, siker esetén behelyettesíti a célsorozat változóit — ez a keresett válasz (pl. $N =$ `'Géza'`).
  - További válaszok alternatív bizonyításokkal állíthatók elő.

A Prolog mint cél-redukciós tételbizonyító

- Példaprogram

```
szuloje('Imre', 'István').                              (sz1)
szuloje('Imre', 'Gizella').                             (sz2)
szuloje('István', 'Géza').                              (sz3)
(...)

nagyszuloje(Gy, N) :- szuloje(Gy, Sz), szuloje(Sz, N).  (nsz)
```

- A kezdeti célsorozat: `:- nagyszuloje('Imre', N).`
  (Most a célsorozatot úgy tekintjük mint bizonyítandó állítások sorozatát.)
- Kiegészítjük a célsorozatot egy vagy több speciális céllal, a keresett változók értékének megőrzése érdekében:
  `:- nagyszuloje('Imre', N), answer(N).`
- A célsorozatot ismételten **redukáljuk** (lásd következő fólia), amíg csak `answer` cél marad:

```
:- szuloje('Imre', Sz), szuloje(Sz, N), answer(N).
:- szuloje('István', N), answer(N).
:- answer('Géza').
```

- A futás eredményét az `answer` argumentumából olvashatjuk ki.

A redukciós lépés

- A példa érintett klózai és a célsorozat:

```
szuloje('Imre', 'István').                              (sz1)
szuloje('István', 'Géza').                              (sz3)
nagyszuloje(Gy, N) :- szuloje(Gy, Sz), szuloje(Sz, N).  (nsz)

:- nagyszuloje('Imre', N), answer(N).
```

- Egy redukciós lépés végrehajtása:
  - A célsorozat **első** eleméhez megkeressük az **első** olyan klózt, amelynek fejét a céllal azonossá tudjuk tenni, változók behelyettesítésével. A példában ez az `(nsz)` klóz.
  - Mind a klózt, mind a célsorozatot **specializáljuk** a kívánt behelyettesítések elvégzésével. A példában előállítjuk `(nsz)` speciális esetét:
    `nagyszuloje('Imre', N) :- szuloje('Imre', Sz), szuloje(Sz, N).  (nsz*)`
  - Az első célt helyettesítjük a klóz törzsével, azaz ezt a célt egy előfeltételére redukáljuk. A példában az új célsorozat: `szuloje('Imre', Sz), szuloje(Sz, N), answer(N).`
- A következő lépésben az `(sz1)` klózzal redukálunk, a **célsorozatot** specializálva az `Sz = 'István'` behelyettesítéssel: `szuloje('István', N), answer(N).`
  Mivel tényállítással redukálunk, üres törzset helyettesítünk, így a célsorozat hossza csökken.
- A `(sz3)` ténnyel való hasonló redukciós lépés eredménye: `answer('Géza').`

Redukciós lépés — további részletek

- Változók kezelése
  - A változók hatásköre egy klózra terjed ki (vö. $\forall X_1 \ldots X_j (F \leftarrow T)$).
  - A redukciós lépés előtt a klózt le kell másolni, a változókat szisztematikusan újakra cserélve (vö. rekurzió).
- **Egyesítés**: két kifejezés/állítás azonos alakra hozása, változók behelyettesítésével.
  - A változókat tetszőleges kifejezéssel lehet helyettesíteni, akár más változóval is.
  - Az egyesítés a **legáltalánosabb** közös alakot állítja elő. Pl.

`sum_tree(leaf(X), X)`	közös alakja	`sum_tree(leaf(X), X)` és nem pl.
`sum_tree(T, V)`		`sum_tree(leaf(0), 0)`

  - Az egyesítés eredménye a legáltalánosabb közös alakot előállító behelyettesítés. Ez változó-átnevezéstől eltekintve egyértelmű. A példában: `T=leaf(X), V=X`.
  - Példák:

Hívás:	*Fej:*	*Behelyettesítés:*
`nagyszuloje('Imre', N)`	`nagyszuloje(Gy, NSz)`	`Gy = 'Imre', NSz = N`
`szuloje('Imre', Sz)`	`szuloje('Imre', 'István')`	`Sz = 'István'`
`szuloje('Imre', Sz)`	`szuloje('István', 'Géza')`	*nem egyesíthető*
`szereti('István', Kit)`	`szereti(X, X)`	`X = 'István', Kit = 'István'`
`szereti(Ki, Kit)`	`szereti(X, X)`	`X = Ki, Kit = Ki`

Választási pontok, visszalépés

- A példában „szerencsénk” volt, a redukciós lépések sorozata elvezetett egy megoldáshoz.
- Az általános esetben zsákutcába, egy nem redukálható célsorozathoz is juthatunk, pl.

```
:- nagyszuloje('Imre', 'Civakodó Henrik').              (nsz)
:- szuloje('Imre', Sz), szuloje(Sz, 'Civakodó Henrik'). (sz1): szuloje('Imre', 'István')
:- szuloje('István', 'Civakodó Henrik').                ???
```

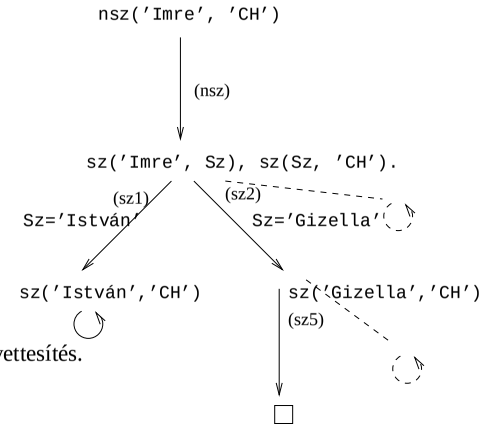
- A 2. célsorozatot `(sz1)`-vel redukáltuk, de a megoldáshoz az `(sz2): szuloje('Imre', 'Gizella')` vezet — nem csak az első egyesíthető klózfejet kell kezelnünk, hanem az összeset!
- Ha nem az utolsó klózzal redukálunk, akkor létrehozunk egy **választási pontot**, ebben elmentjük a célsorozatot és azt, hogy melyik klózzal redukáltuk.
- **Zsákutca**, vagy **új megoldás** kérése esetén visszatérünk a legutóbbi (legfiatalabb) választási ponthoz és ott a **fennmaradó** (még ki nem próbált) klózok között folytatjuk a keresést.
- Ha egy választási pontnál nem találunk újabb klózt, újabb visszalépés következik. Ha nincs választási pont ahova visszaléphetnénk, akkor a célsorozat futása meghiúsul.
- A fenti példában: visszatérünk a második lépéshez, és ott az `(sz2)` klózzal próbálkozunk:

```
(...)
:- szuloje('Imre', Sz), szuloje(Sz, 'Civakodó Henrik').  (sz1)
:- szuloje('Gizella', 'Civakodó Henrik').                (sz5)
□
```

Visszalépéses keresés szemléltetése keresési fával

```
sz('Imre', 'István').      % (sz1)
sz('Imre', 'Gizella').     % (sz2)
sz('István', 'Géza').      % (sz3)
sz('István', 'Sarolt').    % (sz4)
sz('Gizella', 'CH').       % (sz5)
sz('Gizella', 'BG').       % (sz6)

nsz(Gy, N) :-
    sz(Gy, Sz), sz(Sz, N).  % (nsz)
```



- A keresési fa
  - csomópontjai a végrehajtási állapotok
  - címkék:
    - csomópontokban: célsorozatok,
    - éleken: a kiválasztott klóz és a behelyettesítés.
- A Prolog keresés: a keresési fa bejárása
  - balról jobbra,
  - mélységi (depth-first) kereséssel.
- A szaggatott vonalak sikertelen klózkeresésre utalnak, az ún. első argumentum szerinti indexelés a felsőt kiküszöböli.

A keresési tér bejárásának nyomkövetése

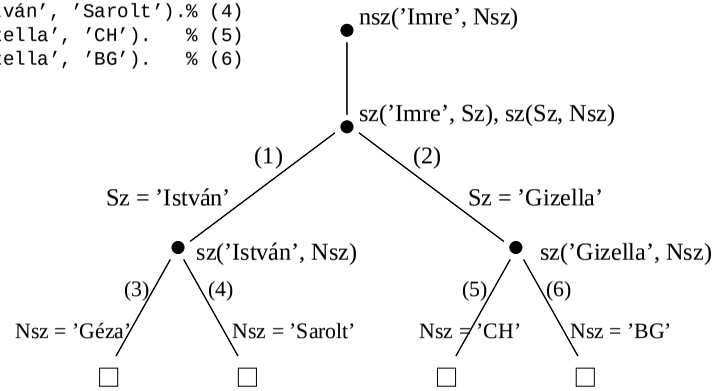
- Egy (szerkesztett) párbeszéd a redukciós nyomkövetővel, a meghiúsuló egyesítéseket elhagytuk.

```
|| ?- nagyszuloje('Imre', 'Civakodó Henrik').
G0:   nagyszuloje('Imre','Civakodó Henrik') ?            <--- ujsor leütésére folytatja
|                                          Trying clause 1 of nagyszuloje/2 ... successful
| (1) {Gyerek_1 = 'Imre', Nagyszulo_1 = 'Civakodó Henrik'}<--- változó-átnevezés
|
G1:   szuloje('Imre',Szulo_1), szuloje(Szulo_1,'Civakodó Henrik') ?
|                                          Trying clause 1 of szuloje/2 ... successful
|   (1) {Szulo_1 = 'István'}
|
|-----G2:   szuloje('István','Civakodó Henrik') ?
(...)                                                    <--- G3-G8 6 sikertelen klózillesztés
|     |<<<< Failing back to goal G1                      <--- Van-e másik szuloje 'Imre'-nek?
|                                          Trying clause 2 of szuloje/2 ... successful
|   (2) {Szulo_1 = 'Gizella'}
|
|-----G9:   szuloje('Gizella','Civakodó Henrik') ?
|     |                                    Trying clause 5 of szuloje/2 ... successful
|     |   (5) {}
|     |-----G14:  [] ?                                   <--- üres klóz, siker
|     |     |++++ Solution:  ?
|     |<<<< Failing back to goal G1              <--- az előző fólián alsó szaggatott
|
|<<<< No more choices                            <--- az előző fólián felső szaggatott
```


Keresési fa — újabb példa

```
sz('Imre', 'István').  % (1)
sz('Imre', 'Gizella'). % (2)
sz('István', 'Géza').  % (3)
sz('István', 'Sarolt').% (4)
sz('Gizella', 'CH').   % (5)
sz('Gizella', 'BG').   % (6)
```

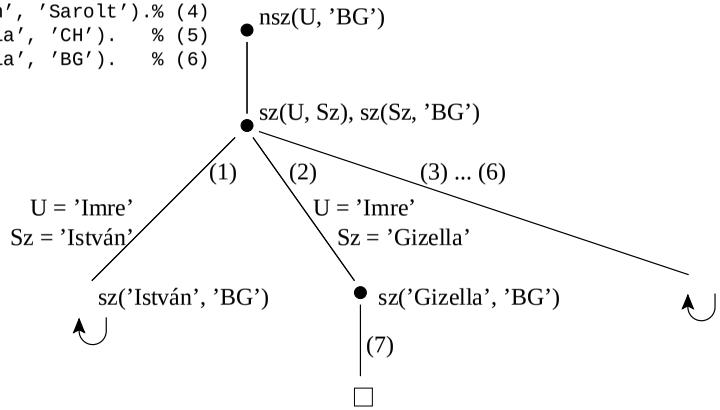
```
nsz(Gy, N) :-
    sz(Gy, Sz), sz(Sz, N).
```



Keresési fa — még újabb példa

```
sz('Imre', 'István').  % (1)
sz('Imre', 'Gizella'). % (2)
sz('István', 'Géza').  % (3)
sz('István', 'Sarolt').% (4)
sz('Gizella', 'CH').   % (5)
sz('Gizella', 'BG').   % (6)
```

```
nsz(Gy, N) :-
    sz(Gy, Sz), sz(Sz, N).
```



Az „őse” predikátum definiálása

- Az „őse” reláció a „szülője” reláció tranzitív lezártja: a szülő ős, és az ős őse is ős, azaz:

```
% ose0(E, Os): E ose Os.
ose0(E, Sz) :- szuloje(E, Sz).                % (1)
ose0(E, Os) :- ose0(E, Os0), ose0(Os0, Os).   % (2)
```

- Az `ose0` definíciója matematikailag helyes, de végtelen Prolog keresési teret ad:

```
szuloje(gyerek,apa). szuloje(gyerek,anya). szuloje(anya,nagyapa).

| ?- ose0(gyerek, Os).
        Os = apa ? ; Os = anya ? ;  {kb 30 másodperc után:}
        ! Resource error: insufficient memory
```

- A végtelen rekurzió oka: Az `:- ose0(apa, X).` cél esetén az (1) klóz meghiúsul, (2) pedig egy `:- ose0(apa, Y), ose0(Y, X).` célsorozathoz vezet stb.
- A balrekurziót kiküszöbölve kapjuk:

```
ose1(E, Sz) :- szuloje(E, Sz).                % (3)
ose1(E, Os) :- szuloje(E, Sz), ose1(Sz, Os).  % (4)

| ?- ose1(gyerek, Os).
Os = apa ? ; Os = anya ? ; Os = nagyapa ? ; no
```

- Ez minden `szuloje(X,Y)` részcélt kétszer hajt végre: (3)-ban és (4)-ben.

A diszjunkció

- Az `ose1` predikátum hatékonyabbá tehető klózai összevonásával:

```
ose2(E, Os) :- szuloje(E, Sz), maga_vagy_ose(Sz, Os).

maga_vagy_ose(E, E).
maga_vagy_ose(E, Os) :- ose2(E, Os).
```

- A `maga_vagy_ose` predikátum egy ún. **diszjunkció** bevezetésével kiküszöbölhető:

```
ose3(E, Os) :-
        szuloje(E, Sz).
        (   Os = Sz
        ;   ose3(Sz, Os)
        ).
```

- A SICStus Prolog ténylegesen úgy implementálja a fenti diszjunkciót, hogy bevezet egy `maga_vagy_ose`-vel azonos segéd-predikátumot és az `ose3` klózt `ose2`-vé alakítja.
- Az `X=Y` beépített predikátum a két argumentumát egyesíti.
- Az `= /2` eljárás egy tényállítással definiálható: `U = U.` ≡ `=(U, U).`

A diszjunkció mint szintaktikus édesítőszer

- A diszjunkció akárhány tagú lehet. A ‘;’ művelet gyengébben köt mint a ‘,’, ezért a diszjunkciót mindig zárójelbe tesszük, mig az ágait nem kell zárójelezni. Példa, „szabványos” formázással:

```
a(X, Y, Z) :-
        p(X, U), q(Y, V),
        (   r(U, T), s(T, Z)
        ;   t(V, Z)
        ;   t(U, Z)
        ),
        u(X, Z).
```

- A diszjunkció egy segéd-predikátummal mindig kiküszöbölhető
  - Megkeressük azokat a változókat, amelyek a diszjunkcióban és azon kívül is előfordulnak
  - A segéd-predikátumnak ezek a változók lesznek az argumentumai
  - A segéd-predikátum minden klóza megfelel a diszjunkció egy ágának

```
seged(U, V, Z) :- r(U, T), s(T, Z).
seged(U, V, Z) :- t(V, Z).
seged(U, V, Z) :- t(U, Z).

a(X, Y, Z) :-
        p(X, U), q(Y, V),
        seged(U, V, X),
        u(X, Z).
```

- A diszjunkció szemantikáját (jelentését) ezzel a segéd-predikátumos átalakítással definiáljuk.

Diszjunkció — megjegyzések

- Az egyes klózok ‘ÉS’ vagy ‘VAGY’ kapcsolatban vannak?
  - A program klózai **ÉS** kapcsolatban vannak, pl.
    `szuloje('Imre', 'István').` `szuloje('Imre', 'Gizella').`
    jelentése: Imre szülője István **ÉS** Imre szülője Gizella.
  - Az ÉS kapcsolatban levő klózok alternatív (VAGY kapcsolatban levő) válaszokhoz vezetnek:
    `:- szuloje('Imre' Sz).` ⇒ `Sz = 'István' ? ; Sz = 'Gizella' ? ; no`
    A „Ki Imre szülője?” kérdésre a válasz: István vagy Gizella.
- A fenti két klózos predikátum átalakítható egyetlen klózzá, diszjunkció segítségével:

```
szuloje('Imre', Sz) :-
        (   Sz = 'István'          (*)
        ;   Sz = 'Gizella'         (*)
        ).
```

  A konjunkció ezáltal diszjunkcióvá alakult (vö. De Morgan azonosságok).
- Általánosan: tetszőleges predikátum egyklózossá alakítható:
  - a klózokat átalakítjuk azonos fejűvé, új változók és egyenlőségek bevezetésével:

```
szuloje('Imre', Sz) :- Sz = 'István'.
szuloje('Imre', Sz) :- Sz = 'Gizella'.
```

  - a klóztörzseket egy diszjunkcióvá fogjuk össze, amely az új predikátum törzse (lásd `(*)`).

Családi kapcsolatok: a „testvére” reláció

- Példa-adatbázis:

```
szuloje('Imre', 'István').
szuloje('Imre', 'Gizella').
szuloje('Ottó', 'István').
szuloje('Ottó', 'Gizella').
```

- Definiáljuk a „testvére” kapcsolatot! Első kisérlet:

```
testvere(X, Y) :-                       % X es Y testvérek
        szuloje(X, Sz), szuloje(Y, Sz).   % ha van közös szülőjük.
```

```
| ?- testvere('Imre', X).     ⟹ X = 'Imre' ?
```

- Használjuk a `X\=Y` beépített predikátumot!
  `X\=Y` jelentése: `X` és `Y` nem egyesíthető:

```
testvere(X, Y) :-                       % X es Y testvérek
        szuloje(X, Sz), szuloje(Y, Sz),   % ha van közös szülőjük,
        X \= Y.                           % és nem azonosak
```

```
| ?- testvere('Imre', X). ⟹ X = 'Ottó' ? ; X = 'Ottó' ? ; no
```

- Azért kapunk két `'Ottó'` választ, mert mindkét szülő segítségével bizonyítható az, hogy ő Imre testvére. Később lesz szó arról, hogyan szüntethető meg ez a „dupla válasz” ...

Családi kapcsolatok: további példák

- Tegyük fel, hogy adatbázisunkban a gyerek-szülő kapcsolat mellett tároljuk az emberek nemét és születési dátumát:

```
% szuloje(Gy, Sz): Gy szülője Sz.
% neme(E, Nem): E neme Nem, ahol Nem lehet ferfi vagy no.
% szuletesi_datuma(E, D): E szuletesi datuma D, D egy EEEEHHNN alakú szám.
```

- „Házi feladat” (nem beadható): gyakorlásképpen definiálják az alábbi családi kapcsolatokat!

```
% apja(Gy, Apa): Gy apja Apa.
% nagyanyja(Gy, NA): Gy nagyanyja NA.
% occse(E1, E2): E1 öccse E2.
% novere(E1, E2): E1 nővére E2.
% nagynenje(E1, E2): E1 nagynénje E2.
% unokatestvere(E1, E2); E1 unokatestvére E2.
```

- Dátumok összehasonlítására használható az `X < Y` beépített predikátum.

És mi történt a logika függvényjeleivel?

- A Prolog logikai rendszere nem engedi meg, hogy a függvények értékére bármilyen megkötést tegyünk, nem írhatunk fel állításokat a függvényértékekre. Például **nem megengedett** állítások:
  `(X+Y)+Z = X+(Y+Z).` `X+Y > X :- Y > 0.`
- Emiatt Prologban $f(x_1,\ldots,x_n) = f(y_1,\ldots,y_n) \leftrightarrow x_1 = y_1,\ldots,x_n = y_n$
- Ez azt jelenti, hogy minden függvény adat-konstruktor!
- `f(`X_1`,...,`X_n`)` nem más, mint az $X_1,\ldots,X_n$ **mezők**ből felépített `f` címkéjű **fastruktúra**, pl.

```
% sum_tree(Tree, S): A számokból felépített Tree bináris fa levélösszege S.
sum_tree(leaf(Value), Value).
sum_tree(node(Left,Right), S) :-
        sum_tree(Left, S1), sum_tree(Right, S2), S is S1+S2.
```

- A függvények fastruktúrákká „szilárdulása” az egyesítési algoritmus által valósul meg. Ez csak akkor tekint két „függvény”-kifejezést azonos alakra hozhatónak, ha:
  - nevük, argumentumszámuk megegyezik, és
  - (rekurzív módon) argumentumaik is rendre azonos alakra hozhatóak.
- Például: `| ?- 2+2 = 2*2` ⟹ `no`, `| ?- 2+3 = 3+2` ⟹ `no` stb.
- Az `f(`X_1`,...,`X_n`)` alakú Prolog kifejezést ezután **összetett-** vagy **struktúra-kifejezés**nek hívjuk, amelynek **struktúraneve** `f`. A struktúra **funktora** `f/`n.

A logika adatfogalma

- A logika adatfogalma: Herbrand univerzum
  - Egy adott logikai állítás-gyűjtemény (klózhalmaz) Herbrand univerzuma:
    - tekintsük az állításokban előforduló konstansneveket és függvényneveket,
    - a Herbrand univerzum: a konstansnevek és a belőlük, a függvénynevek tetszőleges számú alkalmazásával előálló kifejezések
    - az ilyen változómentes kifejezéseket **tömör** (ground) kifejezésnek hívjuk.
  - P1 példaprogram:
    `szuloje(a,b).  szuloje(c,b).  szuloje(b,d).`
    `mvo(X,X). mvo(X,Z) :- szuloje(X,Y), mvo(Y,Z). % mvo` = maga vagy őse
    - Herbrand univerzum: H_1 = {a, b, c}
  - P2 példaprogram: `p(a).  p(f(X)) :- p(X). q(Y). :- q(c).`
    - Herbrand univerzum: H_2 = {a, c, f(a), f(c), f(f(a)), f(f(c)), ...}
  - A Herbrand univerzummal izomorf részt minden modell alaphalmazának tartalmaznia kell
  - A Herbrand univerzumon definiálható egy ún. minimális model (minden hamis, amiről nem bizonyítható, hogy igaz). Pl. a P1 példaprogram minimális modelljében
    `szuloje` jelentése = $\{\langle a,b\rangle, \langle b,d\rangle, \langle c,b\rangle\} \subset H_1 \times H_1$
    `mvo` jelentése = $\{\langle a,a\rangle, \langle a,b\rangle, \langle a,d\rangle, \langle b,b\rangle, \langle b,d\rangle, \langle c,c\rangle, \langle c,b\rangle, \langle c,d\rangle, \langle c,b\rangle\} \subset H_1 \times H_1$

A Prolog modellalapú szemantikája

- A Prolog végrehajtás adathalmaza: megfelel egy olyan Herbrand univerzumnak, ahol a konstansnevek közé a számokat is beveszük. Például a P2 példaprogramban:
  - $H_2' = H_2 \cup$ {0,f(0),f(f(0)),-1,f(-1),f(f(-1)),0.5,f(0.5),f(f(0.5)),...}
- Prolog deklaratív szemantika — újabb változat:
  - Egy célsorozat futása azokat a behelyettesítéseket állítja elő, amelyekre a célsorozat fennáll a minimális modelben.
  - A Prolog nem csak tömör behelyettesítéseket állít elő, hanem olyanokat is, amelyek (általában összekapcsolt) változókat tartalmaznak.

```
| ?- szuloje(X,Y).
X = a, Y = b ? ;
X = c, Y = b ? ;
X = b, Y = d ? ;
no
```

```
| ?- mvo(X,Y).
Y = X ? ;  % <-- 3 megoldást lefed!
X = a, Y = b ? ;
X = a, Y = d ? ;
X = c, Y = b ? ;
X = c, Y = d ? ;
X = b, Y = d ? ;
no
```

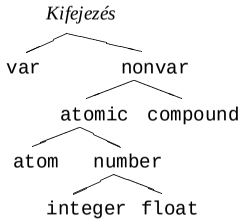
  - Vö. `szuloje` jelentése = {⟨a,b⟩, ⟨b,d⟩, ⟨c,b⟩}
    `mvo` jelentése = {⟨a,a⟩, ⟨a,b⟩, ⟨a,d⟩, ⟨b,b⟩, ⟨b,d⟩, ⟨c,c⟩, ⟨c,b⟩, ⟨c,d⟩, ⟨c,b⟩}

Változókat tartalmazó megoldások

- `| ?- mvo(X,Y).` ⟹ `Y = X ?`
- Ez egy „paraméteres” megoldásközlés: az `X` paraméter **tetszőlegesen** megválasztható, de `Y`-nak ezzel azonosnak kell lennie.
- Tehát `X` tetszőlegesen megválasztható, mint a Herbrand univerzum egy eleme, és `Y` ugyanez az érték kell legyen.
- `| ?- sum_tree(X,Y).` ⟹ `X = leaf(Y) ?`
- Itt `Y` választható meg tetszőlegesen, ennek `leaf/1`-be „csomagolt” változata lesz `X`.
- Ez „idegen” megoldásokat is jelent, pl. `mvo(X,Y)` teljesül `X=Y=1` esetén, `sum_tree(X,Y)`, `X=leaf(a), Y=a` esetén.
- Ez ellen nem érdemes programkód beépítésével védekezni, mert az lényegesen lassíthatja a futást, vagy ronthatja a program többirányú használatát. (Szóba jöhet viszont egy fordítási idejű típusellenőrző használata, pl. TCLP, `http://contraintes.inria.fr/~coquery/tclp`.)

A Prolog adatfogalma

- Prolog kifejezések osztályozása — osztályozó beépített predikátumok



`var(X)`	X változó
`nonvar(X)`	X nem változó
`atomic(X)`	X konstans
`compound(X)`	X struktúra
`atom(X)`	X atom
`number(X)`	X szám
`integer(X)`	X egész szám
`float(X)`	X lebegőpontos szám

- Egy osztályozó predikátum az argumentuma **pillanatnyi** állapotát **ellenőrzi**, logikailag nem tiszta:

```
| ?- X = 1, integer(X).              ⟹ yes
| ?- integer(X), X = 1.              ⟹ no
| ?- atom('István'), atom(istvan).   ⟹ yes
| ?- compound(leaf(X)).              ⟹ yes
| ?- compound(X).                    ⟹ no
```


Az egyesítés mint adat-építő és -kiválasztó művelet

- Példa:

```
% balcsonk(Fa, E): Fa egy olyan bináris fa, amelynek legfelső
%                   csomópontjában balra egy E értékű levél van
balcsonk(node(leaf(V),_), V).

% jobbcsonk(Fa, E): Fa egy olyan bináris fa, amelynek legfelső
%                    csomópontjában jobbra egy E értékű levél van
jobbcsonk(node(_,leaf(V)), V).
```

- A Prolog egyesítés egyaránt használható adat-építésre (konstrukció) és -kiválasztásra (szelekció). Példák:
  - Mindkét argumentum bemenő: ellenőrzés.
    `| ?- Fa=node(leaf(1),leaf(2)), balcsonk(Fa, 1).` ⟹ `yes`
  - A fa adott, az érték kimenő: levélérték elővétele (vagy meghiúsulás).
    `| ?- Fa=node(leaf(1),leaf(2)), balcsonk(Fa, B), jobbcsonk(Fa, J).`
    ⟹ `B = 1, J = 2 ? ; no`
    `| ?- balcsonk(node(node(leaf(1),leaf(2)),leaf(3)), B).` ⟹ `no`
  - Az érték adott, a fa kimenő: fa építése.
    `| ?- balcsonk(Fa, 1), jobbcsonk(Fa, 2).` ⟹ `Fa=node(leaf(1),leaf(2)) ? ; no`

Kifejezések összerakása (folyt.)

- Vizsgáljuk a fa felépítésének részleteit!

célsorozat:	`:- balcsonk(Fa, 1), jobbcsonk(Fa, 2)`
klózfej:	`balcsonk(node(leaf(_B),_A), _B)`
behelyettesítés:	`_B = 1, Fa= node(leaf(1),_A)`
új célsorozat:	`:- jobbcsonk(node(leaf(1), _A), 2)`
klózfej:	`jobbcsonk(node(_C, leaf(_D)), _D)`
behelyettesítés:	`_D = 2, _A = leaf(2), _C = leaf(1)`
eredmény:	`Fa = node(leaf(1),leaf(2))`

- Kövessük nyomon a fenti végrehajtást Prolog hívásonként:
  - `| ?- balcsonk(Fa, 1).` $\Longrightarrow$ `Fa = node(leaf(1),_A) ? ; no`
    `Fa` értéke egy változót tartalmazó Prolog adatstruktúra, mindazon összetett (nem levél) fákat jelenti, amelyek baloldali részfája az 1 értékű levél. Ez egy **kifejezés-minta**.
  - `| ?- Fa = node(leaf(1),_A), jobbcsonk(Fa, 2).`
    $\Longrightarrow$ `Fa = node(leaf(1),leaf(2)) ? ; no`
    A `jobbcsonk` hívás a `Fa` kifejezés-mintát, **finomítja**, ebben az esetben egy tömör fára szűkíti le.

A logikai változó

- A logikai változó fogalma:
  - kifejezésként, kifejezésben egyaránt előfordulhat
  - a változók egymással is azonossá tehetők: pl. két azonos változó egy kifejezésben.
  - a változó „teljes jogú” állampolgár a (rész)kifejezések világában
- SML-ben is van mintaillesztés, de a minta csak szétszedésre használható, összerakásra nem; a mintabeli változók mindig (tömör) értéket kapnak.
- (Egyes újabb funkcionális nyelvek, pl. az Oz nyelv, támogatják a logikai változókat.)
- Példa: Az alábbi célsorozat egy két **azonos** levélből álló bináris fát épít fel a `Fa` változóban. A levelek értéke **azonos** lesz a célsorozatbeli `X` változóval:

```
balcsonk(node(leaf(V),_), V).
jobbcsonk(node(_,leaf(V)), V).

| ?- balcsonk(Fa, X), jobbcsonk(Fa, X). ⟹ Fa = node(leaf(X),leaf(X)) ? ; no
```

- Ha az összekapcsolt változók bármelyike értéket kap, a többi is erre az értékre helyettesítődik:

```
| ?- balcsonk(Fa, X), jobbcsonk(Fa, X), X = 1.
                                ⟹ X = 1, Fa = node(leaf(1),leaf(1)) ? ; no
| ?- balcsonk(Fa, X), jobbcsonk(Fa, X), balcsonk(Fa, 2).
                                ⟹ X = 2, Fa = node(leaf(2),leaf(2)) ? ; no
```


Példasor: bináris fák kezelése

- Az egészekből álló bináris fa különböző meghatározásai:
  - Szöveges definícióként (ismétlés):
    - vagy egy levél (`leaf(V)`), ahol `V` egy egész szám
    - vagy egy csomópont (`node(L,R)`), ahol `L` és `R` egészekből álló bináris fák
  - Matematikai jelöléssel:
    `itree` $\equiv$ {`leaf(`i`)` | $i \in$ `integer` } $\cup$ { `node(`l`,`r`)` | $l,r \in$ `itree` }
  - A Mercury típusos logikai programozási nyelv jelöléseivel:

```
:- type itree --->  node(itree, itree) | leaf(int).
```

  - Egy **ellenőrző** Prolog predikátumként:

```
itree(leaf(V)) :-
        integer(V).          % <--- osztályozó predikátum
itree(node(L,R)) :-
        itree(L), itree(R).
```

- Az ilyen adattípust **megkülönböztetett unió**nak nevezzük, mert az unióban szereplő halmazokat az elemeik funktora megkülönbözteti (`leaf/1`, `node/2`)

Bináris fák kezelése — fa levele

- Írjunk egy predikátumot annak eldöntésére, hogy egy adott érték szerepel-e egy fa levelében!

```
% fa_levele(Fa, Ertek): A Fa bináris fa levelében szerepel az Ertek szám.
fa_levele(leaf(V), V).      % ha a fa egyetlen levélből áll és a levélbeli
                            % érték megegyezik a keresettel, akkor ''siker''
fa_levele(node(L,_), V) :-
        fa_levele(L, V).    % ha szerepel a bal részfában → az egészben is
fa_levele(node(_,R), V) :-
        fa_levele(R, V).    % ha szerepel a jobb részfában → az egészben is
```

- Az aláhúzásjel egy ún. semmis (void) változó, ennek minden előfordulása különböző változó!
- A 2. és 3. klóz összevonható eggyé, törzsében egy diszjunkcióval:

```
fa_levele(leaf(V), V).          % az egyszerű fa levélértéke szerepel a fában
fa_levele(node(L,R), V) :-      % egy összetett fában szerepel egy érték
        (   fa_levele(L, V)     % ha szerepel a bal részfában
        ;   fa_levele(R, V)     % vagy a jobb részfában
        ).
```

- Példák: ellenőrzés, adott fa leveleinek felsorolása, adott levelű fák felsorolása (∞ keresési tér).

```
| ?- fa_levele(node(node(leaf(1),leaf(2)),leaf(7), 2).  ⟹ yes
| ?- fa_levele(node(node(leaf(1),leaf(2)),leaf(7), 3).  ⟹ no
| ?- fa_levele(node(leaf(1),leaf(7))), E). ⟹ E = 1 ? ; E = 7 ? ; no
| ?- fa_levele(Fa, 3). ⟹ Fa = leaf(3) ? ; Fa = node(leaf(3),_A) ? ;  ...
```

Összetett adatstruktúrák konjunktív és diszjunktív bejárása

- Prologban egy összetett adatstruktúrát kétféleképpen lehet bejárni:
  - konjunktívan: a részek bejárása ÉS kapcsolatban van, általában egy eredményt ad
    - pl. fa összegzése (`sum_tree`), fa ellenőrzése (`itree`), fa kiírása:

```
% faki(Fa): Fa kiírható (mindig teljesül :-). Mellékhatásként kiírja a Fa fát.
faki(leaf(V)) :-
        write(@), write(V).    % A write(X) beépített pred. kiírja az X kifejezést.
faki(node(L,R)) :-
        write('('), faki(L), write(' -- '), faki(R), write(')').

| ?- faki(node(node(leaf(1),leaf(8)),leaf(7))).     ⇒ ((@1 -- @8) -- @7)
yes
```

  - diszjunktívan: a részek bejárása VAGY kapcsolatban van, az eredmény egy részre vonatkozik, visszalépéssel kapjuk a többi eredményt
    - pl. fa elemeinek felsorolása (`fa_levele`)
- A diszjunktív, felsoroló bejárás könnyen kiegészíthető további feltételekkel
  - Keressük egy fának az `(5,10)` intervallumba eső leveleit:

```
| ?- _Fa = node(node(leaf(1),leaf(8)),leaf(7)), fa_levele(_Fa, E), 5 < E, E < 10.
      ⟹ E = 8 ? ; E = 7 ? ; no
| ?- _Fa = (...),  fa_levele(_Fa, E), 5 < E, E < 10, write(E), write(' '), fail.
      ⇒ 8 7  ⟹ no
```


Levél elhagyása bináris fából

- Írjunk egy predikátumot annak eldöntésére, hogy egy adott érték szerepel-e egy összetett fa levelében, és adja vissza a levél elhagyása után fennmaradó fát!

```
% flm(Fa, Ertek, Marad): A Fa összetett bináris fa egy Ertek értékű
% levelének elhagyása után marad a Marad fa. (flm = fa_level_maradek)
flm(node(leaf(V),T), V, T).   % ha a bal részfa a keresett levél
                              % akkor a jobb részfa a maradék
flm(node(T,leaf(V)), V, T).   % ugyanez jobboldali levél esetére
flm(node(L0,R), V, node(L,R)) :-
        flm(L0, V, L).        % ha a bal részfából elhagyható a levél
                              % akkor ennek maradéka, kiegészítve
                              % a jobb részfával, lesz a teljes fa maradéka
flm(node(L,R0), V, node(L,R1)) :-
        flm(R0, V, R1).       % ugyanez jobb részfa esetére
```

- Az `flm/3` predikátum használható ellenőrzése, de fa szétbontására is:

```
| ?- flm(node(leaf(1),node(leaf(2),leaf(3))), 2, T).  ⟹
   T = node(leaf(1),leaf(3)) ? ; no
| ?- flm(node(leaf(1),node(leaf(2),leaf(3))), 7, T).  ⟹ no
| ?- flm(node(leaf(1),node(leaf(2),leaf(3))), X,  T). ⟹
   T = node(leaf(2),leaf(3)), X = 1 ? ;
   T = node(leaf(1),leaf(3)), X = 2 ? ;
   T = node(leaf(1),leaf(2)), X = 3 ? ; no
```


Levél beszúrása bináris fába

- Írjunk egy predikátumot arra, hogy egy adott értékű levelet egy fába minden lehetséges módon beszúrjon!
- Nem kell irnunk, már megírtuk! Az `flm` predikátum erre is jó:

```
% flm(Fa, Ertek, Marad): A Fa összetett bináris fa egy Ertek értékű
% levelének elhagyása után marad a Marad fa. Röviden: Fa - Ertek = Marad.

% flm(Fa, Ertek, Marad): A Fa (összetett) bináris fa úgy áll elő, hogy
% a Marad fába beszúrunk egy E értékű levelet. Fa = Marad + Ertek.
flm(node(leaf(V),T), V, T).   % Egy T fába beszúrhatunk egy levelet
(...)                         % úgy, hogy az egylevelű fát T elé tesszük
```

- Példák:

```
| ?- flm(Fa, 2, leaf(1)), faki(Fa), write(' '), fail.
(@2 -- @1) (@1 -- @2)                          ⟹ no
| ?- flm(Fa0, 2, leaf(1)), flm(Fa, 3, Fa0), faki(Fa), write(' '), fail.
(@3 -- (@2 -- @1)) ((@2 -- @1) -- @3) ((@3 -- @2) -- @1) ((@2 -- @3) -- @1)
(@2 -- (@3 -- @1)) (@2 -- (@1 -- @3)) (@3 -- (@1 -- @2)) ((@1 -- @2) -- @3)
((@3 -- @1) -- @2) ((@1 -- @3) -- @2) (@1 -- (@3 -- @2)) (@1 -- (@2 -- @3))  ⟹ no

negylevelu(X, Y, Z, U, Fa) :-  % Fa az X, Y, Z, U levelekből áll
     flm(Fa0, Y, leaf(X)), flm(Fa1, Z, Fa0), flm(Fa, U, Fa1).

| ?- findall(Fa, negylevelu(1,3,4,6,Fa), Fak), length(Fak,Db). ⟹  Db = 120, Fak = (...)
```


Operátoros jelölés

- A matematikához hasonlóan a Prolog is megengedi az ún operátoros jelölést.
- Számos jel beépített operátor, pl. `+, -, *, /, =, is, =:=, \=, <, =<, ...`
- A felhasználó is definiálhat operátort (részletek később), pl.

```
:- op(500, xfx, --).        % ezután (A -- B) ≡ --(A,B)
:- op(450, fx, @).          % ezután @A ≡ @(A)
```

- Az operátoros jelölést a Prolog **beolvasáskor** átalakítja a belső, kanonikus struktúra-kifejezés formára, kiíráskor a belső alakot visszalakítja operátorossá.
- A `write_canonical` beépített predikátum kiírja egy tetszőleges kifejezés belső alakját:

```
| ?- write_canonical(@1--@2). ⟹ --(@(1),@(2))  vö. node(leaf(1),leaf(2))
| ?- write_canonical(x+2*y).  ⟹ +(x,*(2,y))
```

- Operátorokkal olvashatóbbá tehetjük a programjainkat:

```
sum_tree(@Val, Val).                % sum_tree(leaf(Val), Val).
sum_tree(Left--Right, S) :-         % sum_tree(node(Left,Right), S) :-
       sum_tree(Left, S1),          %        sum_tree(Left, S1),
       sum_tree(Right, S2),         %        sum_tree(Right, S2),
       S is S1+S2.                  %        S is S1+S2.

| ?- sum_tree(@1--(@2--@3), Sum). ⟹ Sum = 6 ? ; no
```

Aritmetika Prologban

- Az operátorok teszik lehetővé azt is, hogy a matematikában ill. más programozási nyelvekben megszokott módon értékelhessünk kis aritmetikai kifejezéseket.
- Az `is` beépített predikátum egy aritmetikai kifejezést vár a jobboldalán (2. argumentumában), azt kiértékeli, és az eredményt egyesíti a baloldali argumentummal
- Az `=:=` beépített predikátum mindkét oldalán aritmetikai kifejezést vár, azokat kiértékeli, és csakkor sikerül, ha az értékek megegyeznek.
- Példák:

```
| ?- X = 1+2, write(X), write('  '), write_canonical(X), Y is X.
⇒            1+2                    +(1,2)    ⟹ X = 1+2, Y = 3 ? ; no
| ?- X = 4, Y is X/2, Y =:= 2.     ⟹ X = 4, Y = 2.0 ? ; no
| ?- X = 4, Y is X/2, Y = 2.       ⟹ no
```

- **Fontos**: az aritmetikai operátorokkal (+,-,...) képzett kifejezések **összetett Prolog kifejezés**t jelentenek. Csak az aritmetikai beépített predikátumok értékelik ki ezeket!
- A Prolog kifejezések alapvetően szimbolikusak, az aritmetikai kiértékelés a „kivétel”.

Klasszikus szimbolikus kifejezés-feldolgozás: deriválás

- Írjunk olyan Prolog predikátumot, amely számokból és az `x` névkonstansból a `+, -, *` műveletekkel képzett kifejezések deriválását elvégzi!

```
% deriv(Kif, D):  Kif-nek az x szerinti deriváltja D.
deriv(x, 1).
deriv(C, 0) :-                      number(C).
deriv(U+V, DU+DV) :-                deriv(U, DU), deriv(V, DV).
deriv(U-V, DU-DV) :-                deriv(U, DU), deriv(V, DV).
deriv(U*V, DU*V + U*DV) :-          deriv(U, DU), deriv(V, DV).

| ?- deriv(x*x+x, D).
  ⟹  D = 1*x+x*1+1 ? ; no

| ?- deriv((x+1)*(x+1), D).
  ⟹  D = (1+0)*(x+1)+(x+1)*(1+0) ? ; no

| ?- deriv(I, 1*x+x*1+1).
  ⟹  I = x*x+x ? ; no

| ?- deriv(I, 0).
  ⟹  no
```


Példa: adott értékű kifejezés előállítása

- A feladat: írjunk Prolog programot a következő feladvány megoldására:
  - Az 1, 3, 4, 6 számokból a négy alapművelet felhasználásával állítsuk elő a 24 számértéket!
  - Mind a négy számot fel kell használni, tetszőleges sorrendben.
  - Tetszőleges alapműveletek használhatók, tetszőlegesen zárójelezéssel.
- Már van egy predikátumunk (`negylevelu/5`), amely adott számokból tetszőleges fát épít.
- Definiáljunk egy predikátumot, amely egy fának megfelelő aritmetikai kifejezéseket csinál!

```
% fa_kif(Fa, Kif): Kif a Fa faval azonos alakú, azonos számokból álló
% aritmetikai kifejezés, amelyben a négy alapművelet fordulhat elö.
fa_kif(leaf(V), V).
fa_kif(node(L,R), Exp) :-
        fa_kif(L, E1),
        fa_kif(R, E2),
        alap4(E1, E2, Exp).

% alap4(X, Y, Kif): Kif az X és Y kifejezésekböl a négy alapművelet egyikével áll elö.
alap4(X, Y, X+Y).                       alap4(X, Y, X-Y).
alap4(X, Y, X*Y).                       alap4(X, Y, X/Y).

| ?- fa_kif(node(leaf(1),node(leaf(2),leaf(3))), Kif).
Kif = 1+(2+3) ? ; Kif = 1-(2+3) ? ; Kif = 1*(2+3) ? ; Kif = 1/(2+3) ? ;
(...)
Kif = 1+2/3 ?   ; Kif = 1-2/3 ?   ; Kif = 1*(2/3) ? ; Kif = 1/(2/3) ? ; no
```


Példa: adott értékű kifejezés előállítása (folyt.)

- Korábban elkészített predikátumok:
  - adott számokból álló fákat felsoroló `negylevelu/5`
  - adott fával azonos szerkezetű aritmetikai kifejezéseket felsoroló `fa_kif/2`
- Ezekre építve könnyen megírható a feladvány megoldására használható predikátum:

```
% Kif egy a négy alapművelettel az X, Y, Z, U számokból
% felépített kifejezés, amelynek értéke Ertek.
negylevelu_erteke(X, Y, Z, U, Ertek, Kif) :-
        negylevelu(X, Y, Z, U, Fa),
        fa_kif(Fa, Kif),
        Kif =:= Ertek.

| ?- negylevelu_erteke(1,3,4,6,24,Kif).
Kif = 6 ? (1 ? 3 ? 4)  ? ;
no
```

LEGALAPVETŐBB BEÉPÍTETT PREDIKÁTUMOK

Aritmetikai és egyesítési beépített predikátumok

- Aritmetikai predikátumok
  - `X is Kif`: A `Kif` **aritmetikai** kifejezés értékét egyesíti `X`-szel
  - `Kif1<Kif2, Kif1=<Kif2, Kif1>Kif2, Kif1>=Kif2, Kif1=:=Kif2, Kif1=\=Kif2`: A `Kif1` és `Kif2` aritmetikai kifejezések értéke a megadott relációban van egymással (`=:=` ⇒aritmetikai egyenlő, `=\=` ⇒aritmetikai nem-egyenlő).
  - Ha `Kif, Kif1, Kif2` valamelyike nem **tömör** aritmetikai kifejezés ⇒hiba.
  - Legfontosabb aritmetikai operátorok: `+, -, *, /, mod, //` (egész-osztás)
- Egyesítés: `X=Y`: Az `X` és `Y` **általános** Prolog kifejezések egyesíthetőek (és az egyesítést végre is hajtja). `X\=Y`: `X` és `Y` nem egyesíthető.
- Példák:

```
| ?- X is 1*2+3.      ⟹   X = 5 ?
| ?- X = 1+2*3.       ⟹   X = 1+2*3 ?
| ?- 5 \= 1+2*3.      ⟹   yes
| ?- 5 =\= 1+2*3.     ⟹   no
| ?- X is alma.       ⟹   ! Domain error in argument 2 of is/2
| ?- X =:= 1*2+3.     ⟹   ! Instantiation error in argument 1 of =:= /2
| ?- 1+2*3 > 2*3+1.   ⟹   no
```


Programfejlesztési beépített predikátumok

- `consult(File)` vagy `[File]`: A `File` állományban levő programot beolvassa és értelmezendő alakban eltárolja. (`File = user` ⇒ terminálról olvas.)
- `listing` vagy `listing(Predikátum)`: Az értelmezendő alakban eltárolt összes ill. adott nevű predikátumokat kilistázza.
- `compile(File)`: A `File` állományban levő programot beolvassa, lefordítja.
- A lefordított alak gyorsabb, de nem listázható, **kicsit** kevésbe pontosan nyomkövethető.
- `halt`: A Prolog rendszer befejezi működését.
- Példák:

```
> sicstus
SICStus 3.10.0 (x86-linux-glibc2.1): Tue Dec 17 15:12:52 CET 2002
| ?- consult(fakt).
% consulted /home/user/fakt.pl in module user, 0 msec 712 bytes
| ?- listing(fakt).
fakt(0, 1).
fakt(A, B) :-
        A>0, C is A-1, fakt(C, D), B is D*A.
| ?- halt.
>
```


Kiíró és egyéb predikátumok

- `write(X)`: Az `X` Prolog kifejezést kiírja (ha kell, operátorokkal).
- `write_canonical(X)`: Az `X` Prolog kifejezést kanonikus alapstruktúra-alakban kiírja.
- `nl`: Kiír egy újsort.
- `true, fail`: Mindig sikerül ill. mindig meghiúsul.
- `trace, notrace`: A (teljes) nyomkövetést be- ill. kikapcsolja.
- `spy Predikátum`: Töréspontot helyez a `Predikátum`-ra.
- Példák:

```
| ?- write(+(1,*(2,3))), write('  '), write_canonical(1+2*3), nl.
1+2*3  +(1,*(2,3))
yes
| ?- :- szuloje('István', X), write(X), write(' '), fail ; true.
Géza Sarolt
```

- A Prolog interaktív felületén használható `'?-'` és `':-'` előtagokról:
  - `'?-'` — **kérdés**: (alapértelmezés, elhagyható): futtasd le a célsorozatot, írd ki a változó-behelyettesítéseket, ha van változó, kérdezz a további megoldásról!
  - `':-'` — **direktíva**: futtasd a célsorozatot, első megoldásig, csak meghiúsuláskor szólj!