# Deklaratív Programozás

Hanák Péter
hanak@inf.bme.hu

Irányítástechnika és Informatika Tanszék

(OM Kutatás-Fejlesztési Helyettes Államtitkárság)

Szeredi Péter, Benkő Tamás
{szeredi,benko}@iqsoft.hu

Számítástudományi és Információelméleti Tanszék

(IQSOFT Intelligens Software Rt.)

# KÖVETELMÉNYEK — TUDNIVALÓK

# Deklaratív programozás: tudnivalók

## Honlap, levelezési lista

- Honlap: <http://www.inf.bme.hu/~dp>
- Levlista: <http://www.inf.bme.hu/mailman/listinfo/dp-l>. Csak a feliratkozottak küldhetnek levelet a <dp-l@inf.bme.hu> címre.

## Jegyzet

- Szeredi Péter, Benkő Tamás: Deklaratív programozás. Bevezetés a logikai programozásba.
- Hanák D. Péter: Deklaratív programozás. Bevezetés a funkcionális programozásba.
- Új, bővített kiadások, kötetenként 600-800 Ft, terjedelemtől függően
- Előző kiadások a honlapon (ps, pdf)
- Jegyzetrendelés: a honlapon megadandó módon

# Deklaratív programozás: tudnivalók (folyt.)

## Fordító- és értelmezőprogramok

- SICStus Prolog (3.9, licenszköteles, aláírás ellenében jelszót adunk várhatóan 2002. február 18.-tól)
- 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 készül (ld. honlap)
- Kézikönyvek HTML-változatban (MOSML pdf is)
- Más programok: swiProlog, gnuProlog 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, 8-10 oldalas fejlesztői dokumentáció (TXT, TeX/LaTeX, HTML, PDF, PS; de nem DOC vagy RTF)
- Kiadás az 5.-6. héten, a honlapon, letölthető keretprogrammal
- Beadás a 13. héten; elektronikus levélben (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.
- 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 nyelvenkét összesen max 7,5 pont szerezhető
- A NHF súlya az osztályzatban: 15% (100 pontból 15)

# Deklaratív programozás: félévközi követelmények (folyt.)

## Kis házi feladatok (KHF)

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

## Gyakorló feladatok

- Nem kötelező, de a sikeres ZH-hoz, vizsgához *elengedhetetlen*!
- Gyakorlás a honlapon keresztül

# Deklaratív programozás: félévközi követelmények (folyt.)

## Nagyzárthelyi, pótzárthelyi (NZH, PZH)

- NZH a 7. oktatási héten, március 25-én.
- Kötelező!
- Semmilyen jegyzet, segédlet nem használható
- A megtanulandó könyvtári függvények, ill. eljárások listáját előre megadjuk
- 40%-os szabály (nyelvenként a max. részpontszám 40%-a kell az eredményességhez)
- PZH a 12. oktatási héten
- Súlya az osztályzatban: 15%

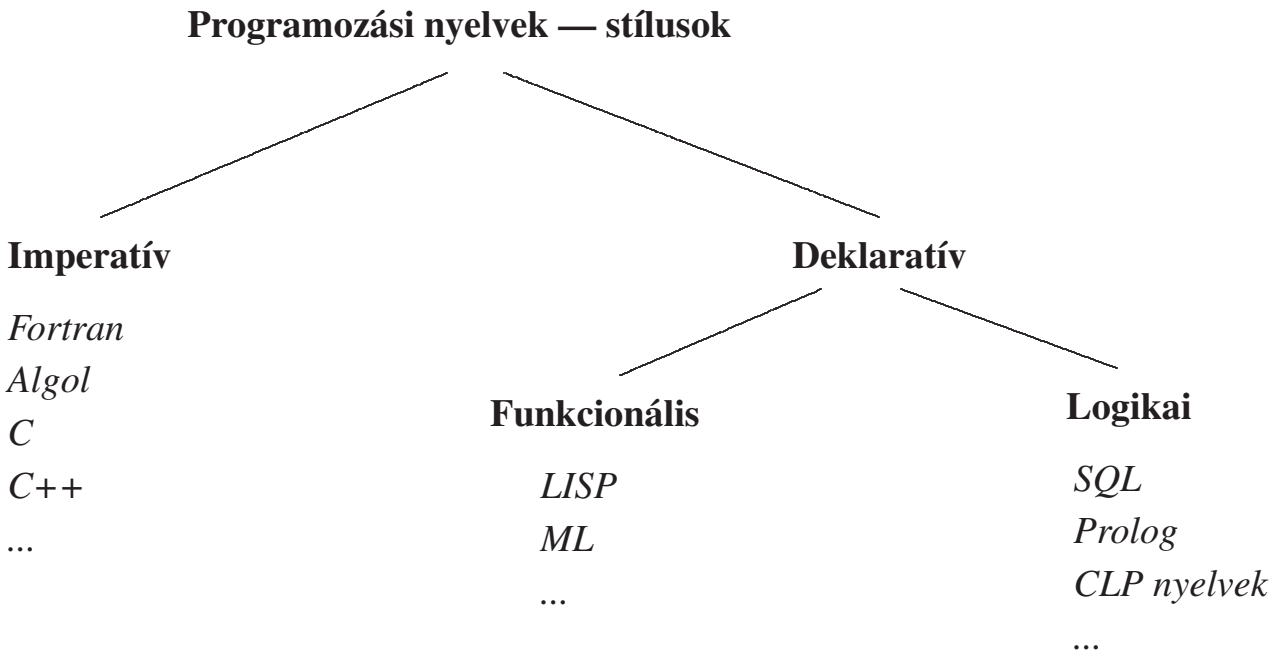
# Deklaratív programozás: vizsga

## Vizsga

- Szóbeli, felkészülés írásban
- Prolog, SML: több kisebb feladat, kétszer 35 pontért (programírás, -elemzés)
- Semmilyen jegyzet, segédlet nem használható
- 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

# DEKLARATÍV ÉS IMPERATÍV PROGRAMOZÁS

# Programozási nyelvek osztályozása



# Imperatív és deklaratív programozási nyelvek

- Imperatív program
  - felszólító módú, utasításokból áll
  - változó: változtatható értékű memóriahely
  - Példa: `int fakt(int n) {int f=1; while (n>1) f*=n--; return f;}`
- Deklaratív program
  - kijelentő módú, egyenletekből, állításokból áll
  - változó: egy ismeretlen, de (előbb–utóbb) rögzített értékű mennyiség
  - SML példa: `fun fakt 0 = 1 | fakt n = n * fakt (n-1);`
  - C példa: `int fakt(int n) {if (n<=1) return 1; else return n*fakt(n-1);}`
- Deklaratív nyelvek jelszavai
  - MIT és nem HOGYAN (WHAT rather than HOW): a *megoldás módja* helyett inkább a megoldandó *feladat leírását* kell megadni
  - Egyszeres értékadás (single assignment) — párhuzamos végrehajthatóság

# 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
  2    from szulok fiatal, szulok oreg
  3    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: leképezések komponálásával
  - 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`), amely két fára mutat (`left,right`)
  - vagy egy levél (`leaf`), amely egy egészet 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:
  - egy csomópont esetén a két részfa levélösszegének összege
  - egy levél esetén a levélben tárolt egész
- Binárisfa-összegzők különböző nyelveken

```
% C nyelvű 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 1.44 (August 1999)
Enter `quit();' to quit.
- use "tree.sml";
[opening file "tree.sml"]
> datatype Tree
  con Node = fn : Tree * Tree -> Tree
  con Leaf = fn : int -> Tree
  val sum_tree = fn : Tree -> int
[closing file "tree.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 3.8.7 (x86-linux-glibc2.1): Tue Oct 23 17:40:50 CEST 2001
Licensed to BME 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: _76 is _73+_74 - arg 2
| ?- 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
  - a programrészek (relációk, függvények) önálló jelentéssel bírnak
  - a kód és a jelentés összevethető: program-verifikáció
- 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/LP rövid történeti áttekintése

| | |
|---|---|
| 1960-as évek | Tételbizonyító programok |
| 1970-72 | A logikai programozás elméleti alapjai (R A Kowalski) |
| 1972 | Az első Prolog interpreter (A Colmerauer) |
| 1975 | A második Prolog interpreter (Szeredi P) |
| 1977 | Az első Prolog fordítóprogram (D H D Warren) |
| 1977–79 | Számos kisérleti Prolog alkalmazás Magyarországon |
| 1981 | A japán 5. generációs projekt a logikai programozást választja |
| 1982 | A magyar MProlog az egyik első kereskedelmi forgalomba kerülő Prolog megvalósítás |
| 1983 | Egy új fordítási modell és absztrakt Prolog gép (WAM) megjelenése (D H D Warren) |
| 1986 | Prolog szabványosítás kezdete |
| 1987–89 | Új logikai programozási nyelvek (CLP, Gödel, stb.) |
| 1990–. . . | Prolog megjelenése párhuzamos számítógépeken |
| | Nagyhatékonyságú Prolog fordítóprogramok |
| | ..... |

# Információk a logikai programozásról

- Prolog megvalósítások:
  - SWI Prolog: `http://www.swi.psy.uva.nl/projects/SWI-Prolog/`
  - SICStus Prolog: `http://www.sics.se/ps/sicstus/sicstus_toc.html`
  - GNU Prolog: `http://pauillac.inria.fr/~diaz/gnu-prolog/`
- Hálózati információforrások:
  - The WWW Virtual Library: Logic Programming: `http://www.comlab.ox.ac.uk/archive/logic-prog.html`
  - CMU Prolog Repository: (a `http://www.cs.cmu.edu/afs/cs/project/ai-repository/ai/lang/prolog/` címen belül)
    - Főlap: `0.html`
    - Prolog FAQ: `faq/prolog.faq`
    - Prolog Resource Guide: `faq/prg_1.faq`, `faq/prg_2.faq`

# Magyar nyelvű Prolog irodalom

**Farkas Zsuzsa, Futó Iván, Langer Tamás, Szeredi Péter:**
Az MProlog programozási nyelv.
Műszaki Könyvkiadó, 1989

**Márkusz Zsuzsa:** Prologban programozni könnyű.
Novotrade, 1988

**Futó Iván (szerk.):** Mesterséges intelligencia. (9.2 fejezet, Szeredi Péter)
Aula Kiadó, 1999

# A PROLOG NYELV KÖZELÍTŐ SZINTAXISA

# Predikátumok, klózok

```
% két klózból álló predikátum definíciója, funktora: sum_tree/2
sum_tree(leaf(Val), Val).              %                    1. klóz, tényállítás
sum_tree(node(Left,Right), S) :-       %              fej   \
        sum_tree(Left, S1),            % cél          \      |
        sum_tree(Right, S2),           % cél          | törzs | 2. klóz, szabály
        S is S1+S2.                    % cél          /      /
```

⟨ Prolog program ⟩ ::= ⟨ predikátum ⟩ ...
⟨ predikátum ⟩ ::= ⟨ klóz ⟩ ... {azonos funktorú}
⟨ klóz ⟩ ::= ⟨ tényállítás ⟩.␣ |
⟨ szabály ⟩.␣
⟨ tényállítás ⟩ ::= ⟨ fej ⟩
⟨ szabály ⟩ ::= ⟨ fej ⟩ :- ⟨ törzs ⟩
⟨ törzs ⟩ ::= ⟨ cél ⟩, ...
⟨ cél ⟩ ::= ⟨ kifejezés ⟩
⟨ fej ⟩ ::= ⟨ kifejezés ⟩

# A Prolog szemléletmódjai

- A Prolog nyelv terminológiája többféle szemléletből, értelmezésből származik.

| Logikai (tételbizonyítási) | Procedurális (eljárásszervezési) | Célvezérelt keresés |
|---|---|---|
| predikátum | eljárás | |
| klóz | (eljárás-változat) | szabály, tényállítás |
| (pozitív literál) | (eljárás)fej | |
| (negatív literálok) | (eljárás)törzs | célsorozat |
| (negatív literál) | (eljárás)hívás | cél |

# Prolog kifejezések

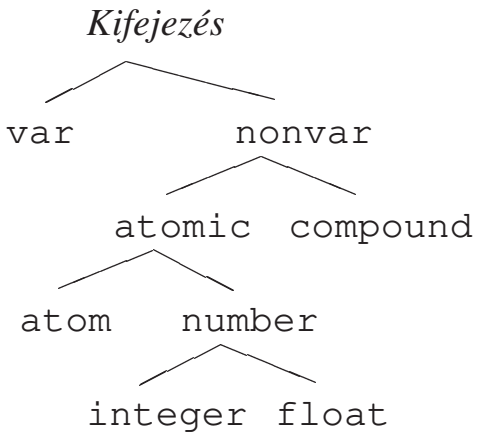
```
%     sum_tree(node(Left,Right), S)    % összetett kif., funktora sum_tree/2
%        |          |              |
% struktúranév      |             argumentum, változó
%                  \- argumentum, összetett kif.
```

| | | | |
|---|---|---|---|
| ⟨ kifejezés ⟩ | ::= | ⟨ változó ⟩ \| | {var} |
| | | ⟨ konstans ⟩ \| | {atomic} |
| | | ⟨ összetett kifejezés ⟩ | {compound} |
| ⟨ konstans ⟩ | ::= | ⟨ névkonstans ⟩ \| | {atom} |
| | | ⟨ számkonstans ⟩ | {number} |
| ⟨ számkonstans ⟩ | ::= | ⟨ egész szám ⟩ \| | {integer} |
| | | ⟨ lebegőp. szám ⟩ | {float} |

| | | |
|---|---|---|
| ⟨ összetett kifejezés ⟩ | ::= | ⟨ struktúranév ⟩ ( ⟨ argumentum ⟩, . . . ) |
| ⟨ struktúranév ⟩ | ::= | ⟨ névkonstans ⟩ |
| ⟨ argumentum ⟩ | ::= | ⟨ kifejezés ⟩ |

# Kifejezések osztályozása

- Kifejezés-osztályok fastruktúrája — osztályozó beépített eljárások




| | |
|---|---|
| `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 |

- Kifejezések funktora (principal functor)
  - összetett kifejezés funktora = struktúranév/argumentumszám, pl. `sum_tree/2`
  - konstans funktora = konstans/0, pl. `'István'/0`
  - változónak nincs funktora

# Lexikai elemek

```
% változó:          Fakt FAKT _fakt X2 _2 _
% névkonstans:      fakt ≡ 'fakt' 'István' [] ; ',' += ** \= ≡ '\\='
% számkonstans:     0 -123 10.0 -12.1e8
% nem (egyetlen) névkonstans:  !=, Istvan
% nem (egyetlen) számkonstans: 1e8 1.e2
```

⟨ változó ⟩ ::= ⟨ nagybetű ⟩⟨ alfanum ⟩...|
_ ⟨ alfanum ⟩...|

⟨ névkonstans ⟩ ::= ' ⟨ névkar ⟩... ' |
⟨ kisbetű ⟩⟨ alfanum ⟩...|
⟨ tapadó jel ⟩...| ! | ; | [] | {}

⟨ névkar ⟩ ::= {tetszőleges nem ' és nem \ karakter} |
\ ⟨ escape szekvencia ⟩

⟨ alfanum ⟩ ::= ⟨ kisbetű ⟩ | ⟨ nagybetű ⟩ | ⟨ számjegy ⟩ | _

⟨ tapadó jel ⟩ ::= + | − | * | / | \ | $ | ^ | < | > | = | ` | ~ | : | . | ? | @ | # | &

⟨ egész szám ⟩ ::= {előjeles vagy előjeltelen számjegysorozat}

⟨ lebegőp. szám ⟩ ::= {belsejében tizedespontot tartalmazó
számjegysorozat esetleges exponenssel}

# Prolog adatstruktúrák, típusok

- Típusok Prologban
  - A Prolog típustalan nyelv
  - De: a Prolog eljárások is adathalmazokon, azaz típusokon értelmezettek
  - Például a korábbi `sum_tree/2` által kezelt fastruktúrák:

    `tree` $\equiv \{$ `leaf(`$i$`)` $\mid$ $i \in$ `integer` $\} \cup \{$ `node(`$l$`,`$r$`)` $\mid$ $l, r \in$ `tree` $\}$
  - A fenti típus két, ekvivalens leírási módja:

```
% :- type tree == {leaf(integer)} \/ {node(tree, tree)}.
% :- type tree ---> leaf(integer) | node(tree, tree).
```

  - Nincs típushiba, csak meghiúsulás!

# Megkülönböztetett és nem-megkülönböztetett úniók

- Megkülönböztetett únió:
  - véges sok, különböző funktorú halmaz uniója
  - például: `:- type tree == {leaf(integer)} \/ {node(tree, tree)}.`
  - a fenti típusban `leaf/1` és `node/2` a különböző funktorok.
- Nem-megkülönböztetett únió
  - Prologban megengedett, pl.

    ```
    % :- type tree2 == integer \/ {node(tree2, tree2)}.
    ```

  - Osztályozó eljárások segítségével ágazhatunk el:

    ```
    sum_tree2(Tree, S) :-
        integer(Tree), S = Tree.
    sum_tree2(Tree, S) :-
        Tree = node(Left,Right),
        sum_tree2(Left, S1), sum_tree2(Right, S2), S is S1+S2.
    ```

  - Egyszerűbb adatábrázolást jelent, pl. `node(5,node(3,2))`

# Szintaktikus édesítőszer: operátorok

```
% S is S1+S2 ekvivalens az is(S, +(S1,S2)) kifejezéssel
```

## Operátor-deklaráció

- `:-` `op` (⟨ **prioritás** ⟩, ⟨ **fajta** ⟩, ⟨ **operátornév** ⟩).
- ⟨ operátornév ⟩ tetszőleges névkonstans
- ⟨ prioritás ⟩ 0–1200 közötti egész (nagyobb prioritás *gyengébben* köt :-)
- ⟨ fajta ⟩
  - infix: `yfx`, `xfy`, `xfx`; `A op B` ≡ `op(A, B)`
  - prefix: `fx`, `fy`; `op A` ≡ `op(A)`
  - postfix: `xf`, `yf`; `A op` ≡ `op(A)`
- a ⟨ fajta ⟩-ban `x` és `y` az asszociativitást határozzák meg:
  - `x`: az adott oldalon nem állhat azonos prioritású operátor zárójelezetlenül
  - `y`: az adott oldalon állhat azonos prioritású operátor zárójelezetlenül
  - Tehát: `xfy` jobbról-balra, `yfx` balról-jobbra zárójelez.

# Beépített operátorok

## Szabványos operátorok

| | | |
|---|---|---|
| 1200 | xfx | :-, --> |
| 1200 | fx | :-, ?- |
| 1100 | xfy | ; |
| 1050 | xfy | -> |
| 1000 | xfy | ',' |
| 900 | fy | \+ |
| 700 | xfx | < = \= =.. |
| | | =:= =< == \== |
| 700 | xfx | =\= > >= is |
| | | @< @=< @> @>= |
| 500 | yfx | + - /\ \/ |
| 400 | yfx | * / // rem |
| | | mod[1] << >> |
| 200 | xfx | ** |
| 200 | xfy | ^ |
| 200 | fy | -[2], \ |

## Egyéb beépített operátorok

| | | |
|---|---|---|
| 1150 | fx | dynamic multifile |
| | | block meta_predicate |
| 900 | fy | spy nospy |
| 550 | xfy | : |
| 500 | yfx | # |
| 500 | fx | +[3] |

[1] `sicstus` módban 300 `xfx` operátor
[2] `sicstus` módban 500 `fx` operátor
[3] `iso` módban 200 `fy` operátor

# Operátorok felhasználása

- Mire jók az operátorok?
  - aritmetikai kifejezések leírására, pl. `X is (Y+3) mod 4`
  - klózok leírására (`:-` és `','` is operátor)
  - klózok átadhatók meta-eljárásoknak, pl `asserta( (p(X):-q(X),r(X)) )`
  - eljárásfejek, eljáráshívások olvashatóbbá tételére:

```
:- op(800, xfx, [nagyszülője, szülője]).

Gy nagyszülője N :-  Gy szülője Sz, Sz szülője N.
```

  - adatstruktúrák olvashatóbbá tételére, pl.

```
:- op(100, xfx, [.]).

sav(kén, h.2-s-o.4).
```

- Miért rosszak az operátorok?
  - egyetlen globális erőforrás, ez nagyobb projektben gondot okozhat.

# Binárisfa-összegző — operátoros változat

```
:- op(500, xfx, --).

% :- type tree3 == integer \/ {tree3--tree3}.

sum_tree3(Left--Right, S) :-
        sum_tree3(Left, S1),
        sum_tree3(Right, S2),
        S is S1+S2.
sum_tree3(Tree, S) :-
        integer(Tree), S = Tree.

| ?- sum_tree3(5--(3--2), Sum).

Sum = 10 ? ;
```

# PROLOG SZEMANTIKA

# A Prolog deklaratív szemantikája

- Egy program és egy rá vonatkozó kérdés jelentése a következő:
  - Minden klóz egy logikai állításnak felel meg, pl.:

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

    logikai formája:

    $\forall U \forall N \forall Sz(\texttt{nagyszuloje}(U, N) \leftarrow \texttt{szuloje}(U, Sz) \wedge \texttt{szuloje}(Sz, N))$
  - A feltett kérdésnek (célsorozatnak) egy bizonyítandó állítás felel meg, pl.

    ```
    | ?- nagyszuloje('Imre', N).
    ```

    logikai formája:

    $\exists N(\texttt{nagyszuloje}(\texttt{'Imre'}, N))$
  - A kérdésre adott válasz: a benne szereplő változók egy olyan behelyettesítése, amely esetén a célsorozat logikai következménye lesz a programnak, pl.

    ```
    N = 'Géza'
    ```
- A deklaratív szemantika teljesen nem valósítható meg:
  - az összes következmény nem biztos, hogy előállítható

# A Prolog procedurális szemantikája

- A procedurális szemantika:
  - egy adott Prolog programra vonatkozó kérdés végrehajtásának pontos leírása,
  - egy nagyon leegyszerűsített tételbizonyítási algoritmus (SLD rezolúció)
- A procedurális szemantika alapelemei:
  - mintaillesztésen (egyesítésen) alapuló eljáráshívási mechanizmus,
  - visszalépéses mélységi keresés.
- A Prolog eljáráshívás alaplépése, az ún. redukciós lépés:
  - megkeressük az első olyan klózt, amelynek feje az első céllal egyesíthető
  - a kiválasztott klóz törzsét az első cél helyébe rakjuk
- A kétféle szemantika kapcsolata (“tiszta” Prolog programokra)
  - a Prolog végrehajtás által előállított megoldás biztosan logikai következmény
  - de nem biztos, hogy minden következmény előáll (hiba, végtelen ciklus)

# Prolog végrehajtási példa

```
sum_tree3(Left--Right, S) :-                                    % (1)
        sum_tree3(Left, S1), sum_tree3(Right, S2), S is S1+S2.
sum_tree3(Tree, S) :-
        integer(Tree), S = Tree.                                % (2)

% Kezdeti célsorozat:
sum_tree3(3--2,A), write(A)
% Redukciós lépés az (1) klózzal
(1) > sum_tree3(3,B), sum_tree3(2,C), A is B+C, write(A)
(2) > integer(3), B=3, sum_tree3(2,C), A is B+C, write(A)
% Redukciós lépés beépített eljárással (BIP = built-in predicate)
BIP > B=3, sum_tree3(2,C), A is B+C, write(A)
BIP > sum_tree3(2,C), A is 3+C, write(A)
(2) > integer(2), C=2, A is 3+C, write(A)
BIP > C=2, A is 3+C, write(A)
BIP > A is 3+2, write(A)
BIP > write(5)                                  % ===> 5
BIP > []
```

# A végrehajtás alapelemei: egyesítés

- Az eljáráshívás és egy klózfej azonos alakra hozása, változók behelyettesítésével
- Példák
  - Bemenő paraméterátadás:
    hívás: `nagyszuloje('Imre', Nsz),`
    fej: `nagyszuloje(Gy, N),`
    behelyettesítés: `Gy = 'Imre', N = Nsz`
  - Kimenő paraméterátadás:
    hívás: `szuloje('Imre', Sz),`
    fej: `szuloje('Imre', 'István'),`
    behelyettesítés: `Sz = 'István'`
  - Bemenő/kimenő paraméterátadás:
    hívás: `sum_tree(leaf(5), Sum)`
    fej: `sum_tree(leaf(V), V)`
    behelyettesítés: `V = 5, Sum = 5`

# Egyesítés: a behelyettesítés fogalma

- A behelyettesítés
  - Egy függvény, amely változókhoz kifejezéseket rendel.
  - Pl. $\sigma = \{$`X`$\leftarrow$`a`, `Y`$\leftarrow$`s(b,B)`, `Z`$\leftarrow$`C`$\}$ `X`-hez `a`-t, `Y`-hoz `s(b,B)`-t stb. rendel.
  - $K\sigma$: $\sigma$ alkalmazása $K$ kif.-re, pl. `f(g(Z,h),A,Y)`$\sigma =$ `f(g(C,h),A,s(b,B))`
  - Két behelyettesítés kompozíciója (függvénykompozíció):

$$\sigma \otimes \theta = \{\, x \leftarrow x\sigma\theta \mid x \in D(\sigma)\} \bigcup \{\, x \leftarrow x\theta \mid x \in D(\theta) \backslash D(\sigma) \,\}$$

  - $\sigma$ általánosabb mint $\theta$, ha létezik olyan $\rho$, hogy $\theta = \sigma \otimes \rho$
- Legáltalánosabb egyesítő ($mgu$ — most general unifier)
  - $A$ és $B$ kifejezések egyesíthetőek ha létezik egy olyan $\sigma$ behelyettesítés, hogy $A\sigma = B\sigma$. Ezt a $\sigma$ behelyettesítést $A$ és $B$ egyesítőjének nevezzük.
  - $A$ és $B$ legáltalánosabb egyesítője $\sigma$ ($mgu(A, B) = \sigma$), ha $\sigma$ $A$ és $B$ minden egyesítőjénél általánosabb (Tétel: átnevezéstől eltekintve egyértelmű.)
  - Tétel: változó-átnevezéstől eltekintve az $mgu$ egyértelmű.

# Az egyesítési algoritmus

- Az egyesítési algoritmus
  - bemenete: két Prolog kifejezés: $A$ és $B$
  - feladata: a két kifejezés egyesíthetőségének eldöntése
  - eredménye: sikeresség esetén a legáltalánosabb egyesítő ($mgu(A, B)$) előállítása.
- Az egyesítési algoritmus, $\sigma = mgu(A, B)$ előállítása
  1. Ha $A$ és $B$ azonos változók vagy konstansok, akkor $\sigma = \emptyset$.
  2. Egyébként, ha $A$ változó, akkor $\sigma = \{A \leftarrow B\}$.
  3. Egyébként, ha $B$ változó, akkor $\sigma = \{B \leftarrow A\}$.
  4. Egyébként, ha $A$ és $B$ azonos nevű és argumentumszámú összetett kifejezések és argumentum-listáik $A_1,\ldots,A_N$ ill. $B_1,\ldots,B_N$, és
     a. $A_1$ és $B_1$ legáltalánosabb egyesítője $\sigma_1$,
     b. $A_2\sigma_1$ és $B_2\sigma_1$ legáltalánosabb egyesítője $\sigma_2$,
     c. $A_3\sigma_1\sigma_2$ és $B_3\sigma_1\sigma_2$ legáltalánosabb egyesítője $\sigma_3$,
     d. ...

     akkor $\sigma = \sigma_1 \otimes \sigma_2 \otimes \sigma_3 \otimes \ldots$.
  5. Minden más esetben a $A$ és $B$ nem egyesíthető.

# Egyesítési példák

- $A$ = `sum_tree(leaf(V), V)`, $B$ = `sum_tree(leaf(5), S)`
  - (4.) $A$ és $B$ neve és argumentumszáma megegyezik
    - (a.) `mgu(leaf(V), leaf(5))` (4., majd 2. szerint) $= \{$`V`$\leftarrow$`5`$\} = \sigma_1$
    - (b.) `mgu(V`$\sigma_1$`, S)` = `mgu(5, S)` (3. szerint) = $\{$`S`$\leftarrow$`5`$\} = \sigma_2$
  - tehát $mgu(A, B) = \sigma_1 \otimes \sigma_2 = \{$`V`$\leftarrow$`5`, `S`$\leftarrow$`5`$\}$
- $A$ = `node(leaf(X), T)`, $B$ = `node(T, leaf(3))`
  - (4.) $A$ és $B$ neve és argumentumszáma megegyezik
    - (a.) `mgu(leaf(X), T)` (3. szerint) $= \{$`T`$\leftarrow$`leaf(X)`$\} = \sigma_1$
    - (b.) `mgu(T`$\sigma_1$`, leaf(3))` = `mgu(leaf(X), leaf(3))` (4, majd 2. szerint) = $\{$`X`$\leftarrow$`3`$\} = \sigma_2$
  - tehát $mgu(A, B) = \sigma_1 \otimes \sigma_2 = \{$`T`$\leftarrow$`leaf(3)`, `X`$\leftarrow$`3`$\}$

# Egyesítési példák a gyakorlatban

- Az `= /2` beépített eljárás egyesíti a két argumentumát
- `= /2` definíciója: `X = X`, azaz `=(X, X).`
- Példák:

```
| ?- 3--(4--5) = Left--Right.
        Left = 3, Right = 4--5 ?
| ?- node(leaf(X), T) = node(T, leaf(3)).
        T = leaf(3), X = 3 ?
| ?- X*Y = 1+2*3.
        no
| ?- f(X, 3/Y-X, Y) = f(U, B-a, 3).
        B = 3/3, U = a, X = a, Y = 3 ?
| ?- f(f(X), U+2*2) = f(U, f(3)+Z).
        U = f(3), X = 3, Z = 2*2 ?
```

# Az egyesítés kiegészítése: előfordulás-ellenőrzés (*occurs check*)

- Kérdés: `X` és `s(X)` egyesíthető-e?
  - A matematikai válasz: *nem*, egy változó nem egyesíthető egy olyan struktúrával, amelyben előfordul (ez az előfordulás-ellenőrzés).
  - Az ellenőrzés költséges, ezért alaphelyzetben nem alkalmazzák.
  - Szabványos eljárásként rendelkezésre áll: `unify_with_occurs_check/2`
  - Kiterjesztés (pl. SICStus): az előfordulás-ellenőrzés elhagyása miatt keletkező ciklikus kifejezések tisztességes kezelése.

- Példák:

```
| ?- X = s(1,X).
        X = s(1,s(1,s(1,s(1,s(...))))) ?
| ?- unify_with_occurs_check(X, s(1,X)).
        no
| ?- X = s(X), Y = s(s(Y)), X = Y.
        X = s(s(s(s(s(...))))), Y = s(s(s(s(s(...))))) ?
```

# A végrehajtás alapelemei: redukciós lépés

- Redukciós lépés: egy célsorozat redukálása egy újabb célsorozattá egy klóz segítségével
- A redukciós lépés végrehajtása:
  - A klózt lemásoljuk, minden változót szisztematikusan új változóra cserélve.
  - A célsorozatot szétbontjuk az első hívásra és a maradékra.
  - Ha az első hívás felhasználói eljárásra vonatkozik:
    - Az első hívást egyesítjük a klózfejjel
    - A szükséges behelyettesítéseket elvégezzük a klóz törzsén és a célsorozatot maradékán
    - Az új célsorozat: a klóztörzs és utána a maradék célsorozat
    - Ha a hívás és a klózfej nem egyesíthető, akkor a redukciós lépés meghiúsul.
  - Ha az első hívás beépített eljárásra vonatkozik:
    - A beépített eljáráshívást végrehajtjuk.
    - Ez lehet sikeres (változó-behelyettesítésekkel), vagy lehet sikertelen.
    - Siker esetén a behelyettesítéseket elvégezzük a célsorozatot maradékán.
    - Az új célsorozat: az első hívás elhagyása után fennmaradó maradék célsorozat.
    - Ha a beépített eljárás hívása sikertelen, akkor a redukciós lépés meghiúsul.

# A Prolog végrehajtási algoritmus — első közelítés

- A Prolog végrehajtás:
  - egy adott célsorozat futtatása egy adott programra vonatkozóan,
  - eredménye lehet:
    - siker — változó-behelyettesítésekkel
    - meghiúsulás (változó-behelyettesítések nélkül)
- Egy célsorozat végrehajtása:
  1. Ha az első hívás beépített eljárásra vonatkozik, végrehajtjuk a redukciós lépést.
  2. Ha az első hívás felhasználói eljárásra vonatkozik, akkor megkeressük az eljárás első (visszalépés után: következő) olyan klózát, amelynek feje egyesíthető a hívással, és végrehajtjuk a redukciós lépést.
  3. Ha nincs egyesíthető fejű klóz, vagy a beépített eljárás meghiúsul, akkor visszalépés következik
  4. Egyébként folytatjuk a végrehajtást 1.-től az új célsorozattal.

# Prolog végrehajtás: visszalépés

- Visszalépés történik:
  - Ha egy beépített eljárás meghiúsul
  - Ha egy felhasználóieljárás-hívást nem lehet (több) klózfejjel egyesíteni.
- A visszalépés végrehajtása:
  - visszatérünk a legutolsó, felhasználói eljárással történt (sikeres) redukciós lépéshez,
  - annak *bemeneti* célsorozatát megpróbáljuk *újabb* klózzal redukálni (végrehajtás 2. lépése)
  - ennek meghiúsulása újabb visszalépést okoz.
- Visszalépések fajtái:
  - sekély: egy eljárás egy klózából ugyanezen eljárás egy későbbi klózába kerül a vezérlés,
  - mély: egy már lefutott eljárás belsejébe térünk vissza, újabb megoldást kérve.

# Sekély visszalépés: példa

```
% :- type tree4 == integer \/ {tree4--tree4}.

sum_tree4(Tree, Sum) :-                    % (1)
        integer(Tree), Sum = Tree.
sum_tree4(Left--Right, Sum) :-             % (2)
        sum_tree4(Left, Sum1),
        sum_tree4(Right, Sum2),
        Sum is Sum1+Sum2.
```

- kezdeti célsorozat: `sum\_tree4(5--3, S)`
- az első (egyetlen) hívás az (1) klóz fejével illeszthető
- a redukciós lépés eredménye: `integer(5--3), S=5--3`
- az első, beépített hívás meghiúsul, visszalépés következik
- visszatérünk a kezdeti célsorozathoz, de a (2) klóztól folytatva az egyesíthető fejű klóz keresését.
- a (2) klózzal redukálva: `sum_tree4(5, S1), sum_tree4(3, S2), S is S1+S2`

# A Prolog vezérlési szerkezete — első közelítés

- Vezérlés egy klózon belül:
  - Példa:

```
p(X) :- q(X), r(Y).
p(X) :- s(X).
```

  - `p(X)` igaz, ha `q(X)` *és* `r(X)` igaz, *vagy* `s(X)` igaz.
  - Közelítő C megfeleltetés: `BOOL p(x) { return q(x) && r(x) || s(x); }`
- Diszjunkció, mint édesítőszer
  - A fenti példa felírható a `;` operátor (diszjunkció) segítségével:

```
p(X) :-
     (   q(X), r(Y)
     ;   s(X)
     ).
```

  - A diszjunktív alakra hozáshoz szükséges lehet `= /2` feltételek bevezetésére, ha a klózfejek nem egyformák.
  - A diszjunkciók (segéd)eljárás bevezetésével mindig kiküszöbölhetőek.

# Diszjunkció: példa

```
:- op(500, xfx, --).

% :- type tree5 == integer \/ {tree5--tree5}.

sum_tree5(Tree, Sum) :-
        (   integer(Tree), Sum = Tree
        ;   Tree = Left--Right,
            sum_tree5(Left, Sum1),
            sum_tree5(Right, Sum2),
            Sum is Sum1+Sum2
        ).

| ?- sum_tree5(5--(3--2), Sum).
Sum = 10 ? ;
no
```

# Egy összetettebb példa: polinom behelyettesítési értéka

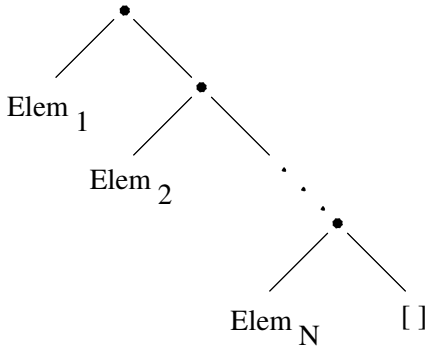
- Formula: számokból és az ‘x’ névkonstansból ‘+’ és ‘*’ operátorokkal felépülő kifejezés.
- `% :- type kif == {x} \/ number \/ {kif+kif} \/ {kif*kif}.`

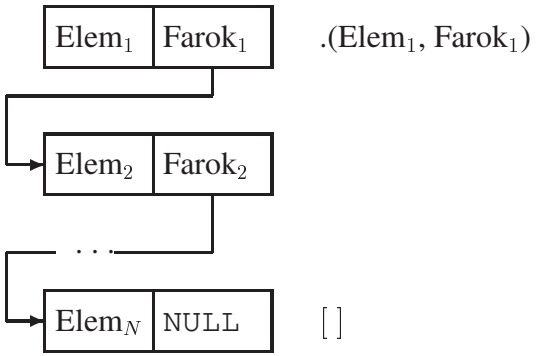
```
% erteke(Kif, X, E): A Kif formula értéke E, az x=X behelyettesítéssel.
erteke(x, X, E) :-
        E = X.
erteke(Kif, _, E) :-
        number(Kif), E = Kif.
erteke(K1+K2, X, E) :-
        erteke(K1, X, E1),
        erteke(K2, X, E2),
        E is E1+E2.
erteke(K1*K2, X, E) :-
        erteke(K1, X, E1),
        erteke(K2, X, E2),
        E is E1*E2.

| ?- erteke((x+1)*x+x+2*(x+x+3), 2, E).
E = 22 ? ;
no
```

# LISTÁK PROLOGBAN

# A Prolog lista-fogalma

- közönséges adattípus: `% :- type list(T) ---> .(T,list(T)) ; [].`
- `T` típusú elemekből álló lista az vagy egy `'.'/2` struktúra, vagy a `[]` atom. A struktúra első argumentuma `T` típusú, a lista feje (első eleme). A második argumentum `list(T)` típusú, a lista farka (a többi elemből álló lista);
- egyszerűsített írásmód („szintaktikus édesítés”);
- hatékonyabb megvalósítás.
- A listák fastruktúra alakja és megvalósítása



# Listák jelölése — szintaktikus édesítőszerek

- `[Fej|Farok]` $\equiv$ `.(Fej, Farok)`
- `[`$\mathtt{Elem}_1$`,`$\mathtt{Elem}_2$`,...,`$\mathtt{Elem}_N$`|Farok]` $\equiv$ `[`$\mathtt{Elem}_1$`|[`$\mathtt{Elem}_2$`,...,`$\mathtt{Elem}_N$`|Farok]]`
- `[`$\mathtt{Elem}_1$`,`$\mathtt{Elem}_2$`,...,`$\mathtt{Elem}_N$`]` $\equiv$ `[`$\mathtt{Elem}_1$`,`$\mathtt{Elem}_2$`,...,`$\mathtt{Elem}_N$`|[]]`

```
| ?- [1,2] = [X|Y].                  ⇒ X = 1, Y = [2] ?
| ?- [1,2] = [X,Y].                  ⇒ X = 1, Y = 2 ?
| ?- [1,2,3] = [X|Y].                ⇒ X = 1, Y = [2,3] ?
| ?- [1,2,3] = [X,Y].                ⇒ no
| ?- [1,2,3,4] = [X,Y|Z].            ⇒ X = 1, Y = 2, Z = [3,4] ?
| ?- L = [1|_], L = [_,2|_].         ⇒ L = [1,2|_A] ? % nyílt végű
| ?- L = .(1,[2,3|[]]).              ⇒ L = [1,2,3] ?
| ?- L = [1,2|.(3,[])].              ⇒ L = [1,2,3] ?
| ?- [X|[3-Y/X|Y]]= .(A, [A-B,6]).  ⇒ A=3, B=[6]/3, X=3, Y=[6] ?
```

# Listák összefűzése: az append/3 eljárás

```
% append(L1, L2, L3): Az L3 lista az L1 és L2 listák elemeinek
% egymás után fűzésével áll elő (jelöljük: L3 = L1⊕L2).
```

```
append0([], L2, L) :- L = L2.
append0([X|L1], L2, L) :-
    append0(L1, L2, L3), L = [X|L3].
```

```
append([], L, L).
append([X|L1], L2, [X|L3]) :-
    append(L1, L2, L3).
```

Helyhiány miatt az alábbi futás-listában az append szót app-ként rövidítjük.

```
    > app0([1,2,3],[4],A)
(2) > app0([2,3],[4],B), A=[1|B]
(2) > app0([3],[4],C), B=[2|C], A=[1|B]
(2) > app0([],[4],D), C=[3|D], B=[2|C],
      A=[1|B]
(1) > D=[4], C=[3|D], B=[2|C], A=[1|B]
BIP > C=[3,4], B=[2|C], A=[1|B]
BIP > B=[2,3,4], A=[1|B]
BIP > A=[1,2,3,4]
BIP > []
L = [1,2,3,4] ?
```

```
    > app([1,2,3],[4],A), write(A)
(2) > app([2,3],[4],B), write([1|B])
(2) > app([3],[4],C), write([1,2|C])
(2) > app([],[4],D), write([1,2,3|D])
(1) > write([1,2,3,4])
[1,2,3,4]
BIP > []
L = [1,2,3,4] ?
```

Az append(L1, ...) komplexitása: futási ideje arányos L1 hosszával.

# Bináris fa leveleinek listája

- Állítsuk elő egy bináris fa leveleinek listáját

```
:- op(500, xfx, --).
:- op(450, fx, @).

% :- type tree6 == @integer \/ {tree6--tree6}.

% leaves(Tree, Leaves): Tree leveleinek listája Leaves.
leaves(@Int, [Int]).
leaves(Left--Right, L) :-
    leaves(Left, L1),
    leaves(Right, L2),
    append(L1, L2, L).

| ?- leaves(@5--(@3-- @2), L).
L = [5,3,2] ? ;
no
```

# Listák megfordítása

- Naív (négyzetes lépésszámú) megoldás

```
% nrev(L, R): Az R lista az L megfordítása.
nrev([], []).
nrev([X|L], R) :-
    nrev(L, RL),
    append(RL, [X], R).
```

- Lineáris lépésszámú megoldás

```
% reverse(R, L): Az R lista az L megfordítása.
reverse(R, L) :-  revapp(L, [], R).

% revapp(L1, L2, R): L1 megfordítását L2 elé fűzve kapjuk R-t.
revapp([], R, R).
revapp([X|L1], L2, R) :-
    revapp(L1, [X|L2], R).
```

- A `lists` könyvtár tartalmazza az `append/3` és `reverse/2` eljárások definícióját.
- A könyvtár betöltése:

```
:- use_module(library(lists)).
```

# append és revapp — listák gyűjtési iránya

- Prolog megvalósítás

```
append([], L, L).
append([X|L1], L2, [X|L3]) :-
    append(L1, L2, L3).
```

```
revapp([], L, L).
revapp([X|L1], L2, L3) :-
    revapp(L1, [X|L2], L3).
```

- C++ megvalósítás

```
struct link   { link *next;
                char elem;
                link(char e): elem(e) {}
              };
typedef link *list;
```

```
list append(list list1, list list2)
{ list list3, *lp = &list3;
  for (list p=list1; p; p=p->next)
  { list newl = new link(p->elem);
    *lp = newl; lp = &newl->next;
  }
  *lp = list2;
  return list3;
}
```

```
list revapp(list list1, list list2)
{ list l = list2;
  for (list p=list1; p; p=p->next)
  { list newl = new link(p->elem);
    newl->next = l; l = newl;
  }

  return l;
}
```

# 2000 tavaszi kis házi feladat

- A feladat szövege:

  Állítsa elő egy $Sz$ nem negatív egész szám $A$ alapú számrendszerben vett jegyeinek listáját ($A > 1$ egész)! Írjon egy `szám/3` Prolog eljárást, amely a legnagyobb helyiértékű jegyet helyezi a lista elejére (természetes sorrend), és egy másik `fszám/3` eljárást, amely a legkisebb helyiértékű jeggyel kezdi a listát (fordított sorrend).

- Számjegyek előállítása természetes sorrendben

```
% szám(Szám, Alap, Jk): A Szám szám Alap alapú számrendszerben vett
% jegyeinek (balról jobbra haladó) listája Jk. (A 0 szám egy jegyből áll.)
szám(0, _, [0]).
szám(Sz, Alap, Jk) :-
        Sz > 0, szám(Sz, Alap, [], Jk).

% szám(Szám, Alap, Jk0, Jk): A Szám szám Alap alapú számrendszerben vett
% jegyeinek listáját Jk0 elé fűzve kapjuk Jk-t (A 0 jegylistája üres).
% Jelölés:  LL = L-L0 <----> az LL listát L0 elé fűzve kapjuk L-t.
szám(0, _, Jk, Jk).
szám(Sz, Alap, Jk0, Jk) :-
        Sz > 0, Sz1 is Sz//Alap, UtsoJegy is Sz mod Alap,
        szám(Sz1, Alap, [UtsoJegy|Jk0], Jk).
```

# 2000 tavaszi kis házi feladat — számjegyek fordított sorrendben

- Számjegyek előállítása fordított sorrendben

```
% fszám(Sz, A, Jk): Az Sz szám A alapú fordított jegylistája Jk.
fszám(0, _, [0]).
fszám(Sz, Alap, Jk) :- Sz > 0, fszám(Sz, Alap, [], Jk).

% fszám(Sz, A, Jk0, Jk): Az Sz szám A alapú fordított jegylistája Jk-Jk0.
fszám(0, _, Jk, Jk).
fszám(Sz, Alap, Jk0, [UtsoJegy|Jk]) :-
        Sz > 0, Sz1 is Sz//Alap, UtsoJegy is Sz mod Alap,
        fszám(Sz1, Alap, Jk0, Jk).
```

- A kétféle irányú gyűjtés összehasonlítása

```
fszám(0, _, Jk, Jk).
fszám(Sz, A, Jk0, [U|Jk]) :-
    Sz > 0, Sz1 is ...,
    U is ...,
    fszám(Sz1, A, Jk0, Jk).
```

```
szám(0, _, Jk, Jk).
szám(Sz, A, Jk0, Jk) :-
    Sz > 0, Sz1 is ...,
    U is ...,
    szám(Sz1, A, [U|Jk0], Jk).
```

# 2000 tavaszi kis házi feladat — egyszerűsítés

- A fordított jegylistát gyűjtő eljárás egyszerűsíthető
  - `fszám/4` minden hívása `fszám(_,_,[],_)` alakú.
  - `fszám(Sz, A, [], Jk)` ⇒`fszám12(Sz, A, Jk)`
- Az egyszerűsített program

```
% fszám(Szám, Alap, Jegyek): A Szám >= 0 szám  Alap > 1 alapú
% számrendszerben jobbról balra vett jegyeinek listája Jegyek.
fszám1(0, _, [0]).
fszám1(Sz, Alap, Jk) :-
        Sz > 0, fszám12(Sz, Alap, Jk).

% fszám12(Szám, Alap, Jk): A Szám >= 0 szám Alap > 1 alapú
% számrendszerben jobbról balra vett jegyeinek listája Jk.
fszám12(0, _, []).
fszám12(Sz, Alap, [UtsoJegy|Jk]) :-
        Sz > 0, Sz1 is Sz//Alap, UtsoJegy is Sz mod Alap,
        fszám12(Sz1, Alap, Jk).
```

# Bináris fa leveleinek listája: 2. megoldás, akkumulátorral

- A korábbi megoldásban az `append/3` hívás kiküszöbölhető.
- Ezáltal a jobb fa bejárása jobbrekurzívvá (azaz ciklussá) válik.

```
% leaves2(Tree, Leaves): Tree leveleinek listája Leaves.
leaves2(Tree, L) :-
        leaves2(Tree, [], L).

% leaves2(Tree, L0, L): Tree leveleinek listáját L0 elé fűzve
%                       kapjuk az L listát.
leaves2(@Int, L0, [Int|L0]).
leaves2(Left--Right, L0, L) :-
        leaves2(Right, L0, L1),
        leaves2(Left, L1, L).
```

# Bináris fa és lista összegzése akkumulátorral

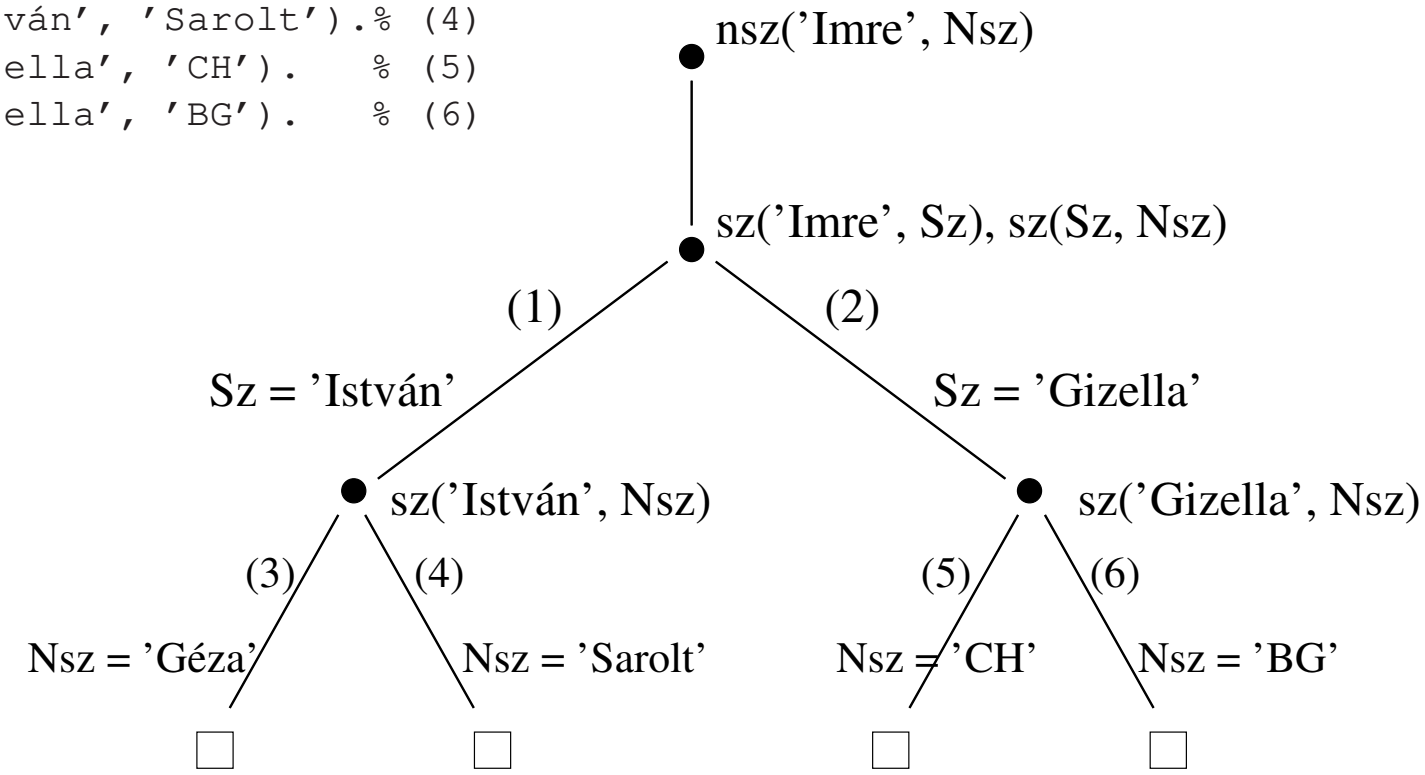
```
 % :- type tree6 == @integer \/ {tree6--tree6}.

% A Tree bináris fa levélösszege Sum.
sum_tree6(Tree, Sum) :-
        sum_tree6(Tree, 0, Sum).

% A Tree bináris fa levélösszege Sum0-hoz adva Sum-ot ad.
sum_tree6(@Tree, Sum0, Sum) :-
        Sum is Sum0+Tree.
sum_tree6(Left--Right, Sum0, Sum) :-
        sum_tree6(Left, Sum0, Sum1),
        sum_tree6(Right, Sum1, Sum).

% Az L számlista összege S.
sum_list1(L, S) :-
        sum_list1(L, 0, S).

% sum_list1(+L, +S0, -S): Az L számlista összege S0-hoz adva S-t ad.
%                         Az L számlista összege S-S0.
sum_list1([], S0, S0).
sum_list1([X|L], S0, S) :-
        S1 is S0+X,
        sum_list1(L, S1, S).
```

# VISSZALÉPÉSES KERESÉS PROLOGBAN

# Prolog végrehajtási 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)
```

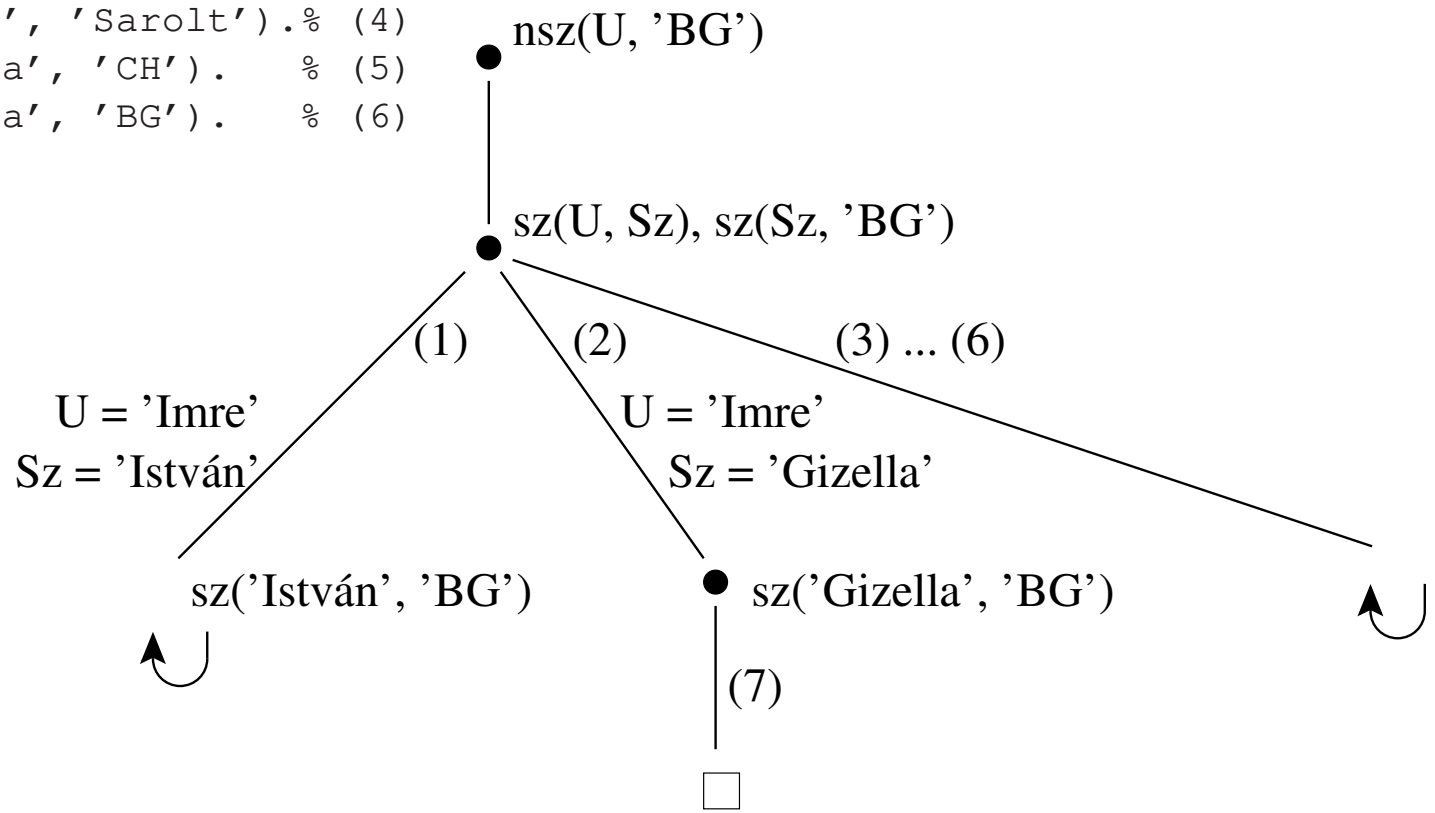
```
nsz(Gy, N) :-
    sz(Gy, Sz), sz(Sz, N).
```



# Újabb végrehajtási 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).
```



# A Prolog végrehajtási algoritmusa

1. *(Kezdeti beállítások:)* A verem üres, `CS := célsorozat`
2. *(Beépített eljárások:)* Ha `CS` első célja beépített akkor hajtsuk végre,
   a. Ha sikertelen $\Rightarrow$ 6. lépés.
   b. Ha sikeres, elvégezzük a behelyettesítéseket, `CS`-ből elhagyjuk az első hívást, $\Rightarrow$ 5. lépés.
3. *(Klózszámláló kezdőértékezése:)* `I = 1`.
4. *(Redukciós lépés:)* `CS` első hívásához tartozó eljárásdefinícióban `N` klóz van.
   a. Ha `I > N` $\Rightarrow$ 6. lépés.
   b. Redukciós lépés az `I`-edik klóz és a `CS` célsorozat között.
   c. Ha sikertelen, akkor `I := I+1` $\Rightarrow$ 4. lépés.
   d. Ha `I < N` (nem utolsó), akkor vermeljük `<CS,I>`-t.
   e. `CS :=` a redukciós lépés eredménye
5. *(Siker:)* Ha `CS` üres, akkor sikeres vég, egyébként $\Rightarrow$ 2. lépés.
6. *(Sikertelenség:)* Ha a verem üres, akkor sikertelen vég.
7. *(Visszalépés:)* Ha a verem nem üres, akkor leemeljük a veremből `<CS, I>`-t, `I := I+1`, és $\Rightarrow$ 4. lépés.

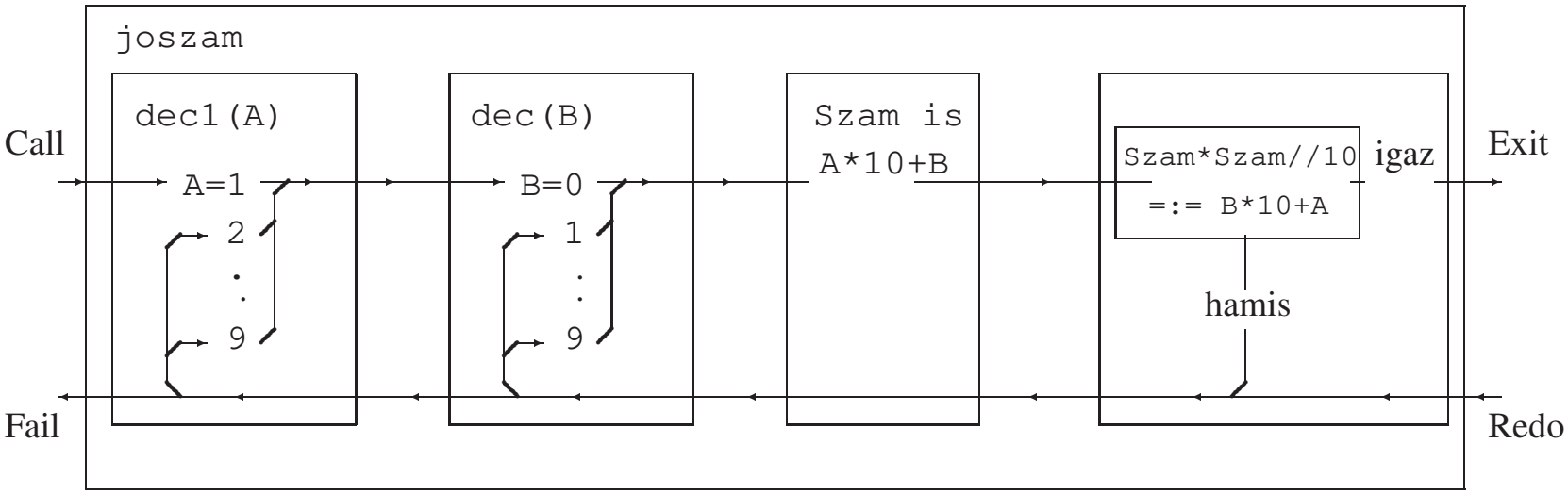
# Visszalépéses keresés — egy aritmetikai példa

- Példa: „jó” számok keresése
- A feladat: keressük meg azokat a kétjegyű számokat amelyek négyzete háromjegyű és a szám fordítottjával kezdődik
- A program:

```
% dec1(J): J egy pozitív decimális számjegy.
dec1(1). dec1(2). dec1(3). dec1(4).
dec1(5). dec1(6). dec1(7), dec1(8). dec1(9).

% dec(J): J egy decimális számjegy.
dec(0).
dec(J) :- dec1(J).

% Szam négyzete háromjegyű és a Szam fordítottjával kezdődik.
joszam(Szam):-
        dec1(A), dec(B),
        Szam is A * 10 + B, Szam * Szam // 10 =:= B * 10 + A.
```

# Prolog végrehajtás — a 4-kapus doboz modell

```
joszam(Szam):-
        dec1(A), dec(B),
        Szam is A * 10 + B, Szam * Szam // 10 =:= B * 10 + A.
```



# Visszalépéses keresés — számintervallum felsorolása

```
% between(M, N, I): M =< I =< N, I egész.
between(M, N, M) :-
        M =< N.
between(M, N, I) :-
        M < N,
        M1 is M+1,
        between(M1, N, I).

dec(X) :- between(0, 9, X).

| ?- between(1, 2, _X), between(3, 4, _Y), Z is 10*_X+_Y.
Z = 13 ? ;
Z = 14 ? ;
Z = 23 ? ;
Z = 24 ? ;
no
```

# Keresés listában: `member(E, L)`: E az L lista eleme

```
member(Elem, [Elem|_]).
member(Elem, [_|Farok]) :-
        member(Elem, Farok).
```

```
member(Elem, [Fej|Farok]) :-
        (   Elem = Fej
        ;   member(Elem, Farok)
        ).
```

## Eldöntendő kérdés

```
| ?- member(2, [1,2,3]).          ⇒ yes
```

## Megválaszolandó kérdések

```
| ?- member(X, [1,2,3]).          ⇒ X = 1 ?  ; X = 2 ?  ; X = 3 ?  ; no
| ?- member(X, [1,2,1]).          ⇒ X = 1 ?  ; X = 2 ?  ; X = 1 ?  ; no
```

## Vegyes használat, listák metszete

```
| ?- member(X, [1,2,3]),
     member(X, [5,4,3,2,3]).      ⇒ X = 2 ?  ; X = 3 ?  ; X = 3 ?  ; no
```

## Lista elemévé tesz, végtelen választás!

```
| ?- member(1, L).                ⇒ L = [1|_A] ? ; L = [_A,1|_B] ? ;
                                    L = [_A,_B,1|_C] ? ; ...
```

# member/2 általánosítása: select/3

```
% select(Elem, Lista, Marad):  Elemet a Lista-ból elhagyva marad Marad.
select(Elem, [Elem|Marad], Marad).     % Elhagyjuk a fejet, marad a farok.
select(Elem, [X|Farok], [X|Marad0]) :-
        select(Elem, Farok, Marad0).   % A farokból hagyunk el elemet.
```

```
| ?- select(1, [2,1,3], L).
        L = [2,3] ? ; no
| ?- select(X, [1,2,3], L).
        L=[2,3], X=1 ? ;  L=[1,3], X=2 ? ;   L=[1,2], X=3 ? ; no
| ?- select(3, L, [1,2]).
        L = [3,1,2] ? ; L = [1,3,2] ? ; L = [1,2,3] ? ; no
| ?- select(3, [2|L], [1,2,7,3,2,1,8,9,4]).
        no
| ?- select(1, [X,2,X,3], L).
        L = [2,1,3], X = 1 ? ; L = [1,2,3], X = 1 ? ; no
```

A `lists` könyvtár tartalmazza a `member/2` és `select/3` eljárások definícióját is.

# A végrehajtás optimalizálása: indexelés

1. *(Kezdeti beállítások:)* A verem üres, `CS := célsorozat`
2. *(Beépített eljárások:)* Ha `CS` első célja beépített akkor hajtsuk végre,

   (...)
3. *(Klózszámláló kezdőértékezése:)* `I = 1`.
4. *(Redukciós lépés:)* `CS` *első hívásához elkészítjük a potenciálisan illeszthető klózok listáját (indexelés). Tegyük fel, hogy ez a lista* `N` *elemű.*

   a. Ha `I > N` $\Rightarrow$ 6. lépés.

   b. Redukciós lépés az *indexelési lista* `I`-edik klóza és a `CS` célsorozat között.

   c. Ha sikertelen, akkor `I := I+1` $\Rightarrow$ 4. lépés.

   d. Ha `I < N` (nem utolsó), akkor vermeljük `<CS,I>`-t.

   e. `CS` := a redukciós lépés eredménye
5. *(Siker:)* Ha `CS` üres, akkor sikeres vég, egyébként $\Rightarrow$ 2. lépés.
6. *(Sikertelenség:)* Ha a verem üres, akkor sikertelen vég.
7. *(Visszalépés:)* Ha a verem nem üres, akkor leemeljük a veremből `<CS, I>`-t, `I := I+1`, és $\Rightarrow$ 4. lépés.

# Indexelés

- Mi az indexelés?
  - egy hívásra illeszthető klózok gyors kiválasztása,
  - egy eljárás klózainak fordítási idejű csoportosításával,
- A legtöbb Prolog rendszer, így a SICStus Prolog is, az első fej-argumentum alapján indexel (first argument indexing).
- Az indexelés alapja az első fejargumentum külső funktora:
  - `C` szám vagy névkonstans esetén `C/0`;
  - `R` nevű és `N` argumentumú struktúra esetén `R/N`;
  - változó esetén nem értelmezett.
- Az indexelés megvalósítása:
  - Fordításkor a funktorokhoz elkészítjük az illeszthető klózok részhalmazát.
  - Futáskor lényegében konstans idő alatt választunk a részhalmazok közül.
  - *Fontos:* ha egyelemű a részhalmaz, nem hozunk létre választási pontot!

# Példa indexelésre

```
p(0, a).            /* (1) */      | q(1).
p(X, t) :- q(X).    /* (2) */      | q(2).
p(s(0), b).         /* (3) */      |
p(s(1), c).         /* (4) */      |
p(9, z).            /* (5) */      |
```

- A `p(A, B)` hívással illesztendő klózhalmaz:
  - {(1) (2) (3) (4) (5)} ha `A` változó;
  - {(1) (2)} ha `A = 0`;
  - {(2) (3) (4)} ha `A` fő funktora `s/1`;
  - {(2) (5)} ha `A = 9`;
  - {(2)} minden más esetben.
- Példák hívásokra:
  - `p(1, Y)` nem hoz létre választási pontot.
  - `p(s(1), Y)` létrehoz választási pontot, de azt lefutás előtt megszünteti.
  - `p(s(0), Y)` választási pontot hagy a lefutásakor.

# Tömör és minta-kifejezések, lista-minták, nyílt végű listák

- Tömör (ground) kifejezés: változót nem tartalmazó kifejezés
- Minta: egy általában nem nem tömör kifejezés, mindazon kifejezéseket „képviseli”, amelyek belőle változó-behelyettesítéssel előállnak.
- Lista-minta: listát (is) képviselő minta.
- Nyílt végű lista: olyan lista-minta, amely bármilyen hosszú listát is képvisel.
- Zárt végű lista: olyan lista(-minta), amely egyféle hosszú listát képvisel.

| Zárt v. | Milyen listákat képvisel |
|---|---|
| `[X]` | egyelemű |
| `[X,Y]` | kételemű |
| `[X,X]` | két egyforma elemből álló |
| `[X,1,Y]` | 3 elemből áll, 2. eleme `1` |

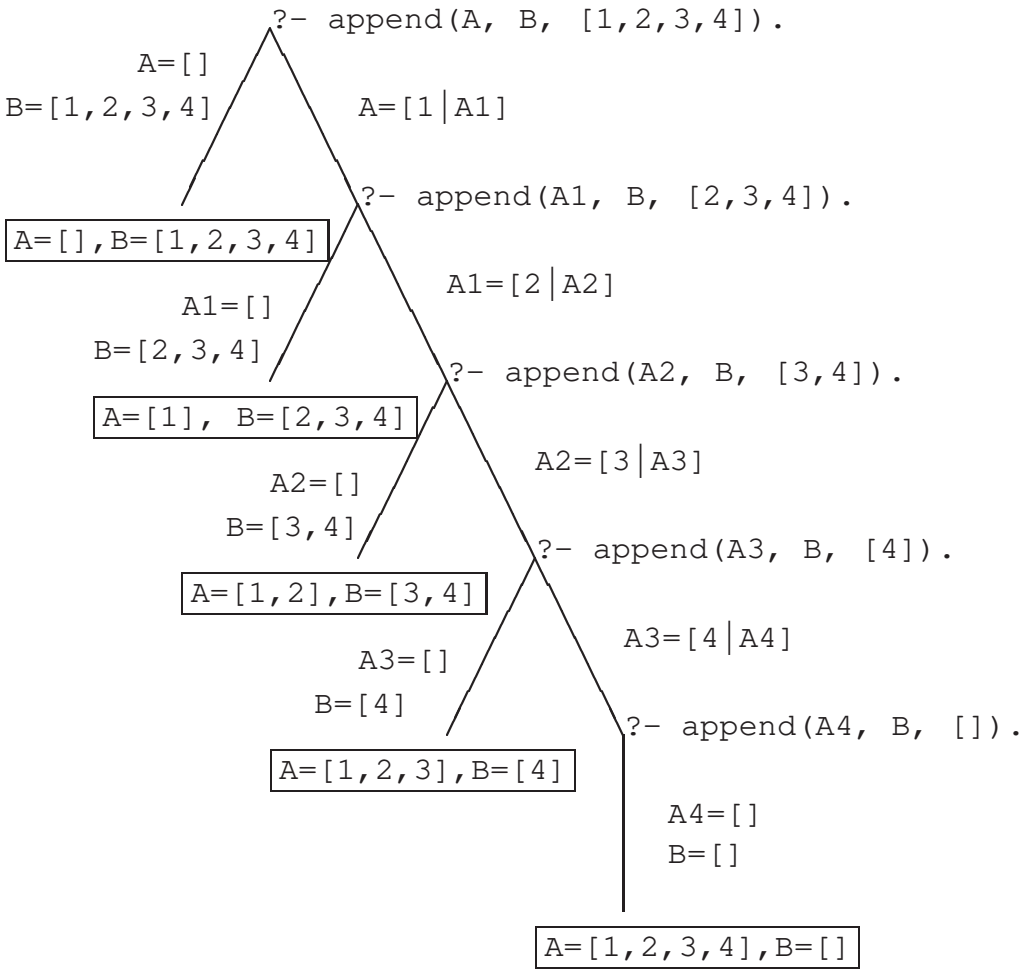
| Nyílt v. | Milyen listákat képvisel |
|---|---|
| `X` | tetszőleges |
| `[X\|Y]` | nem üres (legalább 1 elemű) |
| `[X,Y\|Z]` | legalább 2 elemű |
| `[a,b\|Z]` | legalább 2 elemű, elemei: `a, b, ...` |

„Biztonságos” a futás, azaz véges a keresési tér, ha:

- `member/2` második argumentuma zárt végű.
- `select/3` 2. és 3. argumentuma közül az egyik zárt végű.
- `append/3` 1. és 3. argumentuma közül az egyik zárt végű.

# Listák szétbontása az append/3 segítségével

```
% append(L1, L2, L3):
% Az L3 lista az L1 és L2
% listák elemeinek egymás
% után fűzésével áll elő.
append([], L, L).
append([X|L1], L2, [X|L3]) :-
    append(L1, L2, L3).

| ?- append(A, B, [1,2,3,4]).
A = [], B = [1,2,3,4] ? ;
A = [1], B = [2,3,4] ? ;
A = [1,2], B = [3,4] ? ;
A = [1,2,3], B = [4] ? ;
A = [1,2,3,4], B = [] ? ;
no
```



# Variációk `append`re 1. — Három lista összefűzése

```
% L1 ⊕ L2 ⊕ L3 = L123, ahol L1 és L2 adott.
append(L1, L2, L3, L123) :-
        append(L1, L2, L12), append(L12, L3, L123).
```

- Nem hatékony, pl.: `append([1,...,100],[1,2,3],[1], L)` 103 helyett 203 lépés!
- Szétszedésre nem alkalmas — végtelen választási pontot hoz létre
  (véges a keresési tér, ha az 1. és 3. argumentum legalább egyike zárt végű lista.)

Szétszedésre is alkalmas, hatékony változat

```
% L1 ⊕ L2 ⊕ L3 = L123, ahol vagy L1 és L2 vagy L123 adottă(zárt végű).
append(L1, L2, L3, L123) :-
        append(L1, L23, L123), append(L2, L3, L23).
```

Az első `append/3` hívás nyílt végű listát állít elő:

```
| ?- append([1,2], L23, L).      ⇒      L = [1,2|L23] ?
```

# Variációk appendre 2. — lista folytonos része

```
% L123 folytonos részlistája L2 (L123 =  _ ⊕ L2 ⊕ _).
% L123 adott, L2 ismeretlen.
csublist(L2, L123) :-
        append(_L1, L23, L123),
        append(L2, _L3, L23).

% Adott L123-nak folytonos része egy adott L2.
check_csublist(L2, L123) :-
        append(L2, _L3, L23),
        append(_L1, L23, L123).
```

A két változat hatékonyságának összehasonlítása

- `L123 = [`$\underbrace{0,1,2,3,4}_{\text{x100000}}$`,...,10], L2 = [0,1,2,3,4,10]`
- `csublist(L2, L123):` 570 msec
- `check_csublist(L2, L123):` 430 msec

# Mintakeresés append/3-mal

- Párban előforduló elemek

```
% párban(Lista, Elem): A Lista számlistának Elem olyan
% eleme, amely egy ugyanilyen értékű elemmel szomszédos.
párban(L, E) :-
    append(_, [E,E|_], L).

| ?- párban([1,8,8,3,4,4], E).
        E = 8 ? ; E = 4 ? ; no
```

- Dadogó részek

```
% dadogó(L, D): D olyan nem üres részlistája L-nek,
% amelyet egy vele megegyező részlista követ.
dadogó(L, D) :-
        append(_, Farok, L),
        D = [_|_],
        append(D, Vég, Farok),
        append(D, _, Vég).

| ?- dadogó([2,2,1,2,2,1], D).
        D = [2] ? ; D = [2,2,1] ? ; D = [2] ? ; no
```

# LEGALAPVETŐBB BEÉPÍTETT ELJÁRÁSOK

# Aritmetikai beépített eljárások

- `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 (`=:=` ⇒egyenlő, `=\=` ⇒nem-egyenlő).
- Ha `Kif, Kif1, Kif2` valamelyike nem aritmetikai kifejezés ⇒hiba.
- Legfontosabb aritmetikai operátorok: `+, -, *, /, mod, //` (egész-osztás)
- Példák:

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

# Programfejlesztési beépített eljárások

- `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 beolvasott program elemei Prolog kifejezések (`':-', ','` operátorok!)
- `halt`: A Prolog rendszer befejezi működését.
- Példák:

```
> sicstus
SICStus 3.9.0 (x86-linux-glibc2.1): Tue Feb 12 17:07:18 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 eljárások

- `write(X)`: Az `X` Prolog kifejezést kiírja (ha kell, operátorokkal).
- `write_canonical(X)`: Az `X` Prolog kifejezést struktúra-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), nl, fail.
Géza
Sarolt
no
```

# NAGYOBB PÉLDÁK — NEGÁCIÓ, FELTÉTELES SZERKEZET

# Az útvonalkeresési feladat

A feladat: tekintsük (autóbusz)járatok egy halmazát. Mindegyik járathoz a két végpont és az útvonal hossza van megadva. Írjunk Prolog eljárást, amellyel megállapítható, hogy két pont összeköthető-e pontosan N csatlakozó járattal!

```
% járat(A, B, H): Az A és B városok között van járat, és hossza H km.
járat('Budapest', 'Prága', 515).
járat('Budapest', 'Bécs', 245).
járat('Bécs', 'Berlin', 635).
járat('Bécs', 'Párizs', 1265).

% útszakasz(A, B, H): A-ból B-be eljuthatunk egy H úthosszú járattal.
útszakasz(Kezdet, Cél, H) :-
     (   járat(Kezdet, Cél, H)
     ;   járat(Cél, Kezdet, H)
     ).
```

# Az útvonalkeresési feladat — folytatás

```
% útvonal(N, A, B, H): A és B között van (pontosan)
% N szakaszból álló útvonal, amelynek összhossza H.
útvonal(0, Hová, Hová, 0).
útvonal(N, Honnan, Hová, H) :-
    N > 0,
    N1 is N-1,
    útszakasz(Honnan, Közben, H1),
    útvonal(N1, Közben, Hová, H2),
    H is H1+H2.

| ?- útvonal(2, 'Párizs', Hová, H).
        H = 1900, Hová = 'Berlin' ? ;
        H = 2530, Hová = 'Párizs' ? ;
        H = 1510, Hová = 'Budapest' ? ;
no
| ?-
```

# Körmentes út keresése, útvonal-gyűjtéssel

```
:- use_module(library(lists), [member/2, reverse/2]).

% útvonal_2(N, A, B , Út, H): A és B között van (pontosan)
% N szakaszból álló körmentes Út útvonal, amelynek összhossza H.
útvonal_2(N, Honnan, Hová, Út, H) :-
    útvonal_2(N, Honnan, Hová, [Honnan], Út, H).

% útvonal_2(N, A, B, K, Út, H): A és B között van pontosan
% N szakaszból álló körmentes, K elemein át nem menő H hosszú Út út.
útvonal_2(0, Hová, Hová, Kizártak, Út, 0) :-
    reverse(Kizártak, Út).
útvonal_2(N, Honnan, Hová, Kizártak, Út, H) :-
    N > 0, N1 is N-1,
    útszakasz(Honnan, Közben, H1),
    \+ member(Közben, Kizártak),
    útvonal_2(N1, Közben, Hová, [Közben|Kizártak], Út, H2), H is H1+H2.

| ?- útvonal_2(2, 'Párizs', _, Út, H).
        H = 1900, Út = ['Párizs','Bécs','Berlin'] ? ;
        H = 1510, Út = ['Párizs','Bécs','Budapest'] ? ; no
```

# A meghiúsulásos negáció (NF — Negation by Failure)

- A `\+ Hívás` beépített meta-eljárás (vö. $\nvdash$ — nem bizonyítható)
  - végrehajtja a `Hívás` hívást,
  - ha `Hívás` sikeresen fut le, akkor meghiúsul,
  - egyébként (behelyettesítés nélkül) sikerül.
  - $\backslash + H$ jelentése: $\neg\exists X(H)$, ahol $X$ a $H$-ban a *hívás pillanatában* behelyettesítetlen változókat jelöli.
  - A „zárt világ feltételezése” (CWA) — ami nem bizonyítható, az nem igaz.
- Példák:

```
| ?- \+ szuloje('Imre', X).          ----> no
| ?- \+ szuloje('Géza', X).          ----> true ?
| ?- \+ X = 1, X = 2.                ----> no
| ?- X = 2, \+ X = 1.                ----> X = 2 ?

testvere(T1, T2) :- szuloje(T1, A), szuloje(T2, A), \+ T1 = T2.
```

# Súlyozott gráf ábrázolása éllistával

- A gráf ábrázolása
  - a gráf élek listája,
  - az él egy három-argumentumú struktúra,
  - argumentumai: a két végpont és a súly.
- Típus-definíció

```
% :- type él ---> él(pont, pont, súly).
% :- type pont == atom.
% :- type súly == integer.
% :- type gráf == list(él).
```

- Példa

```
hálózat([él('Budapest','Bécs',245),
         él('Budapest','Prága',515),
         él('Bécs','Berlin',635),
         él('Bécs','Párizs',1265)]).
```

# Ismétlődésmentes útvonal keresése listával ábrázolt gráfban

```
:- use_module(library(lists), [select/3]).

% útvonal_3(N, G, A, B, L, H): A G gráfban van egy  A-ból
% B-be menő N szakaszból álló L út, melynek összhossza H.
útvonal_3(0, _Gráf, Hová, Hová, [Hová], 0).
útvonal_3(N, Gráf, Honnan, Hová, [Honnan|Út], H) :-
        N >0, N1 is N-1,
        select(Él, Gráf, Gráf1),
        él_végpontok_hossz(Él, Honnan, Közben, H1),
        útvonal_3(N1, Gráf1, Közben, Hová, Út, H2),
        H is H1+H2.

% él_végpontok_hossz(Él, A, B, H): Az Él irányítatlan él
% végpontjai A és B, hossza H.
él_végpontok_hossz(él(A,B,H), A, B, H).
él_végpontok_hossz(él(A,B,H), B, A, H).

| ?- hálózat(_Gráf), útvonal_3(2, _Gráf, 'Budapest', _, Út, H).
        H = 880, Út = ['Budapest','Bécs','Berlin'] ? ;
        H = 1510, Út = ['Budapest','Bécs','Párizs'] ? ;
        no
```

# Együttható meghatározása lineáris kifejezésben

- Formula: számokból és az ‘x’ névkonstansból ‘+’ és ‘*’ operátorokkal épül fel.
- `% :- type kif == {x} \/ number \/ {kif+kif} \/ {kif*kif}.`
- Lineáris formula: a ‘*’ operátor legalább egyik oldalán szám áll.

```
% egyhat(Kif, E): A Kif lineáris formulában az x együtthatója E.
egyhat(x, 1).
egyhat(Kif, E) :-
        number(Kif), E = 0.
egyhat(K1+K2, E) :-
        egyhat(K1, E1),
        egyhat(K2, E2),
        E is E1+E2.
egyhat(K1*K2, E) :-
        number(K1),
        egyhat(K2, E0),
        E is K1*E0.
egyhat(K1*K2, E) :-
        number(K2),
        egyhat(K1, E0),
        E is K2*E0.

| ?- egyhat(((x+1)*3)+x+2*(x+x+3), E).
E = 8 ? ;
no

| ?- egyhat(2*3+x, E).
E = 1 ? ;
E = 1 ? ; no
```

# Együttható meghatározása: többszörös megoldások kiküszöbölése

- negáció alkalmazásával:

```
(...)
egyhat(K1*K2, E) :-
        number(K1), egyhat(K2, E0), E is K1*E0.
egyhat(K1*K2, E) :-
        \+ number(K1),
        number(K2), egyhat(K1, E0), E is K2*E0.
```

- feltételes kifejezéssel:

```
(...)
egyhat(K1*K2, E) :-
        (   number(K1) -> egyhat(K2, E0), E is K1*E0
        ;   number(K2), egyhat(K1, E0), E is K2*E0
        ).
```

# Feltételes kifejezések

- Szintaxis (`felt`, `akkor`, `egyébként` tetszőleges célsorozatok):

```
(...) :-
        (...),
        (   felt -> akkor
        ;   egyébként
        ),
        (...).
```

- Deklaratív szemantika: a fenti alak jelentése megegyezik az alábbival, ha a `felt` egy egyszerű feltétel (nem oldható meg többféleképpen):

```
(...) :-
        (...),
        (   felt, akkor
        ;   \+ felt, egyébként
        ),
        (...).
```

# Feltételes kifejezések (folyt.)

- Procedurális szemantika

  A `(felt->akkor;egyébként),folytatás` célsorozat végrehajtása:

  - Végrehajtjuk a `felt` hívást.
  - Ha `felt` sikeres, akkor az `akkor,folytatás` célsorozatra redukáljuk a fenti célsorozatot, a `felt` *első* megoldása által eredményezett behelyettesítésekkel. A `felt` cél többi megoldását nem keressük meg.
  - Ha `felt` sikertelen, akkor az `egyébként,folytatás` célsorozatra redukáljuk, behelyettesítés nélkül.

- Többszörös elágaztatás skatulyázott feltételes kifejezésekkel:

```
(   felt1 -> akkor1
;   felt2 -> akkor2
;   ...
)
```

```
(   felt1 -> akkor1
;  (felt2 -> akkor2
;   ...
...))
```

- Az `egyébként` rész elhagyható, alapértelmezése: `fail`.

# Feltételes kifejezés — példák

- Faktoriális

```
% fakt(+N, ?F): N! = F.
fakt(N, F) :-
        (   N = 0 -> F = 1            % N = 0,  F = 1
        ;   N > 0, N1 is N-1, fakt(N1, F1), F is N*F1
        ).
```

- Jelentése azonos a sima diszjunkciós alakkal (lásd komment), de annál hatékonyabb, mert nem hoz létre választási pontot.
- Szám előjele

```
% Sign = sign(Num)
sign(Num, Sign) :-
        (   Num > 0 -> Sign = 1
        ;   Num < 0 -> Sign = -1
        ;   Sign = 0
        ).
```

# Feltételes kifejezés és negáció

- A `\+ felt` negáció kiváltható a `( felt -> fail ; true )` feltételes kifejezéssel.
- Példa: ellenőrizzük, hogy egy adott kifejezés nem eleme egy listának (pontosabban nem egyesíthető a lista egyik elemével sem).

```
nem_eleme(E, L) :-
        (   member(E, L) -> fail
        ;   true
        ).
```

- A `\=` beépített eljárás jelentése: az argumentumok nem egyesíthetők:
  `X \= Y :- \+ X = Y.`
- A nem-eleme példa-eljárás rekurzív megvalósítása:

```
nem_eleme(E, []).
nem_eleme(E, [X|L]) :-
        E \= X,
        nem_eleme(E, L).
```

# Példa: adott értékű kifejezés előállítása

```
% A K kifejezés értéke E, és
% az L lista összes eleméből
% a négy alapművelettel épül fel.
kif(L, E, K) :-
        fa_levelekbol(L, K),
        E =:= K.

% fa_levelekbol(L, K): K az L
% listából alapműv-kel épül fel.
fa_levelekbol([K], K).
fa_levelekbol(L, K) :-
        L = [_,_|_],
        epit(K1, K2, K),
        L1 = [_|_], L2 = [_|_],
        szetbont(L, L1, L2),
        fa_levelekbol(L1, K1),
        fa_levelekbol(L2, K2).
```

```
% epit(A, B, Kif): Kif A-ból és
% B-ből a négy alapművelet
% egyikével épül fel.
epit(A, B, A+B).
epit(A, B, A-B).
epit(A, B, A*B).
epit(A, B, A/B).

% szetbont(L, L1, L2): Az L
% lista L1 és L2 sorrendtartó
% összefésülésével áll elő.
szetbont([], [], []).
szetbont([X|L], [X|L1], L2) :-
        szetbont(L, L1, L2).
szetbont([X|L], L1, [X|L2]) :-
        szetbont(L, L1, L2).
```

# A PROLOG SZINTAXIS

# A Prolog szintaxis összefoglalása

- A Prolog szintaxis alapelvei
  - Minden programelem kifejezés!
  - A szükséges összekötő jelek (`',', ;, :- -->`): szabványos operátorok.
  - A beolvasott kifejezést funktora alapján osztályozzuk:
    - *kérdés:* `?- Cél.`
      `Cél`t lefuttatja, és a változó-behelyettesítéseket kiírja (ez az alapértelmezés az ún. top-level interaktív felületen).
    - *parancs:* `:- Cél.`
      A `Cél`t csendben lefuttatja. Pl. deklaráció (operátor, ...) elhelyezésére.
    - *szabály:* `Fej :- Törzs.`
      A szabályt felveszi a programba.
    - *nyelvtani szabály:* `Fej --> Törzs.`
      Prolog szabállyá alakítja és felveszi (lásd a DCG nyelvtan).
    - *tényállítás:* `Minden egyéb kifejezés.`
      Üres törzsű szabályként felveszi a programba.

# A Prolog nyelv-változatok

- A SICStus rendszer két üzemmódja
  - `iso` Az ISO Prolog szabványnak megfelelő.
  - `sicstus` Korábbi változatokkal kompatibilis.
  - Állítása: `set_prolog_flag(language,` *`Mód`*`).`
  - Különbségek:
    - szintaxis-részletek, pl. a `0x1ff` szám-alak csak ISO módban,
    - beépített eljárások viselkedésének kisebb eltérései.
  - az eddig ismertetett eljárások hatása lényegében nem változik.

# Szintaktikus édesítőszerek — összefoglalás, gyakorlati tanácsok

- Operátoros kifejezések alapstruktúra alakra hozása
  - Zárójelezzük be a kifejezést, az operátorok prioritása és fajtája alapján, például `-a+b*2` ⇒ `((-a)+(b*2))`.
  - Hozzuk az operátoros kifejezéseket alapstruktúra alakra:
    `(A Inf B)` ⇒ `Inf(A,B)`, `(Pref A)` ⇒ `Pref(A)`, `(A Postf)` ⇒ `Postf(A)`
    Példa: `((-a)+(b*2))` ⇒ `(-(a)+ *(b,2))` ⇒ `+(-(a),*(b,2))`.
  - Trükkös esetek:
    - A vesszőt névként idézni kell: pl. `(pp,(qq;rr))` ⇒ `','(pp,;(qq,rr))`.
    - `-` *`Szám`* ⇒ negatív számkonstans, de `-` *`Egyéb`* ⇒ prefix alak.
      Példa. `-1+2` ⇒ `+(-1,2)`, de `-a+b` ⇒ `+(-(a),b)`.
    - *`Név`*`(...)` ⇒ struktúrakifejezés;
      *`Név`* `(...)` ⇒ prefix operátoros kifejezés. Példák:
      `-(1,2)` ⇒ `-(1,2)` (változatlan), de
      `- (1,2)` ⇒ `-(','(1,2))`.

# Szintaktikus édesítőszerek — listák, egyebek

- Listák alapstruktúra alakra hozása
  - Farok-megadás betoldása.
    `[1,2]` ⇒ `[1,2|[]]`. `[[X|Y]]` ⇒ `[[X|Y]|[]]`
  - Vessző (ismételt) kiküszöbölése `[Elem1,Elem2...]` ⇒ `[Elem1|[Elem2...]]`.
    `[1,2|[]]` ⇒ `[1|[2|[]]]`
    `[1,2,3|[]]` ⇒ `[1|[2,3|[]]]` ⇒ `[1|[2|[3|[]]]]`
  - Struktúrakifejezéssé alakítás: `[Fej|Farok]` ⇒ `.(Fej,Farok)`.
    `[1|[2|[]]]` ⇒ `.(1,.(2,[]))`, `[[X|Y]|[]]` ⇒ `.(.(X,Y),[])`
- Egyéb szintaktikus édesítőszerek:
  - Karakterkód-jelölés: `0'Kar`.
    `0'a` ⇒ `97`, `0'b` ⇒ `98`, `0'c` ⇒ `99`, `0'd` ⇒ `100`, `0'e` ⇒ `101`
  - Füzér (string): `"xyz..."` ⇒ az `xyz...` karakterek kódját tartalmazó lista
    `"abc"` ⇒ `[97,98,99]`, `""` ⇒ `[]`, `"e"` ⇒ `[101]`
  - Kapcsos zárójelezés: `{Kif}` ⇒ `{}(Kif)` (egy `{}` nevű, egyargumentumú struktúra — a `{}` jelpár egy önálló lexikai elem, egy névkonstans).
  - Bináris, hexa, stb alak (csak `iso` módban), pl. `0b101010`, `0x1a`.

# Kifejezések szintaxisa — kétszintű nyelvtanok

- Egy részlet egy „hagyományos” nyelv kifejezés-szintaxisából:

⟨ kifejezés ⟩ ::= ⟨ tag ⟩
| ⟨ kifejezés ⟩ ⟨ additív művelet ⟩ ⟨ tag ⟩

⟨ tag ⟩ ::= ⟨ tényező ⟩
| ⟨ tag ⟩ ⟨ multiplikatív művelet ⟩ ⟨ tényező ⟩

⟨ tényező ⟩ ::= ⟨ szám ⟩ | ⟨ azonosító ⟩ | ( ⟨ kifejezés ⟩ )

- Ugyanez kétszintű nyelvtannal:

⟨ kifejezés ⟩ ::= ⟨ kif *2* ⟩

⟨ kif *N* ⟩ ::= ⟨ kif *N-1* ⟩
| ⟨ kif *N* ⟩ ⟨ *N* prioritású művelet ⟩ ⟨ kif *N-1* ⟩

⟨ kif *0* ⟩ ::= ⟨ szám ⟩ | ⟨ azonosító ⟩ | ( ⟨ kif *2* ⟩ )

{az additív ill. multiplikatív műveletek prioritása *2* ill. *1* }

# Kifejezések szintaxisa

⟨ programelem ⟩ ::= ⟨ kifejezés 1200 ⟩ ⟨ záró-pont ⟩

⟨ kifejezés *N* ⟩ ::= ⟨ op *N* fx ⟩ ⟨ köz ⟩ ⟨ kifejezés *N-1* ⟩
| ⟨ op *N* fy ⟩ ⟨ köz ⟩ ⟨ kifejezés *N* ⟩
| ⟨ kifejezés *N-1* ⟩ ⟨ op *N* xfx ⟩ ⟨ kifejezés *N-1* ⟩
| ⟨ kifejezés *N-1* ⟩ ⟨ op *N* xfy ⟩ ⟨ kifejezés *N* ⟩
| ⟨ kifejezés *N* ⟩ ⟨ op *N* yfx ⟩ ⟨ kifejezés *N-1* ⟩
| ⟨ kifejezés *N-1* ⟩ ⟨ op *N* xf ⟩
| ⟨ kifejezés *N* ⟩ ⟨ op *N* yf ⟩
| ⟨ kifejezés *N-1* ⟩

⟨ kifejezés 1000 ⟩ ::= ⟨ kifejezés 999 ⟩ , ⟨ kifejezés 1000 ⟩

⟨ kifejezés 0 ⟩ ::= ⟨ név ⟩ ( ⟨ argumentumok ⟩ )
{ A ⟨ név ⟩ és a ( közvetlenül egymás után áll! }
| ( ⟨ kifejezés 1200 ⟩ ) | { ⟨ kifejezés 1200 ⟩ }
| ⟨ lista ⟩ | ⟨ füzér ⟩
| ⟨ név ⟩ | ⟨ szám ⟩ | ⟨ változó ⟩

# Kifejezések szintaxisa — folytatás

⟨ op *N T* ⟩ ::= ⟨ név ⟩ {feltéve, hogy ⟨ név ⟩ *N* prioritású és *T* típusú operátornak lett deklarálva}

⟨ argumentumok ⟩ ::= ⟨ kifejezés 999 ⟩
| ⟨ kifejezés 999 ⟩ `,` ⟨ argumentumok ⟩

⟨ lista ⟩ ::= `[]`
| `[` ⟨ listakif ⟩ `]`

⟨ listakif ⟩ ::= ⟨ kifejezés 999 ⟩
| ⟨ kifejezés 999 ⟩ `,` ⟨ listakif ⟩
| ⟨ kifejezés 999 ⟩ | ⟨ kifejezés 999 ⟩

⟨ szám ⟩ ::= ⟨ előjeltelen szám ⟩
| + ⟨ előjeltelen szám ⟩
| − ⟨ előjeltelen szám ⟩

⟨ előjeltelen szám ⟩ ::= ⟨ természetes szám ⟩
| ⟨ lebegőpontos szám ⟩

# Kifejezések szintaxisa — megjegyzések

- A ⟨ kifejezés N ⟩-ben ⟨ köz ⟩ csak akkor kell ha az őt követő kifejezés nyitó-zárójellel kezdődik.
- A `{` ⟨ kifejezés ⟩ `}` azonos a `{}` `(` ⟨ kifejezés ⟩ `)` struktúrával, ez pl a DCG nyelvtanoknál hasznos.
- Egy ⟨ füzér ⟩ `"` jelek közé zárt karaktersorozat, általában a karakterek kódjainak listájával azonos.

```
| ?- op(500, fx, succ).
yes
| ?- write_canonical(succ (1,2)), nl, write_canonical(succ(1,2)).
succ(',',(1,2))
succ(1,2)
yes
| ?- write("baba").
[98,97,98,97]
```

# A Prolog lexikai elemei 1. (ismétlés)

- ⟨ név ⟩
  - kisbetűvel kezdődő alfanumerikus jelsorozat (ebben megengedve kis- és nagybetűt, számjegyeket és aláhúzásjelet);
  - egy vagy több ún. speciális jelből (`+-*/\$^<>=`~:.?@#&`) álló jelsorozat;
  - az önmagában álló `!` vagy `;` jel;
  - a `[]` `{}` jelpárok;
  - idézőjelek (`'`) közé zárt tetszőleges jelsorozat, amelyben `\` jellel kezdődő escape-szekvenciákat is elhelyezhetünk.
- ⟨ változó ⟩
  - nagybetűvel vagy aláhúzással kezdődő alfanumerikus jelsorozat.
  - az azonos jelsorozattal jelölt változók egy klózon belül azonosaknak, különböző klózokban különbözőeknek tekintődnek;
  - kivétel: a semmis változók (_) minden előfordulása különböző.

# A Prolog lexikai elemei 2.

- ⟨ természetes szám ⟩
  - (decimális) számjegysorozat;
  - 2, 8 ill. 16 alapú számrendszerben felírt szám, ilyenkor a számjegyeket rendre a `0b`, `0o`, `0x` karakterekkel kell prefixálni (csak `iso` módban)
  - karakterkód-konstans `0'`*c* alakban, ahol *c* egyetlen karakter
- ⟨ lebegőpontos szám ⟩
  - mindenképpen tartalmaz tizedespontot
  - mindkét oldalán legalább egy (decimális) számjeggyel
  - `e` vagy `E` betűvel jelzett esetleges exponens

# Megjegyzések és formázó-karakterek

- Megjegyzések (comment)
  - A `%` százalékjeltől a sor végéig
  - A `/*` jelpártól a legközelebbi `*/` jelpárig.
- Formázó elemek
  - szóköz, újsor, tabulátor, stb. (nem látható karakterek)
  - megjegyzés
- A programszöveg formázása
  - formázó elemek (szóköz, újsor, stb.) szabadon elhelyezhetők;
  - kivétel: struktúrakifejezés neve után nem szabad formázó elemet tenni;
  - prefix operátor és `(` közé kötelező formázó elemet tenni;
  - ⟨záró-pont⟩: egy `.` karakter amit egy formázó elem követ.

# TÍPUSOK PROLOGBAN

# Típusok leírása Prologban

- Típusleírás: (tömör) Prolog kifejezések egy halmazának megadása
- Alaptípusok leírása: `integer, float, number, atom, any`
- Új típusok felépítése:

  `{ str(`$T_1$`, ..., `$T_n$`) }` $\equiv \{$ `str(`$e_1$`, ..., `$e_n$`)` $\mid$ $e_1 \in T_1$, ..., $e_n \in T_n$ $\}, n \geq 0$

  Példa: `{személy(atom,atom,integer)}` az olyan `személy/3` funktorú struktúrák halmaza, amelyben az első két argumentum atom, a harmadik egész.
- Típusok, mint halmazok uniója képezhető a `\/` operátorral.

  ```
  {személy(atom,atom,integer)} \/ {atom-atom} \/ atom
  ```
- Egy típusleírás elnevezhető (kommentben): `% :- type` *tnév* `==` *tleírás*`.`

  ```
  % :- type t1 == {atom-atom} \/ atom.,
  % :- type ember == {ember-atom} \/ {semmi}.
  ```
- Megkülönböztetett únió: csupa különböző funktorú összetett típus uniója. Egyszerűsített jelölés:

  `:- type` $T$ `==` `{` $S_1$ `}` `\/ ...\/ {` $S_n$ `}.` $\Rightarrow$ `:- type` $T$ `--->` $S_1$ `;` `...;` $S_n$`.`

  ```
  % :- type ember ---> ember-atom; semmi.
  % :- type egészlista ---> []; [integer|egészlista].
  ```

# Típusok leírása Prologban — folytatás

- Paraméteres típusok — példák

```
% :- type list(T) ---> [] ; [T|list(T)]. % T típusú elemekből álló lista.
% :- type pair(T1, T2) ---> T1 - T2.     % egy  '-' nevű kétarg.-ú struktúra,
                                         % első arg. T1, a második T2 típusú.
% :- type assoc_list(KeyT, ValueT)
%           == list(pair(KeyT, ValueT)). % KeyT és ValueT típusú
                                         % párokból álló lista.
% :- type szótár == assoc_list(szó, szó).
% :- type szó == atom.
```

- Típusdeklarációk szintaxisa

⟨ típusdeklaráció ⟩ ::= ⟨ típuselnevezés ⟩ | ⟨ típuskonstrukció ⟩
⟨ típuselnevezés ⟩ ::= `:- type` ⟨ típusazonosító ⟩ == ⟨ típusleírás ⟩ .
⟨ típuskonstrukció ⟩ ::= `:- type` ⟨ típusazonosító ⟩ `--->` ⟨ megkülönb. únió ⟩ .
⟨ megkülönb. únió ⟩ ::= ⟨ konstruktor ⟩ `;` ...
⟨ konstruktor ⟩ ::= ⟨ névkonstans ⟩ | ⟨ struktúranév ⟩ `(` ⟨ típusleírás ⟩, ... `)`
⟨ típusleírás ⟩ ::= ⟨ típusnév ⟩ | ⟨ típusváltozó ⟩ |
⟨ típusleírás ⟩ `\/` ⟨ típusleírás ⟩ |
`{` ⟨ típusnév ⟩ `(` ⟨ típusleírás ⟩, ... `)` `}`
⟨ típusazonosító ⟩ ::= ⟨ típusnév ⟩ | ⟨ típusnév ⟩ `(` ⟨ típusváltozó ⟩, ... `)`

# Predikátum-deklarációk

- Predikátumtípus-deklaráció
  `:- pred` ⟨ eljárásnév ⟩ `(` ⟨ típusazonosító ⟩, ...`)`
- Példák:

```
:- pred member(T, list(T)).
:- pred append(list(T), list(T), list(T)).
```

- Predikátummód-deklaráció (Nem kötelező, több is megadható.)
  `:- mode` ⟨ eljárásnév ⟩ `(` ⟨ módazonosító ⟩, ...`)` ahol ⟨ módazonosító ⟩ ::= `in` | `out`.
- Példák:

```
:- mode append(in, in, in).   % ellenőrzésre
:- mode append(in, in, out).  % két lista összefűzésére
:- mode append(out, out, in). % egy lista szétszedésére
```

- Vegyes típus- és móddeklaráció
  `:- pred` ⟨ eljárásnév ⟩ `(` ⟨ típusazonosító ⟩ `::` ⟨ módazonosító ⟩, ...`)`
- Példa:

```
:- pred between(integer::in, integer::in, integer::out).
```

# Móddeklaráció: a SICStus kézikönyv által használt alak

- A SICStus kézikönyv egy másik jelölést használ a bemenő/kimenő argumentumok jelzésére, pl.

```
append(+L1, ?L2, -L3).
append(?L1, ?L2, +L3).
```

- Mód-jelölő karakterek:
  - `+` bemenő argumentum (behelyettesített)
  - `-` kimenő argumentum (behelyettesítetlen)
  - `:` eljárás-paraméter (meta-eljárásokban)
  - `?` tetszőleges

# II. RÉSZ

# SML-Prolog átvezetés: párhuzamok a két nyelv között

**SML**

```
fun append ([], ys) = ys
  | append (x::xs, ys) =
                x::append (xs, ys)
```

**SML „Prologosítva”**

```
fun append([], L) = L
  | append(X::L1, L2) =
    let val L3 = append(L1, L2)
    in  X::L3 end
```

függvény
klóz
változó: egyetlen, ismert érték
minta: csak fordítási időben értelmes
egyszerű minták: `x::x::xs` nem megengedett
egyirányú mintaillesztés
egyértelmű klózválasztás
egy eredmény
egyirányú használat

adatkonstruktor-függvény
egymásba ágyazott függvényhívások

**Prolog**

```
append([], L, L).
append([X|L1], L2, [X|L3]) :-
        append(L1, L2, L3).
```

**Prolog „SML-esítve”**

```
append([], L, Res) :- Res = L.
append([X|L1], L2, Res) :-
        append(L1, L2, L3),
        Res = [X|L3].
```

predikátum
klóz (lazább a kapcsolat a predikátummal)
változó: egy, esetleg ismeretlen érték
minta: teljes jogú adatstruktúra
összetett minták, pl. `[X,X|Xs]`
kétirányú mintaillesztés
többértelmű klózválasztás
több eredmény (nemdeterminizmus)
többirányú használat
(pl. összerakó és szétszedő `append`)
struktúra (rekord)
konjunkció, segéd-változóval

# SML-Prolog átvezetés: további példák

**SML**

```
fun append xs ys = foldr op:: ys xs
```

```
fun fakt 0 = 1
  | fakt n = n * fakt (n-1)
```

típusos nyelv
magasabbrendű függvény
rekurzió

kivétel

**Prolog**

```
/* Prologban kevésbé használtak
   a magasabbrendű eljárások */
```

```
fakt(0, 1).
fakt(N, F) :-
    N>0, N1 is N-1,
    fakt(N1, F1), F is N*F1.
```

típustalan nyelv
rekurzió, ritkábban magasabbrendű predikátum
visszalépéses ciklus
(pl. két lista közös eleme)
meghiúsulás, kivétel

# Összefoglalás: Prolog programok szemantikája

- Prolog program jelentése = milyen válaszokat (behelyettesítéseket) kapunk egy cél futtatásakor:
  - Procedurális szemantika — az ismertetett végrehajtási, egyesítési algoritmus.
  - Deklaratív szemantika:
    - program: logikai állítások (klózok, azaz implikációk) halmaza;
    - egy cél futási eredménye: olyan behelyettesítés, amelyre a cél **következménye** a programnak.
- A Prolog procedurális szemantika csak olyan választ ad, amely a deklaratív szemantika szerint is helyes! (Ha predikátumaink „igazak”, akkor rossz eredményt nem kaphatunk, csak végtelen ciklust. `:-()`

# Ismétlés: A Prolog végrehajtási mechanizmus, dióhéjban

- (Kezdet:) Ha célsorozat üres $\rightarrow$ sikeres lefutás.
- (Folytatás:) Keresünk az **első** céllal egyesíthető klózfejet (a klózból friss másolatot képezve, felülről lefelé haladva a programbeli klózokon ).
- Ha van ilyen:
  - Ha van esély további illesztésre, akkor választási pontot hozunk létre: a futás jelenlegi állapotát (célsorozat + hányadik klózzal illesztettünk) megjegyezzük, azaz a veremre rakjuk.
  - Az egyesítéshez szükséges behelyettesítéseket a klóztörzsön és a célsorozaton is elvégezzük.
  - Az első cél helyébe a klóztörzset rakjuk, ez lesz az új célsorozat, majd vissza a (Kezdet)-hez.
- Ha nincs illeszthető klózfej, akkor visszalépünk a **legutolsó** választási pontnak megfelelő állapotba (azt leemelve a verem tetejéről), és új egyesíthető fejű klóz keresésével folytatjuk a (Folytatás)-nál.

# Ismétlés: A Prolog egyesítési algoritmus, dióhéjban

- Legáltalánosabb egyesítő behelyettesítés meghatározása
  - Azonos változók ill. konstansok behelyettesítés nélkül egyesíthetőek.
  - Változó minden más kifejezéssel egyesíthető, triviális behelyettesítéssel (tartalmazás-vizsgálat nélkül)
  - Két összetett kifejezés egyesíthető, ha funktoraik azonosak, és az argumentumaik sorra egyesíthetőek, úgy, hogy a megelőző argumentumok egyesítéséhez szükséges behelyettesítéseket már elvégeztük. Az argumentumok egyesítését biztosító behelyettesítések kompozíciója a legáltalánosabb egyesítő.
  - Minden más esetben a két kifejezés nem egyesíthető, az egyesítési algoritmus meghiúsul.

# 4. fejezet: Prolog programozási módszerek

- Az előző előadás-blokk (jegyzetbeli 3. fejezet) célja volt:
  - a Prolog nyelv alapjainak bemutatása,
  - a logikailag „tiszta” résznyelvre koncentrálva.
- A jelen előadás-blokk (jegyzetben a 4. fejezet) célja: olyan
  - beépített eljárások,
  - programozási technikák

  bemutatása, amelyekkel
  - hatékony Prolog programok készíthetők,
  - esetleg a tiszta logikán túlmutató eszközök alkalmazásával.

# Prolog programozási módszerek: tartalomjegyzék

- A keresési tér szűkítése
- Vezérlési eljárások
- Determinizmus és indexelés
- Jobbrekurzió és akkumulátorok
- Algoritmusok Prologban
- Megoldások gyűjtése és felsorolása
- Megoldásgyűjtő eljárások
- Meta-logikai eljárások
- Modularitás
- Magasabbrendű eljárások
- Dinamikus adatbáziskezelés
- Nyelvtani elemzés
- „Hagyományos” beépített eljárások

# A KERESÉSI TÉR SZŰKÍTÉSE

# Prolog nyelvi eszközök a keresési tér szűkítésére

- Eszközök
  - a vágó beépített eljárás: `!` (az első Prolog rendszerektől kezdve)
  - feltételes diszjunktív szerkezet (későbbi kiterjesztés): `( if -> then ; else )`
- Feltételes szerkezet — procedurális szemantika (ismétlés)
  A `(felt->akkor;egyébként),folyt` célsorozat végrehajtása:
  - Végrehajtjuk a `felt` hívást.
  - Ha `felt` sikeres, akkor az `akkor,folyt` célsorozatra redukáljuk a fenti célsorozatot, a `felt` **első** megoldása által eredményezett behelyettesítésekkel. A `felt` cél **többi megoldását nem keressük meg**.
  - Ha `felt` sikertelen, akkor az `egyébként,folyt` célsorozatra redukáljuk.
- Feltételes szerkezet — alternatív procedurális szemantika:
  - Diszjunkciónak tekintjük: redukáljuk a `felt, {vágás}, akkor, folyt` célsorozatra és létrehozunk egy $VP$ választási pontot (`egyébként, folyt` célsorozattal).
  - Ha `felt` sikerül, azaz eljutunk **{vágás}**-hoz, akkor minden választási pontot megszüntetünk, egészen $VP$-ig, azt is beleértve.
  - (Ha `felt` nem sikerül, a diszjunkció másik ágával folytatjuk.)

# Feltételes szerkezet: választási pontok a feltételben

- Példafeladat: `első_poz_elem(L, P)`: `P` az `L` lista első pozitív eleme.
- Első megoldás, rekurzióval (mérnöki `:-)`)

```
első_poz_elem([EP|_], EP) :- EP > 0.
első_poz_elem([X|L], EP) :- X =< 0, első_poz_elem(L, EP).
```

- Második megoldás, visszalépéses kereséssel (matematikusi `:-)`)

```
első_poz_elem(L, EP) :-
        append(Nk, [EP|_], L), EP > 0, \+ van_poz_eleme(Nk).

van_poz_eleme(L)  :- member(P, L), P > 0.
```

- Harmadik megoldás, feltételes szerkezettel (gyorsprogramozás — Prolog hekker `:-)`)

```
első_poz_elem(L, EP) :-
        (   member(EP, L), EP > 0 -> true
        ;   fail                              % ez a sor elhagyható
        ).
```

- Figyelem: a harmadik megoldás épít a `member/2` felsorolási sorrendjére!

# A vágó eljárás

- A vágó beépített eljárás (neve: `!`) végrehajtása: letiltja a a többi klóz választását és megszünteti az összes választási pontot a klóztörzsben őt megelőző eljáráshívásokban.
- Példák a vágó használatára (lista első pozitív eleme)
  - Mérnöki megoldás:

```
első_poz_elem([EP|_], EP) :- EP > 0, !.
első_poz_elem([X|L], EP) :- X =< 0, első_poz_elem(L, EP).
```

  - Prolog hekker megoldása:

```
első_poz_elem(L, EP) :-  member(EP, L), EP > 0, !.
```

- Miért vágunk le ágakat a keresési térben?
  - mert mi tudjuk, hogy ott nincs megoldás, de a Prolog megvalósítás nem — zöld vágás, szemantikailag „ártalmatlan”
    - (Például, a legtöbb Prolog megvalósítás „nem tudja”, hogy a $X > 0$ és $X \leq 0$ feltételek kizárják egymást, lásd indexelés.)
  - ténylegesen eldobunk megoldásokat — vörös vágás, a program jelentését megváltoztatja
    - (Vörös vágás sokszor úgy keletkezik, hogy egy zöld vágót tartalmazó programban a „felesleges” feltételeket elhagyjuk (pl. az `X =< 0` feltételt a fenti 2. klózban)

# Példák a vágó eljárás használatára

```
% fakt(+N, ?F): N! = F.
fakt(0, 1) :- !.                                    % zöld vágó
fakt(N, F) :- N > 0, N1 is N-1, fakt(N1, F1), F is N*F1.

% last(+L, ?E): L utolsó eleme E. (lists könyvtárbeli)
last([E], E) :- !.                                  % zöld vágó
last([_|L], E) :- last(L, E).

% pozitívak(+L, -P): P az L pozitív elemeiből áll.
pozitívak([], []).
pozitívak([E|Ek], [E|Pk]) :-
        E > 0, !,                                   % vörös vágó
        pozitívak(Ek, Pk).
pozitívak([_E|Ek], Pk) :-
        /* \+ _E > 0, */ pozitívak(Ek, Pk).
```

Figyelem: a fenti példák nem tökéletesek, hatékonyabb ill. általánosabban használható változatukat később ismertetjük!

# A vágó definíciója

## Segédfogalom

- Egy cél **szülője** az a cél, amelyik az őt tartalmazó klóz fejével illesztődött.
- Pl. a `last([E], E) :- !.` klózbeli vágó szülője lehet a `last([7], X)` hívás.
- A `g` (ancestors) nyomkövetési parancs kiírja a kurrens cél őseit (szülőjét, annak szülőjét stb.)

## A vágó végrehajtása:

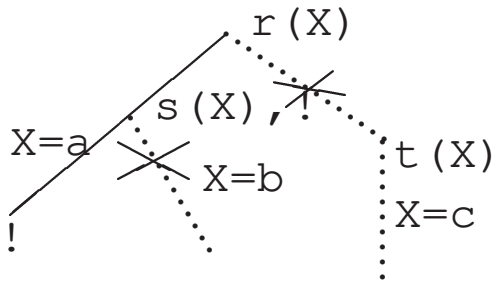
- mindig sikerül; és a végrehajtás adott állapotától visszafelé egészen a szülő célig, azt is beleértve, minden választási pontot megszüntet.

## A vágás kétféle választási pontot szüntet meg:

```
r(X):- s(X), !.   % az s(X)-beli választási pontokat --- a vágót megelőző
                  % cél(ok)nak az első megoldására szorítkozunk
r(X):- t(X).      % az r(X) többi klózának választását --- a vágót tartalmazó
                  % klóz mellett kötelezzük el magunkat (commit)
```

# A vágó által megszüntetett választási pontok

```
% vágó nélküli példa
q(X):- s(X).
q(X):- t(X).

% ugyanaz a példa vágóval
r(X):- s(X), !.
r(X):- t(X).

s(a).      s(b).      t(c).

% a vágó nélküli példa futása
:- q(X), write(X), fail.
                  --->        abc
% a vágót tartalmazó példa futása
:- r(X), write(X), fail.
                  --->        a
```



# A diszjunktív feltételes szerkezet megvalósítása a vágó segítségével

- A diszjunktív feltételes szerkezet, a diszjunkcióhoz hasonlóan egy segédeljárással váltható ki:

```
p :-                                  p :-
    ...                                   ...
    (   felt1 -> akkor1                   segéd(...)
    ;   felt2 -> akkor2                  ... .
    ;   ...                    ⟹
    ;   egyébként                    segéd(...) :- felt1, !, akkor1.
    )                                 segéd(...) :- felt2, !, akkor2.
    ... .                             ...
                                      segéd(...) :- egyébként.
```

- Az `egyébként` alternatíva elmaradhat, ilyenkor a megfelelő klóz is elmarad.
- SICStus módban a `felt` részekben vágó nem lehet, ISO módban lehet, de hatásköre (szülője) a `felt` rész.
- Az `akkor` részekben lehet vágó. Ennek hatásköre, a `->` nyílból generált vágóval ellentétben, a teljes `p` predikátum (ilyenkor a Prolog megvalósítás egy speciális, ún. távolbaható vágót használ).
- Vágót rendkívül ritkán szükséges feltételes szerkezetben szerepeltetni.

# Példák a diszjunktív feltételes szerkezet használatára

```
% fakt(+N, ?F): N! = F.
fakt(N, F) :-
        (   N = 0 -> F = 1
        ;   N > 0, N1 is N-1, fakt(N1, F1), F is N*F1
        ).

% last(+L, ?E): az L nem üres lista utolsó eleme E.
last([E|L], Last) :-
        (   L = [] -> Last = E
        ;   last(L, Last)
        ).

% pozitívak(+L, ?Pk): Pk az L pozitív elemeiből áll.
pozitívak([], []).
pozitívak([E|Ek], Pk) :-
        (   E > 0 -> Pk = [E|Pk0]
        ;   Pk = Pk0
        ),
        pozitívak(Ek, Pk0).
```

# A vágás első alapesete — klóz mellett való elkötelezés

- A klóz melletti elkötelezés általában egyszerű feltételes szerkezetet jelent.

```
szülő :- feltétel, !, akkor.
szülő :- egyébként.
```

- A vágó szükségtelenné teszi a `feltétel` negációjának végrehajtását a többi klózban. A logikailag tiszta, de nem hatékony alak:

```
szülő :- feltétel, akkor.
szülő :- \+ feltétel, egyébként.
```

  A fenti két alak csak akkor ekvivalens, ha `feltétel` egyszerű, nincs benne választás.

- Analógia: ha `a, b` és `c` logikai változók (pl. SML-ben), akkor
  `if a then b else c` $\equiv a \wedge b \vee \neg a \wedge c$

- A vágó által kiváltott negált feltételt célszerű kommentként jelezni:

```
szülő :- feltétel, !, akkor.
szülő :- /* \+ feltétel, */ egyébként.
```

# Feltételes szerkezetek

## Feltételes szerkezet — példa

```
% abs(X, A): A az X abszolút értéke.
abs(X, A)     :- X < 0, !, A is -X.
abs(X, X) /* :- X >= 0 */.
```

## Diszjunktív feltételes szerkezet

```
abs(X, A) :-
        (   X < 0 -> A is -X
        ;   A = X
        ).
```

## Általános alak

```
p :- felt1, !, akkor1.
p :- felt2, !, akkor2.
...
p :- egyébként.
```

## Általános alak

```
p :-
        (   felt1 -> akkor1
        ;   felt2 -> akkor2
        ;   ...
        ;   egyébként
        ).
```

# Feltételes szerkezetek és fejillesztés

- Vigyázat: a tényleges feltétel részét képezik a fejbeli egyesítések!

```
% a vágó előtt fej-egyesítés:      % az egyesítés explicitté téve:
abs(X, X) :- X >= 0, !.            abs(X, A) :- A = X, X >= 0, !.
abs(X, A) :- A is -X.              abs(X, A) :- A is -X.
```

- A fej-egyesítés gondot okozhat, ha az eljárást ellenőrzésre használjuk:

```
| ?- abs(10, -10).  --->  yes
```

- A megoldás a **vágás alapszabálya**:
  - A kimenő paraméterek értékadását mindig a vágó után végezzük!

```
abs(X, A) :- X >= 0, !, A = X.
abs(X, A) :- A is -X.
```

  - Ez nemcsak általánosabban használható, hanem hatékonyabb kódot is ad: csak akkor helyettesíti be a kimenő paramétert, ha már tudja, mi az értéke (nincs „előre-behelyettesítés”, mint a fenti első két példában).
  - (**„kimenő” paraméterek** — vágó alkalmazásakor általában nincs többirányú használat :-)

# A bevezető példáknak a vágás alapszabályát betartó változata

```
% fakt(+N, ?F): N! = F.
fakt(0, F) :- !, F = 1.
fakt(N, F) :- N > 0, N1 is N-1, fakt(N1, F1), F is N*F1.

% last(+L, ?E): az L nem üres lista utolsó eleme E.
last([E], Last) :- !, Last = E.
last([_|L], E)  :- last(L, E).

% pozitívak(+L, ?Pk): Pk az L pozitív elemeiből áll.
pozitívak([], []).
pozitívak([E|Ek], Pk) :-
        E > 0, !, Pk = [E|Pk0], pozitívak(Ek, Pk0).
pozitívak([_E|Ek], Pk) :-
        /* \+ _E > 0, */ pozitívak(Ek, Pk).
```

**Megjegyzés:** a diszjunktív alakban a feltételek eleve explicitek, nincs fejillesztési probléma, ezért **a diszjunktív feltételes szerkezet használatát javasoljuk a vágó helyett**.

# Példasor: `max(X, Y, Z)`: `X` és `Y` maximuma `Z`.

- 1. változat, tiszta Prolog. Lassú (előre-behelyettesítés, két hasonlítás), választási pontot hagy.

```
max(X, Y, X) :- X >= Y.
max(X, Y, Y) :- Y > X.
```

- 2. változat, zöld vágóval. Lassú (előre-behelyettesítés, két hasonlítás), nem hagy választási pontot.

```
max(X, Y, X) :- X >= Y, !.
max(X, Y, Y) :- Y > X.
```

- 3. változat, vörös vágóval. Gyorsabb (előre-behelyettesítés, egy hasonlítás), nem hagy választási pontot, de nem használható ellenőrzésre, pl. `?- max(10, 1, 1)` sikerül.

```
max(X, Y, X) :- X >= Y, !.
max(X, Y, Y).
```

- 4. változat, vörös vágóval. Helyes, nagyon gyors (egy hasonlítás, nincs előre-behelyettesítés) és nem is hoz létre választási pontot.

```
max(X, Y, Z) :- X >= Y, !, Z = X.
max(X, Y, Y) /* :- Y > X */.
```

# A vágás második alapesete — első megoldásra való megszorítás

## Mikor használjuk az első megoldásra megszorító vágót?

- behelyettesítést nem okozó, eldöntendő kérdés esetén;
- feladatspecifikus optimalizálásra;
- végtelen választási pontot létrehozó eljárások hasznosítására.

## Eldöntendő kérdés: eljáráshívás csupa bemenő paraméterrel

```
% van_elég_hosszú_út(+N, +A, +B, +Min):
% A és B között van N lépéses út,
% amelynek összhossza legalább Min km.
van_elég_hosszú_út(N, A, B, Min) :-
        útvonal(N, A, B, Hossz), Hossz >= Min, !.
```

Eldöntendő kérdés esetén általában nincs értelme többszörös választ adni/várni.

# Feladatspecifikus optimalizálás

A feladat: megkeresendő egy lista elején álló „plató” hossza (platónak hívjuk a csupa azonos elemből álló folytonos részlistát).

```
% Az L lista első eleme H-szor ismétlődik
% a lista kezdőszeleteként.
kezdethossz(L, H) :-
        L = [E|_], append(Ek, Farok, L),
        \+ Farok = [E|_], !,              % vörös vágó
        /* egyformák(Ek, E), */
        length(Ek, H).
/*
% egyformák(Ek, E): Az Ek lista minden eleme E.
egyformák([], _).
egyformák([E|Ek], E) :-
        egyformák(Ek, E).
*/
| ?- kezdethossz([1,1,1,2,3,5], H).
H = 3 ? ; no
```

# Végtelen választás megszelidítése: `memberchk` (`lists` könyvtár)

```
% memberchk(X, L): "X eleme az L listának" kérdés első megoldása.

% 1. változat
memberchk(X, L):-
        member(X, L), !.
```

```
% 2. ekvivalens változat
memberchk(X, [X|_]) :- !.
memberchk(X, [_|L]) :-
        memberchk(X, L).
```

## `memberchk/2` használata

- Eldöntő kérdésben (visszalépéskor nem keresi végig a lista maradékát.)

  ```
  memberchk(1, [1,2,3,4,5,6,7,8,9])
  ```

- Nyílt végű lista elemévé tesz, pl.:

  ```
  | ?- memberchk(1,L), memberchk(2,L), memberchk(1,L).
          L = [1,2|_A] ?
  ```

# Nyílt végű listák kezelése `memberchk` segítségével: szótárprogram

```
szótaraz(Sz):-
        read(M-A), !,
        % A read(X) beépített eljárás egy kifejezést
        % olvas be és egyesíti X-szel
        memberchk(M-A,Sz),
        write(M-A), nl,
        szótaraz(Sz).
szótaraz(_).
```

Egy futása:

```
| ?- szótaraz(Sz).
|: alma-apple.
alma-apple
|: korte-pear.
korte-pear
```

```
|: alma-X.
alma-apple
|: X-pear.
korte-pear
|: vege.    % nem egyesíthető M-A-val

Sz = [alma-apple,korte-pear|_A] ?
```

# VEZÉRLÉSI ELJÁRÁSOK

# Vezérlési eljárások, a `call/1` beépített eljárás

- Vezérlési eljárás: A Prolog végrehajtási mechanizmusához kapcsolódó beépített eljárás (pl. a vágó).
- A vezérlési eljárások többsége **magasabbrendű** eljárás, azaz olyan eljárás, amely egy vagy több argumentumát eljáráshívásként értelmezi. (A magasabbrendű Prolog eljárásokat szokás **meta-eljárásnak** is hívni.)
- A meta-eljárások fő képviselője és alapvető építőeleme a `call/1`:
  - Hívási minta: `call(+Cél)`
  - `Cél` egy „meghívható kifejezés” (callable, vö. `callable/1`), azaz struktúra, vagy atom.
  - Jelentése (deklaratív szemantika): `Cél` igaz.
  - Hatása (procedurális szemantika): a `Cél` kifejezést eljáráshívássá alakítja és meghívja.
- A klóztörzsben célként megengedett egy `X` változó használata, ezt a rendszer egy `call(X)` hívássá alakítja át.

```
kétszer(Hívás) :- call(Hívás), Hívás.

?- kétszer(write(ba)), nl.        --->  baba
?- listing(kétszer).              --->  kétszer(A) :-
                                            call(user:A), call(user:A).
```

# Vezérlési szerkezetek mint eljárások

- A `call/1` argumentumában szerepelhetnek vezérlési szerkezetek is, mert ezek maguk beépített eljárásként is jelen vannak a Prolog rendszerben:
  - `(',')/2`: konjunkció.
  - `(;)/2`: diszjunkció.
  - `(->)/2`: if-then.
  - `(;)/2`: if-then-else.
- A `call`-ban szereplő vezérlési szerkezetek lényegében ugyanúgy futnak, mint az interpretált (`consult`-tal betöltött) kód.
- Példák:

```
| ?- _Cél = (kétszer(write(ba)), write(' ')), kétszer(_Cél), nl.
baba baba
| ?- kétszer((member(X, [a,b,c,d]), write(X), fail ; nl)).
abcd
abcd
```

# `call/1` példa: futási időt mérő meta-eljárás

```
% Kiírja Goal első megoldásának előállításához vagy a meghiúsuláshoz
% szükséges időt, a Txt szöveg kiséretében (lásd: peldak/call_koltsege.pl).
time(Txt, Goal) :-
        statistics(runtime, [T0,_]),  % T0 az indítás óta eltelt CPU idő,
                                      % msec-ban (szemétgyűjtés nélkül).
        (   call(Goal) -> Res = true
        ;   Res = false
        ),
        statistics(runtime, [T1,_]), T is T1-T0,
        format('~w futási idő = ~3d sec\n', [Txt,T]),
                    % ~w formázó: kiírás a write/1 segítségével
                    % ~3d formázó: I egész kiírása I/1000-ként, 3 tizedesre
        Res = true.
```

A `call/1` viszonylag költséges: egy 1414 hosszú lista megfordítása `nrev`-vel (kb. 1 millió `append` hívás), minden `append` körül egy felesleges `call`-lal ill. anélkül:

| | `call` nélkül | `call`-lal | Lassulás |
|---|---|---|---|
| lefordítva | 0.140 sec | 1.680 sec | 12.00 |
| interpretálva | 1.710 sec | 3.520 sec | 2.06 |

# További beépített vezérlési eljárások

- `\+ Cél`: `Cél` „nem bizonyítható”. A beépített eljárás definíciója:

```
\+ X :- call(X), !, fail.
\+ _X.
```

- `once(Cél)`: `Cél` igaz, és csak az első megoldását kérjük. Definíciója:

```
once(X) :- call(X), !.
```

- `true`: azonosan igaz (mindig sikerül), `fail`: azonosan hamis (mindig meghiúsul).
- `repeat`: végtelen sokszor igaz (egy végtelen választási pontot hoz létre). Definíciója:

```
repeat.
repeat :- repeat.
```

- A `repeat` eljárást legtöbbször egy mellékhatásos eljárás ismétlésére használjuk. A végtelen választási pontot kötelező egy vágóval semlegesíteni.
- Példa (egyszerű kalkulátor):

```
bc :-  repeat, read(Expr),
            (   Expr = end_of_file -> true
            ;   Res is Expr, write(Expr = Res), nl, fail
            ),
       !.
```

# Példa: magasabbrendű reláció definiálása

- Az implikáció (`P` $\Rightarrow$ `Q`) megvalósítása negáció segítségével:

```
% P minden megoldása esetén Q igaz.
forall(P, Q) :-
     \+ (P, \+Q). % Szintaktikus emlékeztető:
                  % az első \+ után kötelező a szóköz!

| ?- _L = [1,2,3],
     % _L minden eleme pozitív:
     forall(member(X, _L), X > 0).
true ?
| ?- _L = [1,-2,3], forall(member(X, _L), X > 0).
no
| ?- _L = [1,2,3],
     % _L szigorúan monoton növő:
     forall(append(_,[A,B|_], _L), A < B).
true ?
```

- `forall/2` csak eldöntendő kérdés esetén használható.

# DETERMINIZMUS ÉS INDEXELÉS

# Determinizmus

- Egy eljáráshívás **determinisztikus**, ha (legfeljebb) egyféleképpen sikerülhet.
- Egy eljáráshívásnak egy sikeres végrehajtása **determinisztikusan futott le**:
  - ha nem hagyott választási pontot a híváshoz tartozó részfában, azaz
    - vagy választásmentesen futott le, azaz létre sem hozott választási pontot (figyelem: ez a Prolog megvalósítástól függ!);
    - vagy létrehozott ugyan választási pontot, de megszüntette (kimerítette, levágta).
- A SICStus Prolog nyomkövetésében **?** jelzi a **nem**determinisztikus lefutást:

```
p(1, a).
p(2, b).
p(3, b).
```

```
| ?- p(1, X).                          % det. hívás,
                1 1 Exit: p(1,a)       %      det. lefutás
| ?- p(Y, a).                          % det. hívás,
              ? 1 1 Exit: p(1,a)       %      nemdet. lefutás
| ?- p(Y, b), Y > 2.                   % nemdet. hívás
              ? 1 1 Exit: p(2,b)       %      nemdet. lefutás
                1 1 Exit: p(3,b)       %      det. lefutás
```

# A determinisztikus lefutás

- Mi a determinisztikus lefutás haszna?
  - a futás gyorsabb lesz,
  - a tárigény csökken,
  - más optimalizálások (pl. jobbrekurzió) alkalmazható.
- Hogyan ismeri fel a fordító azt, hogy nem kell választási pont?
  - indexelés (indexing)
  - vágók és feltételes szerkezetek
- Az alábbi definíciók esetén a `p(`*`Nonvar`*`, Y)` hívás nem hoz létre választási pontot:

```
p(1, a).
p(2, b).
```

```
p(1, Y) :- !,
    Y = a.
p(_, b).
```

```
p(X, Y) :-
    (   X =:= 1 -> Y = a
    ;   Y = b
    ).
```

# Indexelés — ismétlés

- Mi az indexelés?
  - egy adott hívásra illeszthető klózok gyors kiválasztása,
  - egy eljárás klózainak fordítási idejű csoportosításával.
- A legtöbb Prolog rendszer, így a SICStus Prolog is, az első fej-argumentum alapján indexel (first argument indexing).
- Az indexelés alapja az első fejargumentum külső funktora:
  - `C` szám vagy névkonstans esetén `C/0`;
  - `R` nevű és `N` argumentumú struktúra esetén `R/N`;
  - változó esetén nem értelmezett.
- Az indexelés megvalósítása:
  - Fordításkor a funktorokhoz elkészítjük az illeszthető klózok részhalmazát.
  - Futáskor lényegében konstans idő alatt választunk a részhalmazok közül.
  - **Fontos:** ha egyelemű a részhalmaz, nem hozunk létre választási pontot!

# Példa indexelésre (ismétlés)

```
p(0, a).            /* (1) */
p(X, t) :- q(X).    /* (2) */
p(s(0), b).         /* (3) */
p(s(1), c).         /* (4) */
p(9, z).            /* (5) */
```

```
q(1).
q(2).
```

- A `p(A, B)` hívással illesztendő klózhalmaz:
  - `{(1) (2) (3) (4) (5)}` ha `A` változó;
  - `{(1) (2)}` ha `A = 0`;
  - `{(2) (3) (4)}` ha `A` fő funktora `s/1`;
  - `{(2) (5)}` ha `A = 9`;
  - `{(2)}` minden más esetben.
- Példák hívásokra:
  - `p(1, Y)` nem hoz létre választási pontot.
  - `p(s(1), Y)` létrehoz választási pontot, de determinisztikusan fut le.
  - `p(s(0), Y)` nemdeterminisztikusan fut le.

# Struktúrák, változók a fejargumentumban

- Azonos funktorú struktúrák az első fejargumentumban:
  - Ha a klózok szétválasztásához szükség van az első (struktúra) argumentum részeire is, akkor érdemes segédeljárást bevezetni.
  - Például `p/2` és `q/2` ekvivalens, de `q(`*`Nonvar`*`, Y)` determinisztikusan fut le!

```
p(0, a).
p(s(0), b).
p(s(1), c).
p(9, z).
```

```
q(0, a).
q(s(X), Y) :-
     q_seged(X, Y).
q(9, z).
```

```
q_seged(0, b).
q_seged(1, c).
```

- Fejillesztés kiváltása egyenlőséggel (vö. SML rétegelt minta)
  - Az indexelés figyelembe veszi a törzs elején szereplő egyenlőséget:
    `p(X, ...) :- X = Kif, ...` esetén `Kif` funktora szerint indexel.
  - Példa: lista hosszának reciproka, üres lista esetén 0:

```
rhossz([], 0).
rhossz(L, RH) :- L = [_|_], length(L, H), RH is 1/H.
% rhossz([X|L], RH) :-  length([X|L], H), RH is 1/H.
        % kevésbé hatékony, mert újra felépíti az [X|L] listát.
% rhossz(L, RH) :-  L \= [], length(L, H), RH is 1/H.
        % kevésbé hatékony, mert L=[] esetben választási pontot hagy.
```

# Indexelés — további tudnivalók

- Indexelés és aritmetika
  - Az indexelés nem foglalkozik aritmetikai vizsgálatokkal.
  - Pl. az `N = 0` és `N > 0` feltételek nem „zárják ki” egymást.
  - Az alábbi `fakt/2` eljárás lefutása nem-determinisztikus:

```
fakt(0, 1).
fakt(N, F) :- N > 0, N1 is N-1, fakt(N1, F1), F is N*F1.
```

- Indexelés és listák
  - Gyakran kell az üres és nem-üres lista esetét szétválasztani.
  - A bemenő lista-argumentumot célszerű az első argumentum-pozícióba tenni.
  - Az `[]` és `[...|...]` eseteket az indexelés megkülönbözteti (funktoruk: `'[]'/0` ill. `'.'/2`).
  - A két klóz sorrendje nem érdekes (feltéve, hogy zárt listával hívjuk az első pozíción) — de azért tegyük a leálló klózt mindig előre.

# Listakezelő eljárások indexelése: példák

- Az `append/3` választásmentesen fut le (összefűzésre).

```
append([], L, L).
append([X|L1], L2, [X|L3]) :-
    append(L1, L2, L3).
```

- A `last/2` közvetlen megfogalmazása nemdeterminisztikusan fut le:

```
% last(L, E): Az L lista utolsó eleme E.
last([E], E).
last([_|L], E) :- last(L, E).
```

- Érdemes segédeljárást bevezetni, `last2/2` választásmentesen fut

```
last2([X|L], E) :- last2(L, X, E).

% last2(L, X, E): Az [X|L] lista utolsó eleme E.
last2([], E, E).
last2([X|L], _, E) :- last2(L, X, E).
```

# Az indexelés és a vágó kölcsönhatása

- Hogyan vehető figyelembe a vágó az indexelés fordításakor?
- Példa: a `p(1, A)` hívás választásmentes, de a `q(1, A)` nem!

```
p(1, Y) :- !, Y = 2.      % (1)
p(X, X).                  % (2)
Arg1=1 → (1), Arg1≠1 → (2)
```

```
q(1, 2) :- !.             % (1)
q(X, X).                  % (2)
Arg1=1 → {(1),(2)}, Arg1≠1 → (2)
```

- A fordító figyelembe veszi a vágót az indexelésben, ha garantált, hogy egy adott fő funktor esetén a vágót elérjük. Ennek feltételei:
  - az első argumentum változó, konstans, vagy csak változókat tartalmazó struktúra legyen,
  - a további argumentumok változók legyenek,
  - a fejben az összes változóelőfordulás különböző legyen,
  - a törzs első hívása a vágó (megengedve a fejillesztést kiváltó egyenlőséget).
- Ilyenkor a fordító az adott funktorhoz tartozó listából kihagyja a vágót követő klózokat.
- Példa: `p(X, D, E) :- X = s(A, B, C), !, ....  p(X, Y, Z) :- ....`
- Ez egy újabb érv a vágás alapszabálya mellett:

**A kimenő paraméterek értékadását mindig a vágó után végezzük!**

# A vágó és az indexelés hatékonysága

- Egy Fibonacci-szerű sorozat: $f_1 = 1;\ \ f_2 = 2;\ \ f_n = f_{\lfloor 3n/4 \rfloor} + f_{\lfloor 2n/3 \rfloor},\ \ n > 2$

```
% determinisztikus
fib(1, 1).
fib(2, 2).
fib(N, F) :-
        N > 2,
        N2 is N*3//4,
        N3 is N*2//3,
        fib(N2, F2),
        fib(N3, F3),
        F is F2+F3.
```

```
% determ. lefutású
fibc(1, 1) :- !.
fibc(2, 2) :- !.
fibc(N, F) :-
        N > 2,
        N2 is N*3//4,
        N3 is N*2//3,
        fibc(N2, F2),
        fibc(N3, F3),
        F is F2+F3.
```

```
% választásmentes
fibci(1, F) :- !, F = 1.
fibci(2, F) :- !, F = 2.
fibci(N, F) :-
        N > 2,
        N2 is N*3//4,
        N3 is N*2//3,
        fibci(N2, F2),
        fibci(N3, F3),
        F is F2+F3.
```

- Futási idők $N = 2000$ esetén

| | `fib` | `fibc` | `fibci` |
|---|---|---|---|
| futási idő | 990 msec | 890 msec | 830 msec |
| meghiúsulási idő | 440 msec | 30 msec | 0 msec |
| összesen | 1430 msec | 920 msec | 830 msec |
| nyom-verem mérete | 4.1Mbyte | 2.0 Mbyte | 256 byte |

- `fibc` esetén a meghiúsulási idő azért nem 0, mert a rendszer a nyom-vermet (trail-stack) dolgozza fel. A nyom-verem tárolja a változó-értékadások visszacsinálási információit.

# Választás-mentesség diszjunktív feltételes szerkezetek esetén

- Feltételes szerkezet végrehajtásakor általában választási pont jön létre.
- A **SICStus Prolog** a „`( felt -> akkor ; egyébként )`” szerkezetet választásmentesen hajtja végre, ha a `felt` konjunkció tagjai csak:
  - aritmetikai összehasonlító eljáráshívások (pl. `<, =<, =:=`), és/vagy
  - kifejezés-típust ellenőrző eljáráshívások (pl. `atom, number`), és/vagy
  - általános összehasonlító eljáráshívások (ld. később, pl. `@<, @=<, ==`).
- Analóg módon választásmentes kód keletkezik a „`fej :- felt, !, akkor.`” klózból, ha `fej` argumentumai különböző változók, és `felt` olyan mint fent.
- Például választásmentes kód keletkezik az alábbi definíciókból:

```
vektorfajta(X, Y, Fajta) :-
    (   X =:= 0, Y =:= 0
        % X = 0, Y = 0  nem lenne jó
    ->  Fajta = null
    ;   Fajta = nem_null
    ).
```

```
vektorfajta(X, Y, Fajta) :-
    X =:= 0, Y =:= 0, !,
    Fajta = null.
vektorfajta(_X, _Y, nem_null).
```

# JOBBREKURZIÓ ÉS AKKUMULÁTOROK

# Jobbrekurzió (farok-rekurzió, tail-recursion) optimalizálás

- Az általános rekurzió költséges, helyben és időben is.
- Jobbrekurzióról beszélünk, ha
  - a rekurzív hívás a klóztörzs utolsó helyén van, vagy az utolsó helyen szereplő diszjunkció egyik tagjának utolsó helyén stb., és
  - a rekurzív hívás pillanatában nincs választási pont a predikátumban (a rekurzív hívást megelőző célok determinisztikusan futottak le, nem maradt nyitott diszjunkciós ág).
- Jobbrekurzió optimalizálás: az utolsó hívás végrehajtása **előtt** a predikátum által lefoglalt hely felszabadul ill. szemétgyűjtésre alkalmassá válik.
- Ez az optimalizálás nemcsak rekurzív hívás esetén, hanem minden **utolsó** hívás esetén megvalósul — utolsó hívás optimalizálás (last call optimisation).
- A jobbrekurzió így tehát nem növeli a memória-igényt, korlátlan mélységig futhat — mint a ciklusok az imperatív nyelvekben. Példa:

```
ciklus(Állapot) :- lépés(Állapot, Állapot1), !, ciklus(Állapot1).
ciklus(_Állapot).
```

# Predikátumok jobbrekurzív alakra hozása — listaösszeg

- A listaösszegzés „természetes”, nem jobbrekurzív definíciója:

```
% sum(+L, ?S): Az L számlista elemeinek összege S.
sum([], 0).
sum([X|L], S):- sum(L,S0), S is S0+X.
```

- Első jobbrekurzív változat, csak ellenőrzésre használható:

```
% sum1(+L, +S): Az L számlista elemeinek összege S.
sum1([], 0).
sum1([X|L], S) :- /* S is S0+X helyett: */ S0 is S-X, sum1(L, S0).
```

- Második jobbrekurzív változat, csak kiírni tudja az eredményt:

```
% sum2(+L): Az L számlista elemeinek összegét kiírja.
sum2(L):- sum2(L, 0).

% sum2(+L, +S0): Az L lista S0-lal növelt összegét kiírja.
sum2([], S) :-    write(S), nl.
sum2([X|L], S0):- S1 is S0+X, sum2(L, S1).
```

- Ahhoz, hogy az összeget **eredményként** ki tudjuk adni, szükséges egy további, kimenő argumentum.

# Jobbrekurzív listaösszeg — akkumulátorpár segítségével

- Harmadik változat: teljes értékű jobbrekurzív lista-összegző:

```
% sum3(+L, ?S): Az L számlista elemeinek összege S.
sum3(L, S):- sum3(L, 0, S).

% sum3(+L, +S0, ?S): Az L lista elemeit hozzáadva S0-hoz kapjuk S-et.
sum3([], S, S).
sum3([X|L], S0, S):-
        S1 is S0+X, sum3(L, S1, S).
```

- Az **akkumulátor** az imperatív (azaz megváltoztatható értékű) változó fogalmának deklaratív megfelelője:
  - A `sum3(L, S0, S)` predikátumban az `S0` és `S` argumentumok egy akkumulátorpárt alkotnak.
  - Az akkumulátorpár két része az adott változó mennyiség (a példában az összeg) különböző időpontokban vett értékeit mutatja:
    - `S0` az összeg értéke a `sum3/3` **meghívásakor**: az összegző változó kezdőértéke;
    - `S` az összeg értéke a `sum3/3` **lefutása után**: összegző változó végértéke.

# Az akkumulátorok használata

- Az akkumulátorokkal általánosan több egymás utáni változtatást is leírhatunk:

```
p(..., A0, A):-
        q0(..., A0, A1), ...,
        q1(..., A1, A2), ...,
        qn(..., An, A).
```

- A `sum3/3` második klóza ilyen alakra hozva:

```
sum3([X|L], S0, S):- plus(X, S0, S1), sum3(L, S1, S).

plus(X, S0, S) :- S is S0+X.
```

- Akkumulátorváltozók elnevezési konvenciója: kezdőérték: *`Vált`*`0`; közbülső értékek: *`Vált`*`1`, ..., *`Vált`*`n`; végérték: *`Vált`*.
- A Prolog akkumulátorpár nem más mint a funkcionális programozásból ismert gyűjtőargumentum és a függvény eredményének együttese.

# Akkumulátorok használata — folytatás

- Három lista összege

```
% sum_3_lists(+L, +LL, +LLL, +S0, ?S): Az L, LL, LLL számlisták
% összegeinek összege S-S0
sum_3_lists(L, LL, LLL, S0, S) :-
        sum3(L, S0, S1), sum3(LL, S1, S2), sum3(LLL, S2, S).
```

  Előrebocsátott megjegyzés: a fenti szabály DCG (Definite Clause Grammar) formája

```
sum_3_lists(L, LL, LLL) --> sum3(L), sum3(LL), sum3(LLL).
```

- Többszörös akkumulálás — listák összege és négyzetösszege

```
% sum12(+L, +S0, ?S, +Q0, ?Q): S-S0 =Σ Li, Q-Q0 = Σ Li*Li
sum12([], S, S, Q, Q).
sum12([X|L], S0, S, Q0, Q):-
        S1 is S0+X, Q1 is Q0+X*X, sum12(L, S1, S, Q1, Q).
```

- Többszörös akkumulátorok összevonása

```
% sum12(+L, +S0/Q0, ?S/Q): S-S0 =Σ Li, Q-Q0 = Σ Li*Li
sum12([], SQ, SQ).
sum12([X|L], S0/Q0, SQ):-
        S1 is S0+X, Q1 is Q0+X*X, sum12(L, S1/Q1, SQ).
```

# Korábbi listakezelő predikátumok

- A `revapp` mint akkumuláló eljárás

```
% revapp(Xs, L0, L): Xs megfordítását L0 elé füzve kapjuk L-t.
% Másképpen: Xs megfordítása L-L0.
revapp([], L, L).
revapp([X|Xs], L0, L) :-
        L1 = [X|L0], revapp(Xs, L1, L).
```

- Az `L-L0` jelölés (különbséglista): az a lista amelyet úgy kapunk, hogy `L` végéről elhagyjuk `L0`-t (előfeltétel: `L0` szuffixuma `L`-nek).
- Az `append` is tekinthető akkumuláló eljárásnak (a 2. és 3. arg. felcserélt).
  Az akkumulálás: az `L0` elejéről sorra elhagyjuk `Xs` elemeit, végül marad `L`.

```
% append(Xs, L, L0): L0 elejéről Xs elemeit lehagyva marad L.
% Másképpen: Xs = L0-L.
append([], L, L).
append([X|Xs], L, L0) :-
       L0 = [X|L1], append(Xs, L, L1).
```

# Egy mintafeladat: $a^n b^n$ alakú sorozat előállítása

- Első megoldás, $3n$ lépés

```
% anbn(N, L): Az L lista N db a-ból
% és azt követő N db b-ből áll.
anbn(N, L) :-
        an(N, a, AN),
        an(N, b, BN),
        append(AN, BN, L).

% an(N, A, L): L az A elemet N-szer
% tartalmazó lista
an(0, _A, L) :- !, L = [].
an(N, A, [A|L]) :-
        N > 0,
        N1 is N-1,
        an(N1, A, L).
```

- Második megoldás, $2n$ lépés

```
anbn(N, L) :-
        an(N, b, [], BN),
        an(N, a, BN, L).


% an(N, A, L0, L): L-L0 az A
% elemet N-szer tartalmazó lista
an(0, _A, L0, L) :- !, L = L0.
an(N, A, L0, [A|L]) :-
        N > 0,
        N1 is N-1,
        an(N1, A, L0, L).
```

# $a^n b^n$ alakú sorozatok (folyt.)

- Harmadik megoldás, $n$ lépés

```
anbn(N, L) :-
        anbn(N, [], L).

% anbn(N, L0, L): Az L-L0 lista N db a-ból és azt követő N db b-ből áll.
anbn(0, L0, L) :- !, L = L0.
anbn(N, L0, [a|L]) :-
        N > 0,
        N1 is N-1,
        anbn(N1, [b|L0], L).
```

- A második klóz nem jobbrekurzív változata

```
anbn(N, L0, L) :-
        N > 0, N1 is N-1,
        L1 = [b|L0],        % 1. lépés: L0 elé b => L1
        anbn(N1, L1, L2), % 2. lépés: L1 elé a^N1 b^N1 => L2
        L = [a|L2].         % 3. lépés: L2 elé a => L
```

# $a^n b^n$ alakú sorozatok — más nyelvű megoldások

- SML megoldás

```
local
  fun ab(0, L) = L
    | ab(N, L0) = #"a"::ab(N-1, #"b"::L0)
in fun anbn N = ab(N, [])
end
```

- C++ megoldás

```
link *anbn(unsigned n) {
  link *l = 0, *b = 0;     // ez elé építjük a b-ket
  link **a = &l;           // ebbe tesszük az a-kat
  for (; n > 0; --n) {
    *a = new link('a');    // elölről
    a = &(*a)->next;       // hátra épít
    b = new link('b', b); // hátulról előre épít
  }
  *a = b; return l;
}
```

# Összetettebb adatstruktúrák akkumulálása

- Az adatstruktúra:

```
% :- type bfa --> ures ; bfa(integer, bfa, bfa).
```

- A fa csomópontjaiban tároljuk a számértékeket, a levelek nem tárolnak információt.
- Egészek gyűjtése rendezett bináris fában
  - `beszur(BFa0, E, BFa)`: Az `E` egész számnak a `BFa0` fába való beszúrása a `BFa` bináris fát eredményezi.
  - Itt `BFa0` és `BFa` egy akkumulátor-pár, de az indexelés érdekében `BFa0` az első argumentum-pozícióba kerül.
- Példafutás:

```
| ?- beszur(ures, 3, Fa0),
     beszur(Fa0, 1, Fa1),
     beszur(Fa1, 5, Fa2).

Fa0 = bfa(3,ures,ures),
Fa1 = bfa(3,bfa(1,ures,ures),ures),
Fa2 = bfa(3,bfa(1,ures,ures),bfa(5,ures,ures)) ?
```

# Akkumulálás bináris fákkal

- Elem beszúrása bináris fába

```
% beszur(BF0, E, BF): E beszúrása BF0 rendezett fába
% a BF rendezett fát adja
% :- pred beszur(bfa::in, integer::in, bfa::out).
beszur(ures, Elem, bfa(Elem, ures, ures)).
beszur(BF0, Elem, BF):-
        BF0 = bfa(E,B,J),  % az indexelés működik!
        (   Elem =:= E -> BF = BF0
        ;   Elem < E ->
            BF = bfa(E,B1,J),
            beszur(B, Elem, B1)
        ;   BF = bfa(E,B,J1),
            beszur(J, Elem, J1)
        ).
```

# Akkumulálás bináris fákkal — folyt.

- Lista konverziója bináris fává

```
% lista_bfa(L, BF0, BF): L elemeit beszúrva BF0-ba kapjuk BF-t.
% :- pred lista_bfa(list(integer)::in, bfa::in, bfa::out).
lista_bfa([], BF, BF).
lista_bfa([E|L], BF0, BF):-
        beszur(BF0, E, BF1),
        lista_bfa(L, BF1, BF).

| ?- lista_bfa([3,1,5], ures, BF).
BF = bfa(3,bfa(1,ures,ures),bfa(5,ures,ures)) ? ;
no

| ?- lista_bfa([3,1,5,1,2,4], ures, BF).
BF = bfa(3,bfa(1,ures,bfa(2,ures,ures)),
           bfa(5,bfa(4,ures,ures),ures)) ? ;
no
```

# Akkumulálás bináris fákkal — folyt.

- Bináris fa konverziója listává

```
% bfa_lista(BF, L0, L): A BF fa levelei az L-L0 listát adják.
% :- pred bfa_lista(bfa::in, list(integer)::in,
%                               list(integer)::out).
bfa_lista(ures, L, L).
bfa_lista(bfa(E, B, J), L0, L) :-
        bfa_lista(J, L0, L1),
        bfa_lista(B, [E|L1], L).
```

- Rendezés bináris fával

```
% L lista rendezettje R.
% :- pred rendez(list(integer)::in, list(integer)::out).
rendez(L, R):-
        lista_bfa(L, ures, BF), bfa_lista(BF, [], R).

| ?- rendez([1,5,3,1,2,4], R).
R = [1,2,3,4,5] ? ;
no
```

# IMPERATÍV PROGRAMOK ÁTÍRÁSA PROLOGBA

# Hogyan írjunk át imperatív nyelvű algoritmust Prolog programmá?

- Példafeladat: Hatékony hatványozási algoritmus
  - Alaplépés: a kitevő felezése, az alap négyzetre emelése.
  - Lényegében a kitevő kettes számrendszerbeli alakja szerint hatványoz.
- Az algoritmust megvalósító C nyelvű függvény:

```
/* hatv(a, h) = a**h */
int hatv(int a, unsigned h)
{
  int e = 1;
  while (h > 0)
  {
    if (h & 1) e *= a;
    h >>= 1; a *= a;
  }
  return e;
}
```

- Az algoritmusban három változó van: `a, h, e`:
  - `a` és `h` végértékére nincs szükség,
  - `e` végső értéke szükséges (ez a függvény eredménye).

# A `hatv` C függvénynek megfelelő Prolog eljárás

- A fügvény eredménye a reláció utolsó arg.-a: `hatv(+A, +H, ?E)`: $\mathtt{A}^{\mathtt{H}} = \mathtt{E}$.
- A ciklusnak segédeljárás felel meg: `hatv(+A0, +H0, +E0, ?E)`: $\mathtt{A0}^{\mathtt{H0}} * \mathtt{E0} = \mathtt{E}$.
- Az »a« és »h« C változóknak az »+A« és »+H« bemenő *paraméterek* (nem kell a végérték), az »e« C változónak az »+E0, ?E« *akkumulátor-pár* felel meg (kezdőérték, végérték).

```
hatv(A, H, E) :-
        hatv(A, H, 1, E).

hatv(A0, H0, E0, E) :- H0 > 0, !,
        (   H0 /\ 1 =:= 1
              % /\ ≡ bitenkénti ``és''
        ->  E1 is E0*A0
        ;   E1 = E0
        ),
        H1 is H0 >> 1,
        A1 is A0*A0,
        hatv(A1, H1, E1, E).
hatv(_, _, E, E).
```

```
int hatv(int a, unsigned h)
{       int e = 1;

  ism:  if (h > 0)
        {  if (h & 1)
             e *= a;



           h >>= 1;
           a *= a;
           goto ism;
        } else return e;
}
```

# A C ciklus és a Prolog eljárás kapcsolata

- A ciklust megvalósító Prolog eljárás minden pontján minden C változónak megfeleltetethető egy Prolog változó (pl. `h`-nak `H0, H1, ...`):
  - A ciklusmag elején a C változók a megfelelő Prolog argumentumban levő változónak felelnek meg.
  - Egy C értékadásnak egy új Prolog változó bevezetése felel meg, az ez után következő kódban az új változó felel meg a C változónak.
  - Ha a diszjunkció egyik ága megváltoztat egy változót, akkor a többi ágon is be kell vezetni az új Prolog változót, a régivel azonos értékkel (ld. `if (h & 1) ...`).
- A C ciklusmag végén a Prolog eljárást vissza kell hívni, argumentumaiban az egyes C változóknak pillanatnyilag megfeleltetett Prolog változóval.
- A C ciklus **ciklus-invariánsa** nem más mint a Prolog eljárás fejkommentje, a példában:

  `% hatv(+A0, +H0, +E0, ?E)`: $\mathtt{A0}^{\mathtt{H0}} * \mathtt{E0} = \mathtt{E}$.

# Programhelyesség-bizonyítás

- Egy algoritmus (függvény) specifikációja:
  - **előfeltételek**: a bemenő paramétereknek teljesíteniük kell ezeket,
  - **utófeltételek**: a paraméterek és az eredmény kapcsolatát írják le.
- Egy algoritmus **helyes**, ha minden, az előfeltételeket kielégítő adatra a függvény hibátlanul lefut, és eredményére fennállnak az utófeltételek.
- Példa: `x = mfoku_gyok(a,b,c)`
  - előfeltételek: `b*b-4*a*c >= 0, a` $\neq$ `0`
  - utófeltétel: `a*x*x+b*x+c = 0`
  - a program:

```
double mfoku_gyok(a, b, c)
double a, b, c;
{   double d = sqrt(b*b-4*a*c);
    return (-b+d)/2/a;
}
```

- A program helyességének bizonyítása lineáris kódra viszonylag egyszerű.

# Ciklikus programok helyességének bizonyítása

- A ciklusokat „fel kell vágni” egy **ciklus-invariáns**sal, amely:
  - az előfeltételekből és a ciklust megelőző értékadásokból következik,
  - ha a ciklus elején fennáll, akkor a ciklus végén is (indukció),
  - belőle és a leállási feltételből következik a ciklus utófeltétele.

```
int hatv(int a0, unsigned h0) /* utófeltétel: hatv(a0, h0) = a0^h0 */
{ int e = 1, a = a0, h = h0;
  while (h > 0)
  { /* ciklus-invariáns: a0^h0 == e*a^h */
    /* induláskor a kezdőértékek alapján triviálisan fennáll */
    if (h & 1) e *= a;          /* e' = e * a^(h&1) */
    h >>= 1;                    /* h' = (h-(h&1))/2 */
    a *= a;                     /* a' = a*a */
  }                             /* indukció: e'*a'^h' = ... = e*a^h */
  return e;
  /* Az invariánsból h = 0 miatt következik az utófeltétel */
}
```

# Második példa: Fibonacci sorozat tagjainak hatékony számítása

- A C függvény

```
unsigned fib(unsigned n)
{ unsigned f = 0, fnxt = 1, t;
  while (n > 0) t = fnxt, fnxt += f, f = t, --n;  /* (1) */
  return f;
}
```

- Az (1) ciklusnak bemenő változói: `n, f, fnxt`, kimenő változója: `f`.
- A ciklusnak megfeleltetett Prolog eljárás: `fib(N, F0, FNXT, F)`:
  az `F0` és `FNXT` kezdőértékű Fibonacci sorozat `N`-edik tagja `F`.

```
% "betű szerinti" Prolog átírás:
fib(N, F0, FNXT, F) :- N > 0, !,
    T = FNXT, FNXT1 is FNXT+F0,
    F1 = T, N1 is N-1,
    fib(N1, F1, FNXT1, F).
fib(_, F0, _, F0).
```

```
% Leegyszerűsített alak:
fib(N, F0, FNXT, F) :- N > 0, !,
    FNXT1 is FNXT+F0,
    N1 is N-1,
    fib(N1, FNXT, FNXT1, F).
fib(_, F0, _, F0).
```

# Fibonacci sorozat — Prolog stílusban

- A Fibonacci sorozat teljes Prolog megvalósítása, és az ennek megfeleltethető C kód:

```
fib(N, F) :-
        fib(N, 0, 1, F).

fib(N, F0, F1, F) :-
        N > 0, !,
        N1 is N-1,
        F2 is F0+F1,

        fib(N1, F1, F2, F).

fib(_, F0, _, F0).
```

```
% unsigned fib(unsigned N)
% { unsigned F0=0, F1=1, F2;
%
% ism:
%   if (N > 0)
%   {  --N;
%      F2 = F0+F1;
%      F0 = F1; F1 = F2;
%      goto ism;
%   }
%   return F0;
% }
```

# MEGOLDÁSOK GYŰJTÉSE ÉS FELSOROLÁSA

# Keresési feladat Prologban — felsorolás vagy gyűjtés?

- Keresési feladat: bizonyos feltételeknek megfelelő dolgok meghatározása.
- Prolog nyelven egy ilyen feladat alapvetően kétféle módon oldható meg:
  - gyűjtés — az összes megoldás összegyűjtése, pl. egy listába;
  - felsorolás — a megoldások visszalépéses felsorolása: egyszerre egy megoldást kapunk, de visszalépés esetén sorra előáll minden megoldás.
- Egyszerű példa: egy lista páros elemeinek megkeresése:

```
% Gyűjtés:
% páros_elemei(L, Pk): Pk az L
% lista páros elemeinek listája.
páros_elemei([], []).
páros_elemei([X|L], Pk) :-
    X mod 2 =\= 0, !,
    páros_elemei(L, Pk).
páros_elemei([P|L], [P|Pk]) :-
    páros_elemei(L, Pk).
```

```
% Felsorolás:
% páros_eleme(L, P): P egy páros
% eleme az L listának.
páros_eleme([X|L], P) :-
    X mod 2 =:= 0, P = X.
páros_eleme([_X|L], P) :-
    % _X akár páros, akár páratlan
    % folytatjuk a felsorolást:
    páros_eleme(L, P).

% egyszerűbb megoldás:
páros_eleme2(L, P) :-
    member(P, L), P mod 2 =:= 0.
```

# Mi a közös a felsoroló és gyűjtő megoldásokban?

- Keressük meg a közös részt a `páros_elemei` és `páros_eleme` eljárásokban!
- Mindkettőben át kell lépni a páratlan elemeket, és meg kell keresni az első páros elemet a listában:

```
% köv_páros(L0, P, L) :- Az L0 első páros eleme P, a maradék L.
köv_páros([X|L0], P, L) :-
    X mod 2 =\= 0, !, köv_páros(L0, P, L).
köv_páros([P|L], P, L).
```

- A `köv_páros` eljárásra épülő gyűjtő és felsoroló eljárások:

```
% páros_elemei(L, Pk): Pk az L
% lista páros elemeinek listája.
páros_elemei(L0, Pk) :-
    köv_páros(L0, P, L1), !,
    Pk = [P|Pk1],
    páros_elemei(L1, Pk1).
páros_elemei(_, []).
```

```
% páros_eleme(L, P): P egy páros
% eleme az L listának.
páros_eleme(L0, P) :-
    köv_páros(L0, P0, L1),
    (   P = P0
    ;   páros_eleme(L1, P)
    ).
```

# A gyűjtő és felsoroló sémák összehasonlítása

- A páros elemeket gyűjtő ill. felsoroló eljárások alapján adjunk meg egy általános sémát a kétféle eljárástípusra!
- Az általános esetben a keresésnek lehet egy vagy több `Param` paramétere. Például, kereshetjük a `Param`-mal osztható elemeket.
- A közös építőelem: `következő(V0, Param, E, V1)`: A `V0` kifejezéssel jellemzett keresési térben az első megoldás `E`, és a fennmaradó keresési tér `V1`, a `Param` paraméter-érték mellett.

A gyűjtő séma:

```
% A V0 keresési térben a Param
% paraméterű megoldások listája L.
megoldások(V0, Param, L) :-
  következő(V0, Param, E, V1), !,
  L = [E|L1],
  megoldások(V1, Param, L1).
megoldások(_, _, []).
```

A felsoroló séma:

```
% A V0 keresési térben E egy
% Param paraméterű megoldás.
megoldás(V0, Param, E) :-
  következő(V0, Param, E0, V1),
  (   E = E0
  ;   megoldás(V1, Param, E)
  ).
```

# Egy összetettebb példa: fennsíkok felsorolása

- Egy listában fennsíknak nevezünk:
  - egy csupa azonos elemből álló, legalább kételemű, folytonos részlistát;
  - amely az ilyenek között maximális (egyik irányba sem kiterjeszthető).
- A feladat: felsorolandók egy lista fennsíkjai és kezdőpozíciójuk.
- `fennsík(L, F, H)`: Az `L` listában az `F` (1-től számozott) pozíción egy `H` hosszú fennsík van.
- Egy gyorsprogramozási módszerrel készült (Prolog hekker) megoldás:

```
fennsík0(L, F, H) :-
    Teste = [E,E|_],
    append(Eleje, Teste, L),
    \+ last(Eleje, E),
    length(Eleje, F0), F is F0+1,
    kezdethossz(Teste, H).
    % kezdethossz/2 definícióját
    % lásd korábban
```

```
fennsík1(L, F, H) :-
    Teste = [E,E|_],
    append(Eleje, Teste, L),
    \+ last(Eleje, E),
    length(Eleje, F0), F is F0+1,
    % kezdethossz/2 kifejtve:
    (   append(Ek, Farok, Teste),
        \+ Farok = [E|_] ->
        length(Ek, H)
    ).
```

# Fennsíkok felsorolása — 2., hatékony megoldás

- Használjuk a megoldás-felsoroló sémát: `megoldás(`*`V0`*`, `*`Param`*`, `*`E`*`)`!
  - *V0*: »`L, P`«, a bejárandó lista és első elemének pozíciója;
  - *Param*: üres;
  - *E*: »`F, H`«, a megoldás-fennsík kezdőpozíciója és hossza.

```
% Az L listában az F pozíción egy H hosszú fennsík van.
fennsík(L, F, H) :-
        fennsík(L, 1, F, H).

% A P0-tól számozott L0 listában az F pozíción
% egy H hosszú fennsík van.
fennsík(L0, P0, F, H) :-
        % az első fennsík jellemzői F0 és H0,
        % a fennsík utáni maradéklista L1:
        első_fennsík(L0, P0, F0, H0, L1),
        (   F = F0, H = H0
        ;   P1 is F0+H0,    % L1 kezdőpozíciója, P1, nem más mint
                            % az előző megoldás kezdőpozíciója+hossza
            fennsík(L1, P1, F, H)
        ).
```

# Fennsíkok felsorolása — 2., hatékony megoldás (folyt.)

- Az első fennsík előállítása:

```
% első_fennsík(+L0, +P0, -F, -H, -L): A P0-tól számozott L0 listában az
% első fennsík az F. pozíción van és hossza H, a fennsík után fennmaradó
% rész pedig az L lista.
első_fennsík([E,E|L1], P0, F, H, L) :-
        !, F = P0, azonosak(L1, E, 2, H, L).
első_fennsík([_|L1], P0, F, H, L) :-
        P1 is P0+1,
        első_fennsík(L1, P1, F, H, L).

% azonosak(+L0, +E, +H0, -H, -L): Az L0 lista elejéről a maximális számú
% E-vel azonos elemet lehagyva marad L, a lehagyott elemek száma H-H0.
azonosak([X|L0], E, H0, H, L) :-
        E = X, !,
        H1 is H0+1,
        azonosak(L0, E, H1, H, L).
azonosak(L, _, H, H, L).
```

# MEGOLDÁSGYŰJTŐ BEÉPÍTETT ELJÁRÁSOK

# Gyűjtés és felsorolás kapcsolata

- Korábban láttuk, hogyan lehet egy keresési feladat gyűjtő és felsoroló eljárásait egy közös magból előállítani.
- Most vizsgáljuk meg, hogyan lehet egy felsoroló eljárást visszavezetni a gyűjtőre, és fordítva:
  - felsorolás gyűjtésből: a `member/2` könyvtári eljárás segítségével, pl.

    ```
    páros_eleme(L, P) :-
        páros_elemei(L, Pk), member(P, Pk).
    ```

    Természetesen ez így nem hatékony!
  - gyűjtés felsorolásból: a megoldásgyűjtő beépített eljárások segítségével, pl.

    ```
    páros_elemei(L, Pk) :-
        findall(P, páros_eleme(L, P), Pk).
        % A páros_eleme(L, P) cél
        % összes P megoldásának listája Pk.
    ```

# A `findall(?Gyűjtő, :Cél, ?Lista)` beépített eljárás

- Az eljárás végrehajtása (procedurális szemantikája):
  - a `Cél` kifejezést eljáráshívásként értelmezi, meghívja
    (A `:` annotáció meta- (azaz eljárás) argumentumot jelez);
  - minden egyes megoldásához előállítja `Gyűjtő` egy *másolatát*, azaz a megoldásbeli változókat, ha vannak, szisztematikusan újakkal helyettesíti;
  - Az összes `Gyűjtő` értéket egy listába összegyűjti, és ezt egyesíti `Lista`-val.
- Példák az eljárás használatára:

```
| ?- findall(X, (member(X, [1,7,8,3,2,4]), X>3), L).
                ⟹  L = [7,8,4] ? ; no
| ?- findall(X-Y, (between(1, 3, X), between(1, X, Y)), L).
                ⟹  L = [1-1,2-1,2-2,3-1,3-2,3-3] ? ; no
```

- Az eljárás jelentése (deklaratív szemantikája):
  `Lista` = { `Gyűjtő` másolat | (∃`X` . . . `Z`)`Cél` igaz }
  ahol `X`, . . . , `Z` a `findall` hívásban levő szabad változók (azaz olyan, a hívás pillanatában behelyettesítetlen változók, amelyek a `Cél`-ban előfordulnak de a `Gyűjtő`-ben nem).

# A `bagof(?Gyűjtő, :Cél, ?Lista)` beépített eljárás

- Az eljárás végrehajtása (procedurális szemantikája):
  - a `Cél` kifejezést eljáráshívásként értelmezi, meghívja;
  - összegyűjti a megoldásait (a `Gyűjtő`-t és a szabad változók behelyettesítéseit);
  - a szabad változók összes behelyettesítését *felsorolja* és mindegyikhez a `Lista`-ban megadja az összes hozzá tartozó `Gyűjtő` értéket.
- Példák az eljárás használatára:

```
gráf([a-b,a-c,b-c,c-d,b-d]).

| ?- gráf(_G), findall(B, member(A-B, _G), VegP).
                              ⟹ VegP = [b,c,c,d,d] ? ; no
| ?- gráf(_G), bagof(B, member(A-B, _G), VegP).
                              ⟹ A = a, VegP = [b,c] ? ;
                                 A = b, VegP = [c,d] ? ;
                                 A = c, VegP = [d] ? ; no
```

- A `bagof` eljárás jelentése (deklaratív szemantikája):
  `Lista` $= \{$ `Gyűjtő` $\mid$ `Cél` igaz $\}$, `Lista` $\neq$ `[]`.

# A `bagof` megoldásgyűjtő eljárás (folyt.)

- Explicit kvantorok
  - `bagof(Gyűjtő, V1 ^ ...^ Vn ^ Cél, Lista)` alakú hívása a `V1, ..., Vn` változókat egzisztenciálisan kötöttnek tekinti, nem sorolja fel.
  - jelentése: `Lista` $=$ { `Gyűjtő` $\mid$ $(\exists$`V1`$,\ldots,$`Vn`$)$`Cél` igaz } $\neq$ `[]`.

```
| ?- gráf(_G), bagof(B, A^member(A-B, _G), VegP).
                                  ⟹ VegP = [b,c,c,d,d] ? ; no
```

- Egymásba ágyazott gyűjtések
  - szabad változók esetén a `bagof` nemdeterminisztikus lehet, így skatulyázható:

```
% A G irányított gráf fokszámlistája FL:
% FL = { A-N | N = |{ V | A-V ∈ G }| }
fokszámai(G, FL) :-
    bagof(A-N, Vk^(bagof(V, member(A-V, G), Vk),
                   length(Vk, N)                 ), FL).

| ?- gráf(_G), fokszámai(_G, FL).
               ⟹ FL = [a-2,b-2,c-1] ? ; no
```

# A `bagof` megoldásgyűjtő eljárás (folyt.)

- Fokszámlista hatékonyabb előállítása
  - a vezérlési szerkezeteket célszerű elkerülni a meta-argumentumokban
  - segédeljárás bevezetésével a kvantor is szükségtelenné válik:

```
% Az A pont foka a G irányított gráfban N, N>0.
pont_foka(A, G, N) :-
        bagof(V, member(A-V, G), Vk), length(Vk, N).

% A G irányított gráf fokszámlistája FL:
fokszámai(G, FL) :-    bagof(A-N, pont_foka(A, G, N), FL).
```

- Példák a `bagof/3` és `findall/3` közötti kisebb különbségekre:

```
| ?- findall(X, (between(1, 5, X), X<0), L).   ⟹ L = [] ? ; no
| ?-   bagof(X, (between(1, 5, X), X<0), L).   ⟹ no
| ?- findall(S, member(S, [f(X,X),g(X,Y)]), L).
                                  ⟹ L = [f(_A,_A),g(_B,_C)] ? ; no
| ?-   bagof(S, member(S, [f(X,X),g(X,Y)]), L).
                                  ⟹ L = [f(X,X),g(X,Y)] ? ; no
```

- A `bagof/3` logikailag tisztább mint a `findall/3`, de időigényesebb!

# A `setof(?Gyűjtő, :Cél, ?Lista)` beépített eljárás

- az eljárás végrehajtása:
  - ugyanaz mint: `bagof(Gyűjtő, Cél, L0), sort(L0, Lista),`
  - itt `sort/2` egy univerzális rendező eljárás (lásd később), amely
  - az eredménylistát rendezi (az ismétlődések kiszűrésével).
- Példa a `setof/3` eljárás használatára:

```
gráf([a-b,a-c,b-c,c-d,b-d]).

% Gráf egy pontja P.
pontja(P, Gráf) :- member(A-B, Gráf), ( P = A ; P = B).

% A G gráf pontjainak listája Pk.
gráf_pontjai(G, Pk) :- setof(P, pontja(P, G), Pk).

| ?- gráf(_G), gráf_pontjai(_G, Pk). ⟹ Pk = [a,b,c,d] ? ; no
```

# META-LOGIKAI ELJÁRÁSOK

# A meta-logikai, azaz a logikán túlmutató eljárások fajtái:

- A Prolog kifejezések pillanatnyi behelyettesítettségi állapotát vizsgáló eljárások (értelemszerűen sorrendfüggőek):
  - kifejezések osztályozása (1)

```
| ?- var(X) /* X változó? */, X = 1. ⟹ X = 1
| ?- X = 1, var(X). ⟹ no
```

  - kifejezések rendezése (4)

```
| ?- X @< 3 /* X megelőzi 3-t? */, X = 4. ⟹ X = 4
   % a változók megelőzik a nem változó kifejezéseket
| ?- X = 4, X @< 3. ⟹ no
```

- Prolog kifejezéseket szétszedő vagy összerakó eljárások:
  - (struktúra) kifejezés $\Longleftrightarrow$ név és argumentumok (2)

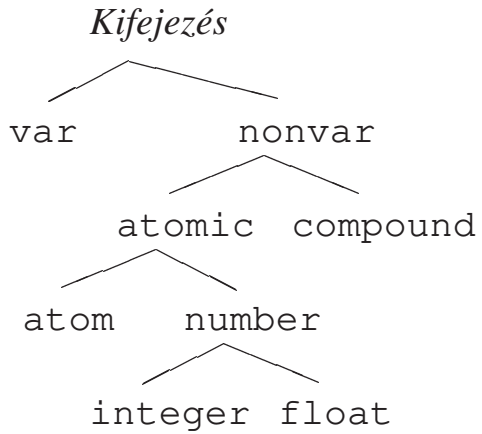
```
| ?- X = f(alma,körte), X =.. L ⟹ L = [f,alma,körte]
```

  - atomok és számok $\Longleftrightarrow$ karaktereik (3)

```
| ?- atom_codes(A, [0'a,0'b,0'a]) ⟹ A = aba
```

# Kifejezések osztályozása

- Kifejezés-osztályok fastruktúrája — osztályozó beépített eljárások (ismétlés)




| | |
|---|---|
| `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 |

- SICStus-specifikus osztályozó eljárások:
  - `simple(X)` : `X` nem összetett (konstans vagy változó);
  - `ground(X)` : `X` tömör, azaz nem tartalmaz behelyettesítetlen változót.
- Az osztályozó eljárások használata — példák
  - `var`, `nonvar` — többirányú eljárásokban a különböző irányok elágaztatása
  - `number`, `atom`, `...` — nem-megkülönböztetett úniók feldolgozása

# Osztályozó eljárások: elágaztatás behelyettesítettség alapján

- Példa: a `length/2` beépített eljárás megvalósítása (SICStus kód!)

```
% length(?L, ?N): Az L lista N hosszú.
length(L, N) :- var(N), !, length(L, 0, N).
length(L, N) :-            dlength(L, 0, N).

% length(?L, +I0, -I):          | % dlength(?L, +I0, +I):
%   Az L lista I-I0 hosszú.     | %   Az L lista I-I0 hosszú.
length([], I, I).               | dlength([], I, I) :- !.
length([_|L], I0, I) :-         | dlength([_|L], I0, I) :-
        I1 is I0+1,             |         I0<I, I1 is I0+1,
        length(L, I1, I).       |         dlength(L, I1, I).
```

```
| ?- length([1,2], Len).   (length/3)   ⟹  Len = 2 ? ; no
| ?- length([1,2], 3).     (dlength/3)   ⟹  no
| ?- length(L, 3).         (dlength/3)   ⟹  L = [_A,_B,_C] ?;no
| ?- length(L, Len).       (length/3)    ⟹  L = [], Len = 0 ? ;
                    L = [_A], Len = 1 ? ; L = [_A,_B], Len = 2 ?
```

# Osztályozó eljárások: nem-megkülönböztetett úniók kezelése

- Példa: egy formula természetes módon ábrázolható Prologban, pl. `x*y+y+1`. A formula típusának leírásához nem-megkülönböztetett úniót kell használni: n

```
% :- type form == atom \/ int \/ { form+form } \/ { form-form } ...
```

- Példa formulakezelésre: szimbolikus derivált előállítása

```
% deriv(+Kif, +X, ?D):  Kif-nek az X atom szerinti deriváltja D.
deriv(X, X, D) :-                       !, D = 1.
deriv(C, _X, D) :-                      atomic(C), !, D = 0.
deriv(U+V, X, DU+DV) :-                 deriv(U, X, DU), deriv(V, X, DV).
deriv(U-V, X, DU-DV) :-                 deriv(U, X, DU), deriv(V, X, DV).
deriv(U*V, X, DU*V + U*DV) :-           deriv(U, X, DU), deriv(V, X, DV).
deriv(U/V, X, (DU*V - U*DV)/(V*V)) :- deriv(U, X, DU), deriv(V, X, DV).

| ?- deriv(x*y+1, x, DX), deriv(x*y+1, y, DY).
                        ⟹ DX = 1*y+x*0+0, DY = 0*y+x*1+0 ? ; no
| ?- deriv((x+y)*(2+x), x, D).
                        ⟹ D = (1+0)*(2+x)+(x+y)*(0+1) ? ; no
```

# Struktúrák szétszedése és összerakása: az *univ* eljárás

- Az *univ* eljárás hívási mintái:
  - `+Kif =..  ?Lista`
  - `-Kif =..  +Lista`
- Az eljárás jelentése: Igaz, ha
  - `Kif =` *Fun*`(`$A_1$`, ..., `$A_n$`)` és `Lista = [`*Fun*`,`$A_1$`,... `$A_n$`]`, ahol *Fun* egy atom és $A_1, \ldots A_n$ tetszőleges kifejezések; vagy
  - `Kif =` *C* és `Lista = [`*C*`]`, ahol *C* egy konstans.
- Példák

```
| ?- el(a,b,10) =.. L.        ⟹  L = [el,a,b,10]
| ?- Kif =.. [el,a,b,10].     ⟹  Kif = el(a,b,10)
| ?- alma =.. L.               ⟹  L = [alma]
| ?- Kif =.. [1234].          ⟹  Kif = 1234
| ?- Kif =.. L.                ⟹  hiba
| ?- f(a,g(10,20)) =.. L.     ⟹  L = [f,a,g(10,20)]
| ?- Kif =.. [/,X,2+X].       ⟹  Kif = X/(2+X)
| ?- [a,b,c] =.. L.           ⟹  L = ['.',a,[b,c]]
```

# Struktúrák szétszedése és összerakása: a `functor` eljárás

- `functor/3`: kifejezés funktorának, adott funktorú kifejezésnek az előállítása
  - Hívási minták: `functor(-Kif, +Név, +Argszám)`
    `functor(+Kif, ?Név, ?Argszám)`
  - Jelentése: igaz, ha `Kif` egy `Név/Argszám` funktorú kifejezés.
    - A konstansok 0-argumentumú kifejezésnek számítanak.
    - Ha `Kif` kimenő, az adott funktorú legáltalánosabb kifejezéssel egyesíti (argumentumaiban csupa különböző változóval).
- Példák:

```
| ?- functor(el(a,b,1), F, N).     ⟹  F = el, N = 3
| ?- functor(E, el, 3).            ⟹  E = el(_A,_B,_C)
| ?- functor(alma, F, N).          ⟹  F = alma, N = 0
| ?- functor(Kif, 122, 0).         ⟹  Kif = 122
| ?- functor(Kif, el, N).          ⟹  hiba
| ?- functor(Kif, 122, 1).         ⟹  hiba
| ?- functor([1,2,3], F, N).      ⟹  F = '.', N = 2
| ?- functor(Kif, ., 2).           ⟹  Kif = [_A|_B]
```

# Struktúrák szétszedése és összerakása: az `arg` eljárás

- `arg/3`: kifejezés adott sorszámú argumentuma.
  - Hívási minta: `arg(+Sorszám, +StrKif, ?Arg)`
  - Jelentése: A `StrKif` struktúra `Sorszám`-adik argumentuma `Arg`.
  - Végrehajtása: `Arg`-ot az adott sorszámú argumentummal **egyesíti**.
  - Az `arg/3` eljárás így nem csak egy argumentum elővételére, hanem a struktúra változó-argumentumának behelyettesítésére is használható (ld. a 2. példát alább).
- Példák:

```
| ?- arg(3, el(a, b, 23), Arg).    ⟹  Arg = 23
| ?- K=el(_,_,_), arg(1, K, a),
     arg(2, K, b), arg(3, K, 23). ⟹  K = el(a,b,23)
| ?- arg(1, [1,2,3], A).          ⟹  A = 1
| ?- arg(2, [1,2,3], B).          ⟹  B = [2,3]
```

- Az *univ* visszavezethető a `functor` és `arg` eljárásokra (és viszont), például:

```
Kif =.. [F,A1,A2]   ⟺   functor(Kif, F, 2),
                        arg(1, Kif, A1), arg(2, Kif, A2)
```

# Az *univ* alkalmazása: ismétlődő sémák összevonása

- A feladat: egy szimbolikus aritmetikai kifejezésben a kiértékelhető (infix) részkifejezések helyettesítése az értékükkel.
- 1. megoldás, *univ* nélkül:

```
% Az X szimbolikus kifejezés egyszerűsítése EX.
egysz0(X, EX) :-
    atomic(X), !, EX = X.
egysz0(U+V, EKif) :-
    egysz0(U, EU), egysz0(V, EV),
    kiszamol(EU+EV, EU, EV, EKif).
egysz0(U*V, EKif) :-
    egysz0(U, EU), egysz0(V, EV),
    kiszamol(EU*EV, EU, EV, EKif).
%...
% EU és EV részekből képzett EUV egyszerűsítése EKif.
kiszamol(EUV, EU, EV, EKif) :-
    number(EU), number(EV), !, EKif is EUV.
kiszamol(EUV, _, _, EUV).
```

```
| ?- deriv((x+y)*(2+x), x, D), egysz0(D, ED).
     ⟹ D = (1+0)*(2+x)+(x+y)*(0+1), ED = 1*(2+x)+(x+y)*1 ? ; no
```

# Az *univ* alkalmazása: ismétlődő sémák összevonása (folyt.)

- Kifejezés-egyszerűsítés, 2. megoldás, *univ* segítségével

```
egysz(X, EX) :-
    atomic(X), !, EX = X.
egysz(Kif, EKif) :-
    Kif =.. [Muv,U,V],     % Kif = Muv(U,V)
    egysz(U, EU), egysz(V, EV),
    EUV =.. [Muv,EU,EV],   % EUV = Muv(EU,EV)
    kiszamol(EUV, EU, EV, EKif).
```

- Kifejezés-egyszerűsítés, általánosítás tetszőleges *tömör* kifejezésre:

```
egysz1(Kif, EKif) :-
    Kif =.. [M|ArgL], egysz1_lista(ArgL, EArgL), EKif0 =.. [M|EArgL],
    % catch(:Cél,?Kiv,:KCél): ha Cél kivételt dob, KCél-t futtatja:
    catch(EKif is EKif0, _, EKif = EKif0).

egysz1_lista([], []).
egysz1_lista([K|Kk], [E|Ek]) :-
    egysz1(K, E), egysz1_lista(Kk, Ek).
```

```
| ?- egysz1(f(1+2+a, exp(3,2), a+1+2), E). ⟹ E = f(3+a,9.0,a+1+2)
```

# *Univ* alkalmazása általános kifejezés-bejárásra: kiiratás

- A feladat: egy tetszőleges kifejezés kiiratása úgy, hogy
  - a kétargumentumú operátorok zárójelezett infix formában,
  - minden más alap-struktúra alakban jelenjék meg.

```
ki(Kif) :-
    compound(Kif), !, Kif =.. [Func, A1|ArgL],
    (   % kétargumentumú kifejezés, funktora infix operátor
        ArgL = [A2], current_op(_, Kind, Func), infix_fajta(Kind),
    ->  write('('), ki(A1),
        write(' '), write(Fun), write(' '), ki(A2), write(')')
    ;   write(Func),
        write('('), ki(A1), arglistaki(ArgL), write(')')
    ).
ki(Kif) :- write(Kif).

% infix_fajta(F): F egy infix operátorfajta.
infix_fajta(xfx). infix_fajta(xfy). infix_fajta(yfx).

% Az [A1,...,An] listát ",A1,...,An" alakban kiírja.
arglistaki([]).
arglistaki([A|AL]) :- write(','), ki(A), arglistaki(AL).
```

```
| ?- ki(f(+a, X*c*X, e)).  ⟹  f(+(a),((_117 * c) * _117),e)
```

# *Univ* alkalmazása általános kifejezés-bejárásra: változómentesítés

- A SICStus Prologban beépített `numbervars(?Kif, +N0, ?N)` eljárás hatása:
  - A tetszőleges `Kif` minden változóját `'$VAR'(I)` alakú kifejezéssel helyettesíti, `I = N0, ..., N-1` (azaz `Kif`-ben `N-N0` különböző változó van).
- A `'$VAR'(0), '$VAR'(1), ...` kifejezések `write`-tal való kiíráskor változónévként (`A, B...`) jelennek meg.
- A `write_term(Kif, Opciók)` beépített eljárás kiírja a `Kif` kifejezést, az `Opciók` által meghatározott módon.
- A `numbervars/3` által létrehozott `'$VAR'/1` struktúrák „eredetiben” is megjeleníthetők:

```
| ?- _K = [f(_X),g(_),_X], numbervars(_K, 0, N), write(_K), nl,
          write_term(_K, [quoted(true),numbervars(false)]), nl.
===>     [f(A),g(B),A]
         [f('$VAR'(0)),g('$VAR'(1)),'$VAR'(0)]
         N = 2
```

- A feladat: elkészítendő egy `numbervars1/3` eljárás, amely `'$VAR'` helyett `'$myvar'` funktort használ.

# Általános kifejezés-bejárás *univ*-val: változómentesítés

- A változómentesítés egy saját megvalósítása:

```
% A Term kifejezésben levő változókat '$myvar(I)' stb.
% struktúrákkal helyettesíti be, I = N0, ... N-1.
numbervars1(Term, N0, N) :-
        var(Term), !,
        Term = '$myvar'(N0), N is N0+1.
numbervars1(Term, N0, N) :-
        Term =.. [_|Args],
        numbervars1_list(Args, N0, N).

% numbervars1_list(L, N0, N): Az L listában levő változókat
% '$myvar(I)' stb. struktúrákkal helyettesíti be, I = N0, ... N-1.
numbervars1_list([], N, N).
numbervars1_list([A|As], N0, N) :-
        numbervars1(A, N0, N1), numbervars1_list(As, N1, N).

| ?- Kif = [f(_X),g(_),_X], numbervars1(Kif, 0, N).
====>       N = 2,
            Kif = [f('$myvar'(0)),g('$myvar'(1)),'$myvar'(0)]
```

# `numbervars1` egy alkalmazása

## Két kifejezés azonossága

- A kifejezések azonosak, ha változó-behelyettesítés *nélkül* egyesíthetőek;
- azaz, ha az egyik változót tartalmaz, akkor a másik ugyanott ugyanazt a változót tartalmazza.
- `azonos/2` == néven, `nem_azonos/2` `\==` néven szabványos beépített eljárás és operátor.

```
nem_azonos(X, Y) :-
    (   numbervars1(X, 0, N), numbervars1(Y, N, _), X = Y -> fail
    ;   true
    ).

azonos(X, Y) :-
    \+ nem_azonos(X, Y).

% azonos2/2 és azonos/2 teljesen ekvivalens.
% \+ \+ X : csakkor sikeres amikor X, de változóbehelyettesítést nem okoz
azonos2(X, Y) :-
    \+ \+ (numbervars1(foo(X,Y), 0, _), X = Y).

| ?- azonos(X, 1).                          ----> no
| ?- azonos(X, Y).                          ----> no
| ?- azonos(X, X).                          ----> true ?
| ?- append([], L1, L2), azonos(L1, L2).    ----> L2 = L1 ?
```

# *Univ* alkalmazása: részkifejezések keresése

- A feladat: egy tetszőleges kifejezéshez soroljuk fel a benne levő számokat, és minden szám esetén adjuk meg annak a *kiválasztóját*!
- Egy részkifejezés kiválasztója egy olyan lista, amely megadja, mely argumentumpozíciók mentén juthatunk el hozzá.
- Az `[`$i_1, i_2, \ldots, i_k$`]` lista egy `Kif`-ből az $i_1$-edik argumentum $i_2$-edik argumentumának, ... $i_k$-adik argumentumát választja ki.
- Pl. `a*b+f(1,2,3)/c`-ben `b` kiválasztója `[1,2]`, `3` kiválasztója `[2,1,3]`.

```
% kif_szám(?Kif, ?N, ?Kiv): Kif Kiv kiválasztójú része az N szám.
kif_szám(X, N, Kiv) :-
        number(X), !, N = X, Kiv = [].
kif_szám(X, N, [I|Kiv]) :-
        compound(X), % a változó kizárása miatt fontos!
        functor(X, F, N), between(1, N, I), arg(I, X, X1),
        kif_szám(X1, N, Kiv).

| ?- kif_szám(f(1,[b,2]), N, K).
====> K = [1], N = 1 ? ;
      K = [2,2,1], N = 2 ? ; no
```

# Atomok szétszedése és összerakása

- `atom_codes/2`: atom és karakterkód-lista közötti átalakítás
  - Hívási minták: `atom_codes(+Atom, ?KódLista)`
    `atom_codes(-Atom, +KódLista)`
  - Jelentése: Igaz, ha `Atom` karakterkódjainak a listája `KódLista`.
  - Végrehajtása:
    - Ha `Atom` adott (bemenő), és a $c_1c_2...c_n$ karakterekből áll, akkor `KódLista`-t egyesíti a `[`$k_1, k_2, ..., k_n$`]` listával, ahol $k_i$ a $c_i$ karakter kódja.
    - Ha `KódLista` egy adott karakterkód-lista, akkor ezekből a karakterekből összerak egy atomot, és azt egyesíti `Atom`-mal.
- Példák:

```
| ?- atom_codes(ab, Cs).              ⟹ Cs = [97,98]
| ?- atom_codes(ab, [0'a|L]).         ⟹ L = [98]
| ?- Cs="bc", atom_codes(Atom, Cs).   ⟹ Cs = [98,99], Atom = bc
| ?- atom_codes(Atom, [0'a|L]).       ⟹ hiba
```

# Atomok szétszedése és összerakása — alkalmazási példák

- Keresés atomokban

```
% Atom-ban a Rész nem üres részatom kétszer ismétlődik.
dadogó_rész(Atom, Rész) :-
    atom_codes(Atom, Cs), dadogó(Cs, Ds), atom_codes(Rész, Ds).

% L-ben a D nem üres részlista kétszer ismétlődik (lásd korábban).
dadogó(L, D) :- D = [_|_],
    append(_, Farok, L), append(D, Vég, Farok), append(D, _, Vég).

| ?- dadogó_rész(babaruhaha, R).    ⟹  R = ba ? ; R = ha ? ; no
```

- Atomok összefűzése

```
% atom_concat(+A, +B, ?C): A és B atomok összefűzése C.
% (Szabványos beépített eljárás atom_concat(?A, ?B, +C) módban is.)
atom_concat(A, B, C) :-
    atom_codes(A, Ak), atom_codes(B, Bk),
    append(Ak, Bk, Ck),
    atom_codes(C, Ck).

| ?- atom_concat(abra, kadabra, A). ⟹ A = abrakadabra ?
```

# Számok szétszedése és összerakása

- `number_codes/2`: szám és karakterkód-lista közötti átalakítás
  - Hívási minták: `number_codes(+Szám, ?KódLista)`
    `number_codes(-Szám, +KódLista)`
  - Jelentése: Igaz, ha `Szám` tizes számrendszerbeli alakja a `KódLista` karakterkód-listának felel meg.
  - Végrehajtása:
    - Ha `Szám` adott (bemenő), és a $c_1c_2...c_n$ karakterekből áll, akkor `KódLista`-t egyesíti a `[`$k_1, k_2, ..., k_n$`]` kifejezéssel, ahol $k_i$ a $c_i$ karakter kódja.
    - Ha `KódLista` egy adott karakterkód-lista, akkor ezekből a karakterekből összerak egy számot (ha nem lehet, hibát jelez), és azt egyesíti `Szám`-mal.
- Példák:

```
| ?- number_codes(12, Cs).          ⟹ Cs = [49,50]
| ?- number_codes(0123, [0'1|L]).   ⟹ L = [50,51]
| ?- number_codes(N, " - 12.0e1").  ⟹ N = -120.0
| ?- number_codes(N, "12e1").       ⟹ hiba (nincs .0)
| ?- number_codes(120.0, "12e1").   ⟹ no (a szám adott! :-)
```

# Kifejezések rendezése: szabványos sorrend

- A Prolog szabvány definiálja két tetszőleges Prolog kifejezés szabványos sorrendjét.
- Jelölés: $X \prec Y$ — az $X$ kifejezés megelőzi az $Y$ kifejezést a szabványos sorrendben.
- A szabványos sorrend definíciója:
  1. Ha $X$ és $Y$ azonos, akkor sem $X \prec Y$ sem $Y \prec X$ nem igaz és fordítva.
  2. Ha $X$ és $Y$ különböző kifejezésosztályba tartozik, akkor az osztály dönt:
     *változó* $\prec$ *lebegőpontos szám* $\prec$ *egész szám* $\prec$ *név* $\prec$ *struktúra*.
  3. Ha $X$ és $Y$ változó, akkor az eredmény rendszerfüggő.
  4. Ha $X$ és $Y$ lebegőpontos vagy egész szám, akkor $X \prec Y \Leftrightarrow X < Y$.
  5. Ha $X$ és $Y$ név, akkor sorrendjük megegyezik a lexikografikus (abc) sorrenddel.
  6. Ha $X$ és $Y$ struktúrák:
     - 6.1. Ha $X$ és $Y$ aritása ($\equiv$ argumentumszáma) különböző, $X \prec Y \Leftrightarrow X$ aritása kisebb mint $Y$ aritása.
     - 6.2. Egyébként, ha a rekordok neve különböző, $X \prec Y \Leftrightarrow X$ neve $\prec Y$ neve.
     - 6.3. Egyébként (azonos név, azonos aritás) balról az első nem azonos argumentum dönt.
- (A SICStus Prologban kiterjesztésként megengedett végtelen (ciklikus) kifejezésekre a fenti rendezés nem érvényes.)

# Kifejezések összehasonlítása — beépített eljárások

- Két tetszőleges kifejezés összehasonlítását végző eljárások:

| hívás | igaz, ha |
|---|---|
| `Kif1 == Kif2` | `Kif1` $\not\prec$ `Kif2` $\wedge$ `Kif2` $\not\prec$ `Kif1` |
| `Kif1 \== Kif2` | `Kif1` $\prec$ `Kif2` $\vee$ `Kif2` $\prec$ `Kif1` |
| `Kif1 @< Kif2` | `Kif1` $\prec$ `Kif2` |
| `Kif1 @=< Kif2` | `Kif2` $\not\prec$ `Kif1` |
| `Kif1 @> Kif2` | `Kif2` $\prec$ `Kif1` |
| `Kif1 @>= Kif2` | `Kif1` $\not\prec$ `Kif2` |

- Az összehasonlító eljárások logikailag nem tiszták:

```
| ?- X @< 3, X = 4.  ⟹ X = 4
| ?- X = 4, X @< 3.  ⟹ no
```

- Az összehasonlítás mindig a belső ábrázolás szerint történik:

```
| ?- [1, 2, 3, 4] @< struktúra(1, 2, 3).
```
⟹ **sikerül (6.1 szabály)**

# A meta-logikai eljárások egy komplex alkalmazása: $\prec$ megvalósítása

```
% T1 megelőzi T2-t a szabványos sorrendben. (Ekvivalens T1 @< T2 -vel, kivéve
% a változókat, ezek rendezése a T1-T2-beli előfordulásuk szerint történik.)
precedes(T1, T2) :-
        \+ \+ (numbervars(T1-T2, 0, _), prec(T1, T2)).

% class(+T, -C): A T kifejezés a C-edik kifejezésosztályba tartozik.
class(T, C) :-
        (   T='$VAR'(_) -> C=0          % változó
        ;   float(T)    -> C=1          % lebegőpontos szám
        ;   integer(T)  -> C=2          % egész szám
        ;   atom(T)     -> C=3          % atom
        ;   compound(T) -> C=4          % összetett kifejezés
        ).

% T1 megelőzi T2-t, a változók már '$VAR'(n) struktúrákra vannak lecserélve.
prec(T1, T2) :-
        class(T1, C1), class(T2, C2),
        (   C1 =:= C2 ->
            (   C1 =:= 1 -> T1 < T2     % 4. szabály (lebegőpontos szám)
            ;   C1 =:= 2 -> T1 < T2     % 4. szabály (egész szám)
            ;   struct_prec(T1, T2)     % 3., 5. és 6. szabály
            )                           % (változó, név, struktúra)
        ;   C1 < C2                     % 2. szabály
        ).
```

# A $\prec$ reláció megvalósítása (folyt.)

```
% S1 megelőzi S2-t (S1 és S2 struktúra-kifejezés vagy atom).
struct_prec(S1, S2) :-
        functor(S1, F1, N1), functor(S2, F2, N2),
        (   N1 < N2 -> true
        ;   N1 = N2,
            (   F1 = F2 -> args_prec(1, N1, S1, S2)
            ;   atom_prec(F1, F2)
            )
        ).

% Az S1 struktúra-kifejezés N0, ..., N sorszámú argumentumai
% lexikografikusan megelőzik S2 azonos sorszámú argumentumait.
args_prec(N0, N, S1, S2) :-
        N0 =< N,
        arg(N0, S1, A1), arg(N0, S2, A2),
        (   A1 = A2 -> N1 is N0+1, args_prec(N1, N, S1, S2)
        ;   prec(A1, A2)
        ).

% A1 atom megelőzi A2 atomot.
atom_prec(A1, A2) :-
    atom_codes(A1, C1), atom_codes(A2, C2), struct_prec(C1, C2).
```

# EGYENLŐSÉGFAJTÁK — ÖSSZEFOGLALÁS

# A Prolog egyenlőség-szerű beépített eljárásai

- *U* `=` *V*: *U* egyesítendő *V*-vel.

  Soha sem jelez hibát.

```
?- X = 1+2.     ⟹ X = 1+2
?- 3 = 1+2.     ⟹ no
```

- *U* `==` *V*: *U* azonos *V*-vel.

  Soha sem jelez hibát és soha sem helyettesít be.

```
?- X == 1+2.    ⟹ no
?- 3 == 1+2.    ⟹ no
?- +(1,2)==1+2  ⟹ yes
```

- *U* `=:=` *V*: Az *U* és *V* aritmetikai kifejezések értéke megegyezik.

  Hibát jelez, ha *U* vagy *V* nem (tömör) aritmetikai kifejezés.

```
?- X =:= 1+2.   ⟹ hiba
?- 1+2 =:= X.   ⟹ hiba
?- 2+1 =:= 1+2. ⟹ yes
?- 2.0 =:= 1+1. ⟹ yes
```

- *U* `is` *V*: *U* egyesítendő a *V* aritmetikai kifejezés értékével.

  Hiba, ha *V* nem (tömör) aritmetikai kifejezés.

```
?- 2.0 is 1+1.  ⟹ no
?- X is 1+2.    ⟹ X = 3
?- 1+2 is X.    ⟹ hiba
?- 3 is 1+2.    ⟹ yes
?- 1+2 is 1+2.  ⟹ no
```

- (*U* `=..` *V*: *U* „szétszedettje” a *V* lista)

```
?- 1+2 =.. X.   ⟹ X = [+,1,2]
?- X =.. [f,1]. ⟹ X = f(1)
```

# A Prolog nem-egyenlőség jellegű beépített eljárásai

- A nem-egyenlőség jellegű eljárások soha sem helyettesítenek be változót!
  - `U \= V`: `U` nem egyesíthető `V`-vel.
    Soha sem jelez hibát.

```
?- X \= 1+2.        ⟹ no
?- +(1,2) \= 1+2.   ⟹ no
```

  - `U \== V`: `U` nem azonos `V`-vel.
    Soha sem jelez hibát.

```
?- X \== 1+2.       ⟹ yes
?- 3 \== 1+2.       ⟹ yes
?- +(1,2)\==1+2     ⟹ no
```

  - `U =\= V`: Az `U` és `V` aritmetikai kifejezések értéke különbözik.
    Hibát jelez, ha `U` vagy `V` nem (tömör) aritmetikai kifejezés.

```
?- X =\= 1+2.       ⟹ hiba
?- 1+2 =\= X.       ⟹ hiba
?- 2+1 =\= 1+2.     ⟹ no
?- 2.0 =\= 1+1.     ⟹ no
```

# A Prolog (nem-)egyenlőség jellegű beépített eljárásai — példák

| | | *Egyesítés* | | *Azonosság* | | *Aritmetika* | | |
|---|---|---|---|---|---|---|---|---|
| *U* | *V* | *U = V* | *U \= V* | *U == V* | *U \== V* | *U =:= V* | *U =\= V* | *U is V* |
| 1 | 2 | *no* | yes | *no* | yes | *no* | yes | *no* |
| a | b | *no* | yes | *no* | yes | error | error | error |
| 1+2 | +(1,2) | yes | *no* | yes | *no* | yes | *no* | *no* |
| 1+2 | 2+1 | *no* | yes | *no* | yes | yes | *no* | *no* |
| 1+2 | 3 | *no* | yes | *no* | yes | yes | *no* | *no* |
| 3 | 1+2 | *no* | yes | *no* | yes | yes | *no* | yes |
| X | 1+2 | X=1+2 | *no* | *no* | yes | error | error | X=3 |
| X | Y | X=Y | *no* | *no* | yes | error | error | error |
| X | X | yes | *no* | yes | *no* | error | error | error |

Jelmagyarázat: yes — siker; *no* — meghiúsulás, error — hiba.

# MODULARITÁS

# Modulok definiálása SICStus Prolog nyelven

- A SICStus Prolog modulfogalmának jellemzői:
  - Minden modul külön állományba kell kerüljön.
  - Az állomány első programeleme egy modul-parancs kell legyen:

    ```
    :- module( Modulnév, [ExpFunktor1, ExpFunktor2, ...]).
    ```

  - *ExpFunktor* = az exportálandó eljárás funktora (név/argumentumszám)
  - Példa:

    ```
    :- module(platók, [fennsík/3]).          % plato állomány első sora
    ```

  - Modul-betöltésre szolgáló beépített eljárások:
    - `use_module(`*ÁllományNév*`)`
    - `use_module(`*ÁllományNév*`, [`*ImpFunktor1*`,`*ImpFunktor2*`,...])`
      *ImpFunktor* — az importálandó eljárás funktora
    - *ÁllományNév* lehet atom, vagy pl. `library(`*KönyvtárNév*`)`:

      ```
      :- use_module(plato).                    % a fenti modul betöltése
      :- use_module(library(lists), [last/2]). % csak last/2 importált
      ```

  - Modulkvalifikált hívási forma: *Modul*`:`*Hívás* a *Modul*-ban futtatja *Hívás*-t.
  - A modulfogalom nem szigorú: `platók:első_fennsík(...)` meghívható!

# Meta-eljárások modularizált programban

- Eljárások átadása paraméterként modulközi hívásban gondot okozhat

`m1.pl` állomány:

```
:- module(m1, [kétszer/1]).

% :- meta_predicate kétszer(:).   (*)
kétszer(X) :-
        X, X.

p :- write(bu).
```

`m2.pl` állomány:

```
:- module(m2, [q/0,r/0]).

:- use_module(m1).

q :- kétszer(p).

r :- kétszer(m2:p).

p :- write(ba).
```

- Futtatás:

```
| ?- [m1,m2].
| ?- q.   ⟹ bubu
| ?- r.   ⟹ baba
```

- Automatikus modul-kvalifikáció meta-predikátum deklarációval:
  Ha `m1.pl`-ben elhagyjuk a `(*)`-gal jelzett sor előtti `%` kommentjelet, akkor

```
| ?- q.   ⟹ baba!
```

# Meta-predikátum deklaráció, modulnév-kiterjesztés

- Meta-predikátum deklaráció
  - Formája:

    `:- meta_predicate` ⟨`eljárásnév`⟩`(`⟨`módspec`$_1$⟩`, ..., `⟨`módspec`$_n$⟩`), ....`
  - ⟨`módspec`$_i$⟩ lehet '`:`', '`+`', '`-`', vagy '`?`'.
  - A '`:`' mód azt jelzi, hogy az adott argumentumot **betöltéskor** ún. modulnév-kiterjesztésnek kell alávetni. (A többi mód hatása azonos, be/kimenő irányt jelezhetünk segítségükkel.)
- Egy *`Kif`* kifejezés modulnév-kiterjesztése a következő átalakítást jelenti:
  - ha *`Kif`* `M:X` alakú, vagy egy olyan változó, amely az adott eljárás fejében meta-argumentum pozíción szerepelt, akkor változatlanul hagyjuk;
  - egyébként helyettesítjük *`CurMod`*`:`*`Kif`*-fel (*`CurMod`* a kurrens modul).
- Példa folyt. (tfh. az `m1`-beli `kétszer` meta-predikátumnak deklarált!)

```
:- module(m2, [négyszer/1,q/0]).
:- use_module(m1).                          % tárolt alak:
q :- kétszer(p).                        ⟹  q :- kétszer(m2:p).

:- meta_predicate négyszer(:).
négyszer(X) :- kétszer(X), kétszer(X).  ⟹  változatlan
```

# MAGASABBRENDŰ ELJÁRÁSOK

# Magasabbrendű eljárások — listakezelés

- Magasabbrendű (vagy meta-eljárás) egy eljárás,
  - ha eljárásként értelmezi egy vagy több argumentumát
  - pl. `call/1, findall/3, \+ /1, stb.`
- Listafeldolgozás `findall` segítségével — példák
  - Páros elemek kiválasztása

```
% Az L egész-lista páros elemeinek listája Pk.
páros_elemei(L, Pk) :-
        findall(X, (member(X, L), X mod 2 =:= 0), Pk).

| ?- páros_elemei([1,2,3,4], Pk).  ⟹ Pk = [2,4]
```

  - A listaelemek négyzetre emelése

```
% Az L számlista  elemei négyzeteinek listája Nk.
négyzetei(L, Nk) :-
        findall(Y, (member(X, L), Y is X*X), Nk).

| ?- négyzetei([1,2,3,4], Nk).    ⟹ Nk = [1,4,9,16]
```

# Általános listakezelő meta-eljárások, `findall/3`-ra építve

- Lista szűrése (vö. a `filter` SML függvénnyel!)

```
% Az L lista X elemeinek Pred szerinti szűrése FL.
:- meta_predicate filter(+, ?, :, -).
filter(L, X, Pred, FL) :-
        findall(X, (member(X, L), call(Pred)), FL).
```

```
| ?- filter([1,2,3,4], X, X mod 2 =:= 0, Pk).   ⟹ Pk = [2,4]
```

- Lista leképezése (vö. a `map` SML függvénnyel!)

```
% Az L lista X elemeit Pred-del Y-ba képezve
% kapjuk az ML listát.
:- meta_predicate map(+, ?, :, ?, -).
map(L, X, Pred, Y, ML) :-
        findall(Y, (member(X, L), Pred), ML).
```

```
| ?- map([1,2,3,4], X, Y is X*X, Y, Nk).   ⟹ Nk = [1,4,9,16]
```

- A példákban a szűrést az ⟨`X, Pred`⟩ argumentumpár, a leképezést az ⟨`X, Pred, Y`⟩ hármas határozza meg. Ezek egy egy- ill. kétargumentumú predikátumot írnak le (vö. a funkcionális nyelvek $\lambda$-kifejezéseivel).

# Részlegesen paraméterezett eljáráshívások

- A listát elemenként négyzetreemelő eljárás egy másik változata:

```
négyzete(X, Y) :- Y is X*X.

négyzeteik(Xk, Yk) :- map(Xk, X, négyzete(X,Y), Y, Yk).
```

- A lista elemeire az $x \rightarrow x^2 + \mathtt{P}x + \mathtt{Q}$ hozzárendelést alkalmazó eljárás:

```
másodfokú_képe(P, Q, X, Y) :- Y is X*X + P*X + Q.

másodfokú_képeik(P, Q, Xk, Yk) :-
        map(Xk, X, másodfokú_képe(P,Q,X,Y), Y, Yk).
```

- Konvenció: a meta-alkalmazásban változó paramétereket az eljárás végére tesszük — így egyszerűsíthető a meta-eljárás hívása. Példa: `map/3`:

```
map(Xk, RészlPred, Yk) :-
        RészlPred =.. L0, append(L0, [X,Y], L), Pred =.. L,    (*)
        findall(Y, (member(X, Xk), Pred), Yk).

másodfokú_képeik(P, Q, Xk, Yk) :-
        map(Xk, másodfokú_képe(P,Q), Yk).
```

- A `másodfokú_képe(P,Q)` kifejezés itt a `másodfokú_képe/4` **részlegesen paraméterezett** hívásának tekinthető — a hiányzó két paraméterrel való kiegészítés a (*) sorban történik.

# Részlegesen paraméterezett eljáráshívások — segédeszközök

- A `call/1` eljárás általánosítása: a `call/2`, `call/3`, ... eljárások.
- `call(RPred, A1, A2, ...)` végrehajtása: az `RPred` hívást kiegészíti az `A1, A2, ...` argumentumokkal, és meghívja.
- A `call/N` eljárások sok Prologban beépítettek, SICStusban definiálandók:

```
:- meta_predicate call(:, ?), call(:, ?, ?), ....

% Pred az A utolsó argumentummal meghívva igaz.
call(M:Pred, A) :-
        Pred =.. FAs0, append(FAs0, [A], FAs1),
        Pred1 =.. FAs1, call(M:Pred1).

% Pred az A és B utolsó argumentumokkal meghívva igaz.
call(M:Pred, A, B) :-
        Pred =.. FAs0, append(FAs0, [A,B], FAs2),
        Pred2 =.. FAs2, call(M:Pred2).

...
```

# Részlegesen paraméterezett eljárások — rekurzív `map/3`

- A részleges paraméterezés segítségével a `map/3` meta-eljárás rekurzívan is definiálható, `findall/3` nélkül:

```
% map(Xs, Pred, Ys): Az Xs lista elemeire a Pred transzformációt
% alkalmazva kapjuk az Ys listát.
map([X|Xs], Pred, [Y|Ys]) :-
        call(Pred, X, Y), map(Xs, Pred, Ys).
map([], _, []).
```

- Példák:

```
| ?- map([1,2,3,4], négyzete, L).              ⟹ L = [1,4,9,16]
| ?- map([1,2,3,4], másodfokú_képe(2,1), L).  ⟹ L = [4,9,16,25]
```

- A `call/N`-re épülő megoldás előnyei:
  - általánosabb és hatékonyabb lehet, mint a `findall`-ra épülő;
  - alkalmazható akkor is, ha az elemekre elvégzendő műveletek nem függetlenek, pl. `foldl`.

# Rekurzív meta-eljárások — `foldl` és `foldr`

```
% foldl(+Xs, :Pred, +Y0, -Y): Y0-ból indulva, az Xs elemeire balról jobbra
% sorra alkalmazva a Pred által leírt kétargumentumú függvényt kapjuk Y-t.
foldl([X|Xs], Pred, Y0, Y) :-
        call(Pred, X, Y0, Y1), foldl(Xs, Pred, Y1, Y).
foldl([], _, Y, Y).

jegyhozzá(Alap, Jegy, Szam0, Szam) :- Szam is Szam0*Alap+Jegy.

| ?- foldl([1,2,3], jegyhozzá(10), 0, E).  ⟹ E = 123
```

- Ugyanez SML-ben:

```
- fun jegyhozza alap (jegy,szam) = szam*alap+jegy;
> val jegyhozza = fn : int -> int * int -> int
- foldl (jegyhozza 10) 0 [1,2,3];
> val it = 123 : int
```

```
% foldr(+Xs, :Pred, +Y0, -Y): Y0-ból indulva, az Xs elemeire jobbról balra
% sorra alkalmazva a Pred kétargumentumú függvényt kapjuk Y-t.
foldr([X|Xs], Pred, Y0, Y) :-
        foldr(Xs, Pred, Y0, Y1), call(Pred, X, Y1, Y).
foldr([], _, Y, Y).

| ?- foldr([1,2,3], jegyhozzá(10), 0, E).  ⟹ E = 321
```

# DINAMIKUS ADATBÁZISKEZELÉS

# Dinamikus predikátumok

- A dinamikus predikátum jellemzői:
  - a program szövegében lehet 0 vagy több klóza;
  - futási időben hozzáadhatunk és elvehetünk klózokat belőle;
  - végrehajtása mindenképpen interpretált.
- Létrehozása
  - programszövegbeli deklarációval:

    `:- dynamic(Eljárásnév/Argumentumszám).`

    (ha van klóza a programban, akkor az első előtt — ilyenkor kötelező);
  - futási időben, adatbáziskezelő beépített eljárással
- Adatbáziskezelő eljárások („adatbázis” = a program klózainak összessége):
  - klóz felvétele első, utolsó helyre: `asserta/1, assertz/1`
  - klóz törlése (illesztéssel, többszörösen sikerülhet): `retract/1`
  - klóz lekérdezése (illesztéssel, többszörösen sikerülhet): `clause/2`

# Klóz felvétele: `asserta/1, assertz/1`

- `asserta(:@Klóz)`
  - A `Klóz` kifejezést klózként értelmezve felveszi a programba az adott predikátum *első* klózaként.
  - A '`@`' mód jelentése: tisztán bemenő paraméter, az eljárás a paraméterbeli változókat nem helyettesíti be (a '`+`' mód speciális esete).
  - A '`:`' mód modul-kvalifikált paramétert jelez.
- `assertz(:@Klóz)`
  - A `Klóz` kifejezést az adott predikátum *utolsó* klózaként veszi fel
- Példa:

```
| ?- assertz((p(1,X):-q(X))), asserta(p(2,0)),
     assertz((p(2,Z):-r(Z))), listing(p).

p(2, 0).
p(1, A) :-   q(A).
p(2, A) :-   r(A).
```

# Klóz törlése: `retract/1`

- `retract(:@Klóz)`
  - A `Klóz` klóz-kifejezésből megállapítja a predikátum funktorát.
  - Az adott predikátum klózait sorra megpróbálja illeszteni `Klóz`-zal.
  - Ha az illesztés sikerült, akkor kitörli a klózt és sikeresen lefut.
  - Visszalépés esetén folytatja a keresést (illeszt, töröl, sikerül, stb.)
- Példa (folytatás):

```
| ?- listing(p), retract((p(2,_):-_)), listing(p), fail. ⟹ no
```

- A futás kimenete:

```
p(2, 0).
p(1, A) :-
        q(A).
p(2, A) :-
        r(A).
```

```
p(1, A) :-
          q(A).
p(2, A) :-
          r(A).
```

```
p(1, A) :-
        q(A).
```

# Alkalmazási példa — egyszerűsített `findall`

- A `findall1/3` eljárás hatása megegyezik a beépített `findall`-lal, de
- nem működik helyesen, ha a `Cél`-ban újabb `findall1` hívás van.

```
:- dynamic(megoldás/1).

% findall1(Minta, Cél, L): Cél összes megoldására Minták listája L.
findall1(Minta, Cél, _MegoldL) :-
    call(Cél),
    asserta(megoldás(Minta)), % fordított sorrendben vesszük fel!
    fail.
findall1(_Minta, _Cél, MegoldL) :-
    megoldás_lista([], MegoldL).

% A megoldás/1 tényállításokban tárolt kifejezések fordított listája L-L0.
megoldás_lista(L0, L) :-
    retract(megoldás(M)), !,
    megoldás_lista([M|L0], L).
megoldás_lista(L, L).

| ?- findall1(Y, (member(X, [1,2,3]),Y is X*X), ML). ⟹ ML = [1,4,9]
```

# Klóz lekérdezése: `clause/2`

- `clause(:@Fej, ?Törzs)`
  - A `Fej` alapján megállapítja a predikátum funktorát.
  - Az adott predikátum klózait sorra megpróbálja illeszteni a `Fej :- Törzs` kifejezéssel (tényállítás esetén `Törzs = true`).
  - Ha az illesztés sikerült, akkor sikeresen lefut.
  - Visszalépés esetén folytatja a keresést (illeszt, sikerül, stb.)
- Példa:

```
:- listing(p), clause(p(2, 0), T).
```

```
p(2, 0).
p(1, A) :-
        q(A).
p(2, A) :-
        r(A).
```

```
T = true ? ;
T = r(0) ? ;
no
```

# A `clause` eljárás alkalmazása: egyszerű nyomkövető interpreter

- Az alábbi interpreter csak „tiszta”, beépített eljárást nem alkalmazó Prolog programok futtatására alkalmas.

```
% interp(G, D): A G cél futását D bekezdésű nyomkövetéssel mutatja.
interp(true, _) :- !.
interp((G1, G2), D) :- !,
    interp(G1, D), interp(G2, D).
interp(G, D) :-
    (   trace(G, D, call)
    ;   trace(G, D, fail), fail   % követi a fail kaput, tovább-hiúsul
    ),
    D2 is D+2,
    clause(G, B), interp(B, D2),
    (   trace(G, D, exit)
    ;   trace(G, D, redo), fail   % követi a redo kaput, tovább-hiúsul
    ).

% A G cél áthaladását a Port kapun D bekezdésű nyomkövetéssel mutatja.
trace(G, D, Port) :-
    /*D szóközt ír ki:*/ tab(D),
    write(Port), write(': '), write(G), nl.
```

# Nyomkövető interpreter - példafutás

```
:- dynamic app/3, app/4.

app([], L, L).
app([X|L1], L2, [X|L3]) :-
    app(L1, L2, L3).

app(L1, L2, L3, L123) :-
    app(L1, L23, L123),
    app(L2, L3, L23).
```

```
| ?-  interp(app(_, [b,c], L, [c,b,c,b]), 0).
call: app(_203,[b,c],_253,[c,b,c,b])
  call: app(_203,_666,[c,b,c,b])
  exit: app([],[c,b,c,b],[c,b,c,b])
  call: app([b,c],_253,[c,b,c,b])
  fail: app([b,c],_253,[c,b,c,b])
  redo: app([],[c,b,c,b],[c,b,c,b])
    call: app(_873,_666,[b,c,b])
    exit: app([],[b,c,b],[b,c,b])
  exit: app([c],[b,c,b],[c,b,c,b])
  call: app([b,c],_253,[b,c,b])
    call: app([c],_253,[c,b])
      call: app([],_253,[b])
      exit: app([],[b],[b])
    exit: app([c],[b],[c,b])
  exit: app([b,c],[b],[b,c,b])
exit: app([c],[b,c],[b],[c,b,c,b])
L = [b] ?
```

# NYELVTANI ELEMZÉS PROLOGBAN

# Egy egyszerű nyelvtani elemzési példa

- Bináris számok nyelvtana

  ⟨ szám ⟩ ::= ⟨ számjegy ⟩ ⟨ számmaradék ⟩
  ⟨ számmaradék ⟩ ::= ⟨ számjegy ⟩ ⟨ számmaradék ⟩ | $\epsilon$
  ⟨ számjegy ⟩ ::= 0 | 1

- Ugyanez DCG (Definite Clause Grammar) jelöléssel:

```
szám -->              számjegy, számmaradék.
számmaradék -->       számjegy, számmaradék | "".
számjegy -->          "0" | "1".
```

- A definit klóz nyelvtan (DCG):
  - egy általános nyelvtani formalizmus,
  - amely egyszerűen Prologra fordítható,
  - a legtöbb Prolog rendszer része (bár a szabványnak nem).

# Nyelvtani elemzés „bevetítése” Prologba

- Nyelvtani elemzés: annak eldöntése, hogy egy (Prolog listában tárolt) jelsorozat megfelel-e egy adott nem-terminális nyelvtani fogalomnak.
- A lista tetszőleges elemekből állhat, pl. karakterkódok listája, lexikai elemek (token-ek) listája.
- A nem-terminálisoknak kétargumentumú Prolog szabályok felelnek meg, pl.

```
szám -->        számjegy,          számmaradék.
szám(L0, L) :- számjegy(L0, L1), számmaradék(L1, L).
% Az L0 kódlistáról "leelemezhető" egy <szám>, marad L ha
%           L0-ról leelemezhető egy <számjegy>, marad L1, és
%           L1-ről leelemezhető egy <számmaradék>, marad L.
```

- Általánosan: az adott nem-terminálisnak megfelelő jelsorozatot „leelemezve” (lehagyva) egy `L0` lista elejéről marad egy `L` lista.
- Terminális szimbólumok esetén egyetlen elemet kell lehagyni a listáról, erre szolgál a `'C'/3` beépített eljárás. Definíciója: `'C'(L0, X, L) :- L0 = [X|L].`
  (A SICStus fordító a `'C'/3` hívást ténylegesen a fenti egyenlőséggel helyettesíti.)
- A „leelemzés” tulajdonképpen akkumulálási folyamat, ahol az elemi akkumulálási lépés: egy terminális lehagyása a lista elejéről (`'C'/3`).

# A DCG szabályok lefordított alakja

- A korábbi DCG példa:

```
szám -->              számjegy, számmaradék.        % A | B ≡ A ; B
számmaradék -->       számjegy, számmaradék | "".    % ""  ≡ []
számjegy -->          "0" | "1".                     % "0" ≡ [48]
```

- A fenti DCG szabályok betöltésekor a következő Prolog kód keletkezik:

```
szám(L0, L) :-
        számjegy(L0, L1), számmaradék(L1, L).

számmaradék(L0, L) :-
        (   számjegy(L0, L1), számmaradék(L1, L)
        ;   L = L0
        ).

számjegy(L0, L) :-
        (   'C'(L0, 48, L)
        ;   'C'(L0, 49, L)
        ).
```

- A DCG elemző futtatása:

```
| ?- szám("101", "").  ⟹ yes                          % "101" ≡ [0'1,0'0,0'1]
| ?- szám("102", L).   ⟹ L = "2" ; L = "02" ; no % Valójában L = [50] ; ...
```

# Vezérlési szerkezetek DCG szabályokban

- DCG szabályokban használható: vágó, diszjunkció, negáció és feltételes diszjunktív szerkezet.
- Ezek változtatás nélkül átkerülnek a Prolog alakba. Példák:

```
% Leelemezhető számjegyek egy MAXIMÁLIS (esetleg üres) listája.
számmaradék -->
        (   számjegy -> számmaradék
        ;   []                          % Vigyázat: [] helyett true nem jó!
        ).

% Ugyanez vágóval
számmaradék -->   számjegy, !, számmaradék.
számmaradék -->   [].                   % Figyelem: nincsenek DCG tényállítások!

% Az utóbbi Prolog alakja:
számmaradék(L0, L) :-
        számjegy(L0, L1), !, számmaradék(L1, L).
számmaradék(L0, L) :-
        L = L0.

| ?- számmaradék("102", L).  ⟹ L = "2" ; no
```

# Prolog hívás beillesztése DCG szabályba

- Általánosabb példa: decimális számjegyek elemzése

```
számjegy --> "0" ; "1" ; "2" ; "3" ; "4" ;
             "5" ; "6" ; "7" ; "8" ; "9".

% Ugyanez általánosabban és egyszerűbben:
számjegy -->
        [K],                          % K a következő terminális
        {decimális_jegy_kódja(K)}.    % Prolog hívás

% K egy számjegy kódja.
decimális_jegy_kódja(K):-
        K >= 0'0, K =< 0'9.
```

- A fenti DCG szabály Prolog megfelelője:

```
% Leelemezhető egy számjegy kódja.
számjegy(L0, L) :-
        'C'(L0, K, L),                % K a következő terminális
        decimális_jegy_kódja(K).      % megfelelő-e a K?
```

# Az elemző kiegészítése argumentumokkal

- Egy DCG szabály az elemzéssel párhuzamosan további (kimenő) argumentum(ok)ban felépítheti a kielemzett dolog „jelentését”, pl. egy elemzési fát, vagy annak egy kiértékelését.
- Példa: szám elemzése és értékének kiszámítása:

```
% leelemezhető egy Sz értékű decimálisszámjegy-sorozat
szám(Sz) --> számjegy(J), számmaradék(J, Sz).

% leelemezhető számjegyek egy esetleg üres listája, amelynek
% az eddig leelemzett Sz0-val együtt vett értéke Sz.
számmaradék(Sz0, Sz) -->
    számjegy(J), !, {Sz1 is Sz0*10+J}, számmaradék(Sz1, Sz).
számmaradék(Sz0, Sz0) --> [].

% leelemezhető egy J értékű számjegy.
számjegy(J) --> [K], {decimális_jegy_kódja(K), J is K-0'0}.
```

```
| ?- szám(Sz, "102 56", L). ⟹ L = " 56", Sz = 102; no
```

- A `számmaradék` DCG szabály Prolog alakja:

```
számmaradék(Sz0, Sz, L0,L) :-
    számjegy(J, L0,L1), !, Sz1 is Sz0*10+J, számmaradék(Sz1, Sz, L1,L).
számmaradék(Sz0, Sz0, L0,L) :- L=L0.
```

- Vegyük észre, hogy itt két akkumulátorpár van, egy „kézi” (`Sz`) és egy DCG-ből generált (`L`).

# A DCG nyelvtani szabályok szerkezete — összefoglalás

- A DCG szabály alakja: ⟨ *Baloldal* ⟩ `-->` ⟨ *Jobboldal* ⟩ `.`
- ⟨ *Baloldal* ⟩: egy nem-terminális(, amit esetleg terminálisok listája követ).
- ⟨ *Jobboldal* ⟩: konjunkció (`,`), diszjunkció (`;`), ha-akkor (`->`) és negáció (`\+`) segítségével épül fel terminálisokból, nem-terminálisokból és Prolog hívásokból.
- Nem-terminális: tetszőleges *hívható* kifejezés (atom vagy struktúra).
- Terminális: *tetszőleges* Prolog kifejezés; 0, 1 vagy több terminális jel sorozata *listaként* helyezhető el a DCG szabályokban.
- Prolog hívás: `{}` zárójelekbe zárva helyezhető el (vágó köré nem kell zárójel).
- A DCG egy darab „automatikus” akkumulátort biztosít (az akkumulálási lépés: `'C'`, egy elem levétele):

```
p(A,...) -->
    q0(B,...), ...,      [X],              qi(C,...), ...,
                                                {Cél}, ..., qn(D,...).

p(A,...,L0,L):-
    q0(B,...,L0,L1), ..., 'C'(Li-1, X, Li), qi(C,...,Li,Li+1),...,
                                              Cél, ...,  qn(D,...,Ln,L).
```

# DCG példa: kifejezés kiértékelése

- Egyszerű aritmetikai kifejezés elemzése és kiértékelése.

```
% kif(Z, L0, L): L0 elején egy Z értékű aritmetikai kifejezés áll, marad L.
kif(Z) --> tag(X), "+", kif(Y), {Z is X + Y}.
kif(Z) --> tag(X), "-", kif(Y), {Z is X - Y}.
kif(X) --> tag(X).

% tag(Z, L0, L): L0-ból leelemezhető egy Z értékű tag, marad L.
tag(Z) --> szám(X), "*", tag(Y), {Z is X * Y}.
tag(Z) --> szám(X), "/", tag(Y), {Z is X / Y}.
tag(X) --> szám(X).
```

```
| ?- kif(Z, "10*10-6*6", "").   ⟹ Z = 64 ;  no
| ?- kif(Z, "10*10-6*6", L).   ⟹ L = [], Z = 64 ; L = "*6", Z = 94 ; ...
| ?- kif(Z, "4-2+1", []).      ⟹ Z = 1   Probléma: jobbról balra elemez!
```

- Egy lehetséges javítás

```
kif(Z) --> tag(X), kifmaradék(X, Z).

kifmaradék(X, Z) --> "+", tag(Y), W is X + Y, kifmaradék(W, Z).
kifmaradék(X, Z) --> "-", tag(Y), W is X - Y, kifmaradék(W, Z).
kifmaradék(X, X) --> [].
...
```

# Egy nagyobb DCG példa: „természetes” nyelvű beszélgetés

```
:- use_module(library(lists)).

% mondat(Alany, Áll, L0, L): L0-L kielemezhető egy Alany alanyból és Áll
% állítmányból álló mondattá. Alany lehet első vagy második személyű
% névmás, vagy egyetlen szóból álló (harmadik személyű) alany.
mondat(Alany, Áll) -->
        {én_te(Alany, Ige)}, én_te_perm(Alany, Ige, Áll).
mondat(Alany, Áll) -->
        szó(Alany), szavak(Áll).

% én_te(Alany, Ige):
% Az Alany első/második személyű névmásnak megfelelő létige az Ige.
én_te("én", "vagyok").
én_te("te", "vagy").

% én_te_perm(Ki, Ige, Áll, L0, L): L0-L kielemezhető egy Ki
% névmásból, Ige igealakból és Áll állítmányból álló mondattá.
én_te_perm(Alany, Ige, Áll)  -->
        (   szó(Alany), szó(Ige), szavak(Áll)
        ;   szó(Alany), szavak(Áll), szó(Ige)
        ;   szavak(Áll), szó(Ige), szó(Alany)
        ;   szavak(Áll), szó(Ige)
        ).
```

# Példa: “természetes” nyelvű beszélgetés — szavak elemzése

```
% szó(Sz, L0, L): L0-L egy Sz betűsorozatból álló (nem üres) szó.
szó(Sz) -->
        betű(B), szómaradék(SzM), {illik([B|SzM], Sz)}, köz.

% szómaradék(Sz, L0, L): L0-L egy Sz kódlistából álló (esetleg üres) szó.
szómaradék([B|Sz]) -->
        betű(B), !, szómaradék(Sz).
szómaradék([]) --> [].

% illik(Szó0, Szó): Szó0 = Szó, vagy a kezdő kis-nagy betűben különböznek.
illik([B0|L], [B|L]) :-
        (   B = B0 -> true
        ;   abs(B-B0) =:= 32
        ).

% köz(L0, L): L0-L nulla, egy vagy több szóköz.
köz --> (   " " -> köz ; ""   ).

% betű(K, L0, L): L0-L egy K kódú "betű" (különbözik a " .?" jelektől)
betű(K) --> [K], {\+ member(K, " .?")}.

% szavak(SzL, L0, L): L0-L egy SzL szó-lista.
szavak([Sz|Szk]) -->
        szó(Sz), (   szavak(Szk)
                 ;   {Szk = []}
                 ).
```

# Példa: “természetes” nyelvű beszélgetés — párbeszéd-szervezés

```
% :- type mondás ---> kérdez(szó) ; kijelent(szó,list(szó)) ; un.

% Megvalósít egy párbeszédet.
párbeszéd :-
        repeat,
            read_line(L),   % beolvas egy sort, L a karakterkódok listája
            (   menet(Mondás, L, [])
            ->  feldolgoz(Mondás)
            ;   write('Nem értem\n'), fail
            ),
        Mondás = un, !.

% menet(Mondás, L0, L): Az L0-L kielemzett alakja Mondás.
menet(kérdez(Alany)) -->
        {kérdő(Szó)}, mondat(Alany, [Szó]), "?".
menet(kijelent(Alany,Áll)) -->
        mondat(Alany, Áll), ".".
menet(un) -->
        szó("unlak"), ".".

% kérdő(Szó): Szó egy kérdőszó.
kérdő("mi").
kérdő("ki").
kérdő("kicsoda").
```

# Példa: “természetes” nyelvű beszélgetés — válaszok előállítása

```
:- dynamic tudom/2.

% feldolgoz(Mondás): feldolgozza a felhasználótól érkező Mondás üzenetet.
feldolgoz(un) :-
        write('Én is.\n').
feldolgoz(kijelent(Alany, Áll)) :-
        assertz(tudom(Alany,Áll)),
        write('Felfogtam.\n').
feldolgoz(kérdez(Alany)) :-
        tudom(Alany, _), !,
        válasz(Alany).
feldolgoz(kérdez(_)) :-
        write('Nem tudom.\n').

% Felsorolja az Alany ismert tulajdonságait.
válasz(Alany) :-
        tudom(Alany, Áll),
        (   member(Szó, Áll), format('~s ', [Szó]), fail
        ;   nl
        ),
        fail.
válasz(_).
```

# Beszélgetős DCG példa — egy párbeszéd

```
| ?- 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.
|: Én vagyok Jeromos.
Felfogtam.
|: Te egy Prolog program vagy.
Felfogtam.
|: Ki vagyok én?
Magyar legény
Boldog
Jeromos
|: Okos vagy.
Felfogtam.
|: Ki vagy  te?
egy Prolog program
Okos
|: Valóban?
Nem értem
|: Unlak.
Én is.
```

# A DCG formalizmus felhasználása elemzésen kívül

- A DCG szabályok kényelmesen használhatók általános akkumulálásra
  - Listák akkumulálása — az elemi akkumulálási lépést a `'C'/3` adja

```
% anbn(+N, ?L): Az L lista N db a-ból és azt követő N db b-ből áll.
% Nem csak elemzésre, hanem L felépítésére is használható!
anbn(N, L) :- anbn(N, L, []).

% anbn(N, L0, L): L0-L N db a-ból és azt követő N db b-ből áll.
anbn(0) --> !.
anbn(N) --> {N > 0, N1 is N-1}, [a], anbn(N1), [b].

% a fenti DCG szabály kifejtve:
anbn(N, L0, L) :-
    N > 0, N1 is N-1, L0=[a|L1], anbn(N1, L1, L2), L2=[b|L].
```

  - Egyébként az elemi akkumulálási lépést DCG-n kívül kell megírni:

```
% sum(L, S0, S): L összege S-S0.
sum([]) --> [].
sum([X|L]) -->
        plus(X), sum(L).
```

```
% L számlista összege S.
sum(L, S) :- sum(L, 0, S).

plus(X, S0, S) :- S is S0+X.
```

# „HAGYOMÁNYOS” BEÉPÍTETT ELJÁRÁSOK

# Aritmetikai beépített eljárások

- `X is Kif`: `Kif` aritmetikai kifejezés kell legyen, értékét egyesíti `X`-szel.
- `Kif1` $\rho$ `Kif2`: `Kif1` és `Kif2` aritmetikai kifejezések kell legyenek, értékeik között elvégzi a $\rho$ összehasonlítást ($\rho$ lehet `=, =\=, <, =<, >, >=`).
- Aritmetikai kifejezésekben felhasználható funktorok:

| Infix operátorok | | | | | |
|---|---|---|---|---|---|
| `+` | összeadás | `//` | egész osztás | `/\` | bitenkénti és |
| `-` | kivonás | `**` | hatványozás | `\/` | bitenkénti vagy |
| `*` | szorzás | `mod` | modulus képzés | `<<` | bitenkénti balra léptetés |
| `/` | osztás | `rem` | maradék képzés | `>>` | bitenkénti jobbra léptetés |
| Prefix operátorok: | | `-` | negáció | `\` | bitenkénti negáció |

| Függvény jelölésűek | | | |
|---|---|---|---|
| `abs/1` | `exp/1` | `floor/1` | `sign/1` |
| `atan/1` | `float/1` | `log/1` | `sin/1` |
| `ceiling/1` | `float_fractional_part/1` | `max/2,min/2` | `sqrt/1` |
| `cos/1` | `float_integer_part/1` | `round/1` | `truncate/1` |

# Listakezelő beépített eljárások

- Lista hossza: `length(?L, ?N)`
  - Jelentése: az `L` lista hossza `N`.
  - `length(-L, +N)` módban adott hosszúságú, csupa különböző változóból álló listát hoz létre.
  - `length(-L, -N)` módban rendre felsorolja a 0, 1, ... hosszú listákat.
  - Megvalósítását lásd korábban.
- Lista rendezése: `sort(@L, ?S)`
  - Jelentése: az `L` lista `@<` szerinti rendezése `S`, (`==/2` szerint azonos elemek ismétlődését kiszűrve).
- Lista kulcs szerinti rendezése: `keysort(@L, ?S)`
  - Az `L` argumentum `Kulcs-Érték` alakú kifejezések listája.
  - Az eljárás jelentése: az `S` lista az `L` lista `Kulcs` értékei szerinti szabványos (`@<` általi) rendezése, ismétlődéseket nem szűr.

# Kifejezések kiírása

- `write(@X)`: Kiírja `X`-et, ha szükséges operátorokat, zárójeleket használva.
- `writeq(@X)`: Mint `write(X)`, csak gondoskodik, hogy szükség esetén az atomok idézőjelek közé legyenek téve.
- `write_canonical(@X)`: Mint `writeq(X)`, csak operátorok nélkül, minden struktúra szabványos alakban jelenik meg.
- `write_term(@X, +Opciók)`: Az `Opciók` opciólista szerint kiírja `X`-et.
- `format(@Formátum, @AdatLista)`: A `Formátum`-nak megfelelő módon kiírja `AdatLista`-t. A formázójelek alakja: `~`⟨`szám esetleg`⟩⟨`formázójel`⟩.

```
| ?- write('Helló világ').                    ⟹ Helló világ
| ?- writeq('Helló világ').                   ⟹ 'Helló világ'
| ?- write_canonical('*' - '%').              ⟹ -(*,'%')
| ?- write_canonical([1,2]).                  ⟹ '.'(1,'.'(2,[]))
| ?- write_term([1,2,3], [max_depth(2)]).     ⟹ [1,2|...]
| ?- format('X=~s --- ~3d s', [[0'j,0'ó],3245]). ⟹ X=jó --- 3.245 s
```

# Kifejezések kiírása — felhasználó vezérelte formázás

- `print(@X)`: Alapértelmezésben azonos `write`-tal. Ha a felhasználó definiál egy `portray/1` eljárást, akkor a rendszer minden a `print`-tel kinyomtatandó részkifejezésre meghívja `portray`-t. Ennek sikere esetén feltételezi, hogy a kiírás megtörtént, meghiúsulás esetén maga írja ki a részkifejezést.

  A rendszer a `print` eljárást használja a változó-behelyettesítések és a nyomkövetés kiírására!

- `portray(@Kif)` (felhasználó által definiálandó ún. *kampó eljárás*): Igaz, ha `Kif` kifejezést a Prolog rendszernek *nem* kell kiírnia (és ekkor maga a `portray` kell, hogy elvégzze a kiírást).

- Példa:

```
portray(Matrix) :-
     Matrix = [[_|_]|_],
     (   member(Row, Matrix),
         nl, print(Row), fail
     ;   true
     ).
```

```
| ?- X = [[1,2],[3,4],[5,6]].

X =
[1,2]
[3,4]
[5,6] ?
```

# Karakterek kiírása és beolvasása

- `put_code(+Kód)`: Kiírja az adott kódú karaktert.
- `tab(+N)`: Kiír `N` szóközt feltéve, hogy $N > 0$.
- `nl`: Kiír egy soremelést.
- `get_code(?Kód)`: Beolvas egy karaktert és (karakterkódját) egyesíti `Kód`-dal. (File végénél `Kód = -1`.)
- `peek_code(?Kód)`: A soronkövetkező karakter kódját egyesíti `Kód`-dal. A karaktert nem távolítja el a bemenetről. (File végénél `Kód = -1`.)
- Példa:

```
% rd_line(L): L a következő sor karakterkódjainak listája.
% read_line néven beépített eljárás SICStus 3.9.0-tól.
rd_line(L) :-
        peek_code(0'\n), !, get_code(_), L = [].
rd_line([C|L]) :-
        get_code(C), rd_line(L).

| ?- rd_line(L), tab(20), member(X, L), put_code(X), tab(1), fail ; nl.
|: Hello world!
                    H e l l o   w o r l d !
```

# Példa: számbeolvasás

```
% számbe(Szám): a Szám szám következik az input-folyamban.
számbe(Szám) :-
        számjegy(Érték),
        számbe(Érték, Szám).

% Az eddig beolvasott Szám0-val együtt az input-folyamban következő
% szám értéke Szám.
számbe(Szám0, Szám) :-
        számjegy(E), !,
        Szám1 is Szám0*10+E,
        számbe(Szám1, Szám).
számbe(Szám, Szám).

% Érték értékű számjegy következik.
számjegy(Érték) :-
        peek_code(Kar),
        Kar >= 0'0, Kar =< 0'9,
        get_code(_),
        Érték is Kar - 0'0.
```

```
| ?- számbe(X), get_code(_), számbe(Y).
|: 123 456
                    ⟹ X = 123, Y = 456
```

# Kifejezések beolvasása

- `read(?Kif)`: Beolvas egy ponttal lezárt kifejezést és egyesíti `Kif`-fel.
  (File végénél `Kif = end_of_file`.)
- `read_term(?Kif, +Opciók)`: Mint `read/1`, de az `Opciók` opciólistát is figyelembe veszi.
- Példa — botcsinálta programbeolvasó:

```
consult_body :-
        repeat,
            read(Term),
            (   Term = end_of_file -> true
            ;   assertz(Term), fail
            ),
        !.

| ?- consult_body.
|: p(X) :- q(X), r(X).
|: ^D
yes
```

```
| ?- listing([p/1]).
p(A) :-
        q(A),
        r(A).
yes
```

# Be- és kiviteli csatornák

- Csatornák megnyitása és kezelése:
  - `open(@Filenév, @Mód, -Csatorna)`: Megnyitja a `Filenév` nevű állományt `Mód` módban (`read`, `write` vagy `append`). A `Csatorna` argumentumban visszaadja a megnyitott csatorna „nyelét”.
  - `set_input(@Csatorna)`, `set_output(@Csatorna)`: Az ezt követő beviteli/kiviteli eljárások `Csatorna`-t használják majd (jelenlegi csatorna).
  - `current_input(?Csatorna)`, `current_output(?Csatorna)`: A jelenlegi beviteli/kiviteli csatornát egyesíti `Csatorna`-val.
  - `close(@Csatorna)`: Lezárja a `Csatorna` csatornát.
- Explicit csatornamegadás be- és kiviteli eljárásokban
  - Az eddig ismertetett összes be- és kiviteli eljárásnak van egy eggyel több argumentumú változata, amelynek első argumentuma a csatorna. Ezek: `write/2`, `writeq/2`, `write_canonical/2`, `write_term/3`, `print/2`, `read/2`, `read_term/3`, `format/3`, `put_code/2`, `tab/2`, `nl/1`, `get_code/2`, `peek_code/2`.

# Egy egyszerűbb be- és kiviteli szervezés: DEC10 I/O

- `see(@Filenév)`, `tell(@Filenév)`: Megnyitja a `Filenév` file-t olvasásra/írásra és a jelenlegi csatornává teszi. Újabb híváskor csak a jelenlegi csatornává teszi.
- `seeing(?Filenév)`, `telling(?Filenév)`: A jelenlegi beviteli/kiviteli csatorna állománynevét egyesíti `Filenév`-vel.
- `seen`, `told`: Lezárja a jelenlegi beviteli/kiviteli csatornát.
- Példák — nagyon egyszerű `consult` variánsok:

```
consult_dec10_style(File) :-
        seeing(Old), see(File),
        repeat,
            read(Term),
            (   Term = end_of_file
            ->  seen
            ;   assertz(Term), fail
            ),
        !,
        see(Old).
```

```
consult_with_streams(File) :-
        open(File, read, S),
        repeat,
            read(S, Term),
            (   Term = end_of_file
            ->  close(S)
            ;   assertz(Term), fail
            ),
        !.
```

# Hibakezelési beépített eljárások

- Hibahelyzetet beépített eljárás rossz argumentumokkal való meghívása, vagy a `throw/1` (`raise_exception/1`) eljárás válthat ki.
- Minden hibahelyzetet egy Prolog kifejezés (ún. hiba-kifejezés) jellemez.
- Hiba „dobása”, azaz a `HibaKif` hibahelyzet kiváltása:
```
throw(@HibaKif),
raise_exception(@HibaKif)
```
- Hiba „elkapása”:
```
catch(:+Cél, ?Minta, :+Hibaág),
on_exception(?Minta, :+Cél, :+Hibaág)
```
  - Hatása: Futtatja a `Cél` hívást.
    - Ha `Cél` végrehajtása során hibahelyzet nem fordul elő, futása azonos `Cél`-lal.
    - Ha `Cél`-ban hiba van, a hiba-kifejezést egyesíti `Mintá`-val.
    - Ha ez sikeres, meghívja a `Hibaág`-at.
    - Ellenkező esetben továbbdobja a hiba-kifejezést, hogy a további körülvevő `catch` eljárások esetleg elkaphassák azt.

# Programfejlesztési beépített eljárások (SICStus specifikusak)

- `set_prolog_flag(+Jelző, @Érték)`: `Jelző` értkékét `Érték`-re állítja.
- `current_prolog_flag(?Jelző, ?Érték)`: `Jelző` pillanatnyi értéke `Érték`.
- Néhány fontos Prolog jelző:
  - `language`: végrehajtási mód (`sicstus`, `iso`).
  - `argv`: csak olvasható, a parancssorbeli argumentumok listája.
  - `unknown`: viselkedés definiálatlan eljárás hívásakor (`trace`, `fail`, `error`).
  - `source_info`: forrásszintű nyomkövetés (`on`, `off`, `emacs`).
- `consult(:@Files)`, `[:@File,...]`: Betölti a `File`(ok)at, interpretált alakban.
- `compile(:@File)`: Betölti a `File`(ok)at, leforditott alakot hozva létre.
- `listing`: Kiírja az összes interpretált eljárást az aktuális kimenetre.
- `listing(:@EljárásSpec)`: Kiírja a megnevezett interpretált eljárásokat.
- Itt és később: `EljárásSpec` — név vagy funktor, eseteg modul-kvalifikációval ellátva, ill. ezek listája, pl. `listing(p)`, `listing([m:q,p/1])`.

# Programfejlesztési eljárások (folytatás)

- `statistics`: Különféle statisztikákat ír ki az aktuális kimenetre.
- `statistics(?Fajta, ?Érték)`: `Érték` a `Fajta` fajtájú mennyiség értéke.
  - Példa: `statistics(runtime, E)` $\Longrightarrow$`E=[Tdiff, T]`, `Tdiff` az előző lekérdezés óta, `T` a rendszerindítás óta eltelt idő, ezredmásodpercben.
- `break`: Egy új interakciós szintet hoz létre.
- `abort`, `halt`: Kilép a legkülső interakciós szintre ill. a Prolog rendszerből.
- `trace`: Elindítja az interaktív nyomkövetést.
- `debug`, `zip`: Elindítja a szelektív nyomkövetést, csak spion-pontoknál áll meg.
  (A `zip` mód gyorsabb, de nem gyűjt annyi információt mint a `debug` mód.)
- `nodebug`, `notrace`, `nozip`: Leállítja a nyomkövetést.
- `spy(:@EljárásSpec)`: Spion-pontot tesz a megadott eljárásokra.
- `nospy(:@EljárásSpec)`: Megszünteti a megadott spion-pontokat.
- `nospyall`: Az összes spion-pontot megszünteti.

# FEJLETTEBB NYELVI ÉS RENDSZERELEMEK

# Külső nyelvi interfész

- Hagyományos (pl. C nyelvű) programrészek meghívásának módja:
  - A Prolog rendszer elvégzi az átalakítást a Prolog alak és a külső nyelvi alak között. Kényelmesebb, biztonságosabb mint a másik módszer, de kevésbé hatékony. Többnyire csak egyszerű adatokra (egész, valós, atom). (MProlog)
  - A külső nyelvi rutin pointereket kap Prolog adatstruktúrákra, valamint hozzáférési algoritmusokat ezek kezelésére. Nehézkesebb, veszélyesebb, de jóval hatékonyabb mint az előző megoldás. Összetett adatok adásvételére is jó. (SWI, SICStus)

# Külső nyelvi interfész — példa

- A példa a `library(bdb)` megvalósításából származik.
- A C nyelven megirandó eljárás Prolog hívási alakja:
  `index_keys(+Spec, +Kif, -Kulcs, -Szám)`
- A megirandó eljárás jelentése:
  - Ha *`Spec`* és *`Kif`* különböző funktorú kifejezések, akkor *`Szám`* `= -1` és *`Kulcs`* `= []`.
  - Egyébként, ha *`Spec`* valamelyik argumentuma + és *`Kif`* megfelelő argumentuma változó, akkor *`Szám`* `= -2` és *`Kulcs`* `= []`.
  - Egyébként *`Szám`* a *`Spec`* argumentumaként előforduló + atomok száma, *`Kulcs`* pedig *`Kif`* megfelelő argumentumainak *kivonatából* képzett lista. A kivonat lényegében az argumentum funktora, azzal az eltéréssel, hogy a konstansok kivonata maga a konstans, struktúrák esetén pedig a struktúra neve és az aritása külön elemként kerül a kivonat-listába.

# Külső nyelvi interfész — példa

- A példaeljárás használata

```
| ?- [ixtest].
| ?- index_keys(f(+, -, +, +),
                f(12.3, _, s(1, _, z(2)), t),
                Kulcs, Szam).
Kulcs = [12.3,s,3,t], Szam = 3 ?
```

- Az `ixtest.pl` Prolog file tartalmazza az interfész specifikációját:

```
foreign(ixkeys, index_keys(+term, +term, -term, [-integer])).
                            % 1. arg: bemenő, általános kifejezés
                            % 2. arg: bemenő, általános kifejezés
                            % 3. arg: kimenő, általános kifejezés
                            % 4. arg: a C függvény értéke, egész (long)
foreign_resource(ixkeys, [ixkeys]).

:- load_foreign_resource(ixkeys).
```

- A C programot elő kell készíteni a Prolog számára az `splfr` (link foreign resource) eszköz segítségével:

```
splfr ixkeys ixtest.pl +c ixkeys.c
```

# Külső nyelvi interfész — a C kód (`ixkeys.c` állomány)

```
#include <sicstus/sicstus.h>

#define NA -1  /* not applicable */
#define NI -2  /* instantiatedness */

long ixkeys(SP_term_ref spec,
      SP_term_ref term, SP_term_ref list)
{
  unsigned long sname, tname, plus;
  int sarity, tarity, i;
  long ret = 0;
  SP_term_ref arg = SP_new_term_ref(),
              tmp = SP_new_term_ref();

  SP_get_functor(spec, &sname, &sarity);
  SP_get_functor(term, &tname, &tarity);
  if (sname != tname || sarity != tarity)
    return NA;

  plus = SP_atom_from_string("+");

  for (i = sarity; i > 0; --i) {
    unsigned long t;
    SP_get_arg(i, spec, arg);
    SP_get_atom(arg, &t); /* no check */
    if (t != plus) continue;

    SP_get_arg(i, term, arg);
    switch (SP_term_type(arg)) {
    case SP_TYPE_VARIABLE:
      return NI;
    case SP_TYPE_COMPOUND:
      SP_get_functor(arg, &tname, &tarity);
      SP_put_integer(tmp, (long)tarity);
      SP_cons_list(list, tmp, list);
      SP_put_atom(arg, tname);
      break;
    }
    SP_cons_list(list, arg, list); ++ret;
  }
  return ret;
}
```

# Hasznos lehetőségek SICStus Prolog-ban

- Tetszőleges nagyságú egész számok

  pl.:

```
| ?- fakt(40,F).

F = 815915283247897734345611269596115894272000000000 ?
```

- Globális változók (Blackboard)

  `bb_put(Kulcs, Érték)`

  A `Kulcs` kulcs alatt eltárolja `Érték`-et, az előző értéket, ha van, törölve. (`Kulcs` egy (kis) egész szám vagy atom lehet.)

  `bb_get(Kulcs, Érték)`

  Előhívja `Érték`-be a `Kulcs` értékét.

  `bb_delete(Kulcs, Érték)`

  Előhívja `Érték`-be a `Kulcs` értékét, majd kitörli.

# Hasznos lehetőségek SICStus Prolog-ban *(folytatás)*

- Visszaléptethető módon változtatható kifejezések

  `create_mutable(Adat, ValtKif)`

  `Adat` kezdőértékkel létrehoz egy új változtatható kifejezést, ez lesz `ValtKif`. `Adat` nem lehet üres változó.

  `get_mutable(Adat, ValtKif)`

  `Adat`-ba előveszi `ValtKif` pillanatnyi értékét.

  `update_mutable(Adat, ValtKif)`

  A `ValtKif` változtatható kifejezés új értéke `Adat` lesz. Ez a változtatás visszalépéskor visszacsinálódik. `Adat` nem lehet üres változó.

- Takarító eljárás

  `call_cleanup(Hivas, Tiszito)`

  Meghívja `call(Hivas)`-t és ha az véglegesen befejezte futását, meghívja `Tiszito`-t. Egy eljárás akkor fejezte be véglegesen a futását, ha további alternatívák nélkül sikerült, meghiúsult vagy kivételt dobott.

# Fejlett vezérlési lehetőségek SICStusban: Blokk-deklarációk

- Példa:

```
:- block p(-, ?, -, ?, ?).
```

Jelentése: ha az első és a harmadik argumentum is behelyettesítetlen változó (blokkolási feltétel), akkor a `p` hívás felfüggesztődik.

Ugyanarra az eljárásra több vagylagos feltétel is szerepelhet, pl.

```
:- block p(-, ?), p(?, -).
```

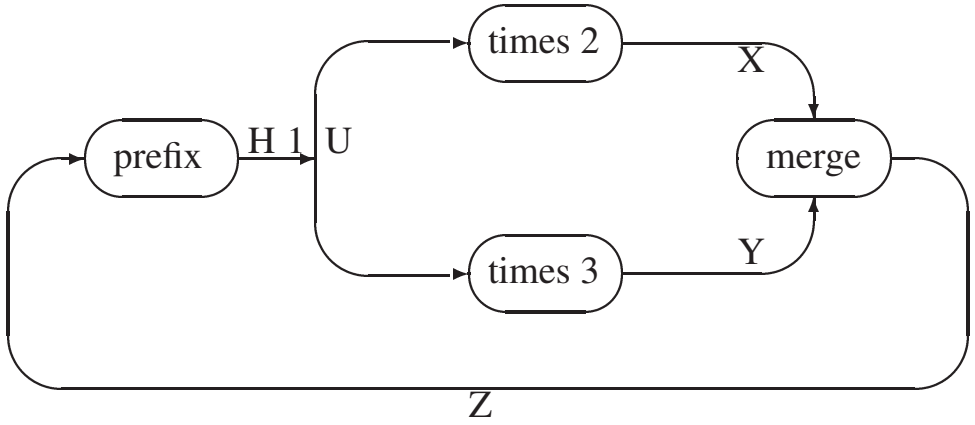
- Végtelen választási pontok kiküszöbölése blokk-deklarációval

```
:- block append(-, ?, -).

append([], L, L).
append([X|L1], L2, [X|L3]) :-
    append(L1, L2, L3).
```

# Blokk-deklarációk *(folytatás)*

- Generál-és-ellenőriz típusú programok gyorsítása
  - általában nem hatékonyak (pl megrajzolja_1), mert túl sok visszalépést használnak
  - korutinszervezéssel a generáló és ellenőrző rész "automatikusan" összefésülhető
  - ehhez az ellenőrző részt kell előre tenni és megfelelően blokkolni
- Korutinszervezésre épülő programok
  - Példa: egyszerűsített Hamming feladat
    - keressük a $2^i * 3^j (i \geq 1, j \geq 1)$ alakú számok közül az első `N` darabot nagyság szerint rendezve.
    - "stream-and-parallelism" közelítésmódot használva korutinszervezéssel egyszerűen lehet megoldani

# Hamming probléma




```
% A H lista az első N, csak a 2 és 3 tényezőkből álló szám.
hamming(N, H) :-
     U = [1|H], times(U, 2, X), times(U, 3, Y),
     merge(X, Y, Z), prefix(N, Z, H).

% times(X, M, Z): A Z lista az X elemeinek M-szerese
:- block times(-, ?, ?).
times([A|X], M, Z) :- B is M*A, Z = [B|U], times(X, M, U).
times([], _, []).
```

# Hamming probléma (folyt.)

```
% merge(X, Y, Z): Z az X és Y összefésülése.
:- block merge(-, ?, ?), merge(?, -, ?).
% Csak akkor fusson, ha az első két argumentum ismert
merge([A|X], [B|Y], V) :-
     A < B, !, V = [A|Z], merge(X, [B|Y], Z).
merge([A|X], [B|Y], V) :-
     B < A, !, V = [B|Z], merge([A|X], Y, Z).
merge([A|X], [A|Y], [A|Z]) :-
     merge(X, Y, Z).
merge([], X, X) :- !.
merge(_, [], []).

% prefix(N, X, Y): Az X lista első N eleme Y.
prefix(0, _, []) :- !.
prefix(N, [A|X], [A|Y]) :-
     N > 0, N1 is N-1, prefix(N1, X, Y).
```

# Korutinszervező eljárások

- `freeze(X, Hivas)`

  `Hivas`t felfüggeszti mindaddig, amig `X` behelyettesítetlen változó.

- `frozen(X, Hivas)`

  Az `X` változó miatt felfüggesztett hívás(oka)t egyesíti `Hivas`-sal.

- `dif(X, Y)`

  `X` és `Y` nem egyesíthető. Mindaddig felfüggesztődik, amig ez el nem dönthető.

- `call_residue(Hivas, Maradék)`

  `Hivas`-t végrehajtja, és ha a sikeres lefutás után maradnak felfüggesztett hívások, akkor azokat visszaadja `Maradékban`. Pl.

```
| ?- call_residue(dif(X, f(Y)), Maradek).
        ==> Maradek = [[X]-(prolog:dif(X,f(Y)))]
| ?- call_residue((dif(X, f(Y)), X=f(Z)), Maradek).
        ==> X = f(Z), Maradek = [[Y,Z]-(prolog:dif(f(Z),f(Y)))]
```

# SICStus könyvtárak

- Könyvtár betöltése

```
:- use_module(library(könyvtárnév)).
```

- A legfontosabb könyvtárak
  - `arrays` Logaritmikus elérési idejű kiterjeszthető tömbök megvalósítását tartalmazza.
  - `assoc` AVL fák segítségével valósítja meg az „asszociációs listák”, azaz véges Prolog kifejezéshalmazokon definiált kiterjeszthető leképezések fogalmát.
  - `atts` tetszőleges attributumokat enged a Prolog változókhoz rendelni, ezeket tárolórekeszként és a Prolog egyesítési mechanizmusának módosítására is engedi használni.
  - `heaps` A bináris kazal (heap) fogalmát valósítja meg, amely főként prioritásos sorok (priority queue) megvalósítására használható.
  - `lists` Biztosítja a listakezelő alapműveleteket.
  - `terms` Különböző kifejezéskezelő eljárásokat tartalmaz.
  - `ordsets` Halmazműveleteket definiál, ahol a halmazokat a Prolog szabványos rendezése szerint (`compare`) rendezett listákkal ábrázolja.
  - `queues` Sorokra (queue, FIFO store) vonatkozó műveleteket definiál.

- `random` Egy véletelenszám-generátort tartalmaz.
- `system` Különféle operációsrendszer-szolgáltatások elérését biztosítja.
- `trees` Az `arrays` könyvtárhoz hasonló, de nem-kiterjeszthető logaritmikus elérési idejű tömbfogalmat valósít meg, bináris fák segítségével (kicsit hatékonyabb mint az `arrays` könyvtár).
- `ugraphs` Irányított és irányítatlan gráffogalmat valósít meg, élcimkék nélkül.
- `wgraphs` Olyan irányított és irányítatlan gráffogalmat valósít meg, ahol minden él egy egészértékű súllyal rendelkezik.
- `sockets` A socket-ek kezelésére szolgáló eljárásokat biztosít.
- `linda/client` és `linda/server` Linda-szerű processzkommunikációs eszközöket ad.
- `bdb` Felhasználó által definiált többszörös indexelést lehetővé tevö, Prolog kifejezések lemezen való tárolására szolgáló adatbázis-rendszer.
- `clpb` Boole-értékekre vonatkozó feltétel-megoldó (constraint solver).
- `clpq` és `clpr` Feltétel-megoldó a Q (racionális számok) ill. R (valós számok) tartományán.
- `clpfd` Véges tartományokra vonatkozó feltétel-megoldó (constraint solver).
- `tcltk` A *Tcl/Tk* nyelv és eszközkészlet elérését biztosítja.
- `gauge` Prolog programok a profilirozására szolgáló, a `tcltk` -n alapuló grafikus interfésszel rendelkező eszköz.

- `charsio` Karaktersorozatból olvasó ill. abba író be- és kiviteli eljárások gyüjteménye.
- `timeout` Lehetőséget ad arra, hogy célok futási idejét korlátozzuk.
- `xref` A nyomkövetés és a program-analízis segítésére használható keresztreferencia készítő program.

# ÚJ IRÁNYZATOK A LOGIKAI PROGRAMOZÁSBAN

# Új irányzatok a logikai programozásban — kitekintés

- Bevezetés a Logikai Programozásba c. jegyzet 6. fejezete:
  - Párhuzamos megvalósítások
  - Az Andorra-I rendszer rövid bemutatása
  - A Mercury nagyhatékonyságú LP megvalósítás
  - CLP (Constraint Logic Programming)
- Az utolsó két témával foglalkozik a **„Nagyhatékonyságú logikai programozás”** c. választható tárgy (általában őszi félévben)
- Rövid izelítőként áttekintjük a korlát-logikai programozás (CLP) témakörét.
- Constraint = megszorítás, kényszer, korlátozás, korlát, ...
- A továbbiakban a „constraint” angol kifejezésre a „korlát” fordítást használjuk

# KORLÁT-LOGIKAI PROGRAMOZÁS – RÖVID ÁTTEKINTÉS

# A korlát-logikai programozás (CLP,Constraint Logic Programming) alapgondolata

- A CLP($\mathcal{X}$) séma

  Prolog + egy valamilyen $\mathcal{X}$ adattartományra és azon értelmezett korlátokra (relációkra) vonatkozó „erős” következtetési mechanizmus.

- Példák az $\mathcal{X}$ tartomány megválasztására
  - $\mathcal{X}$ = Q vagy R (a racionális vagy valós számok)
    korlátok = lineáris egyenlőségek és egyenlőtlenségek
    következtetési mechanizmus = Gauß elimináció és szimplex módszer
  - $\mathcal{X}$ = FD (egész számok Véges Tartománya, angolul FD — Finite Domain)
    korlátok = különféle aritmetikai és kombinatorikus relációk
    következtetési mechanizmus = MI CSP–módszerek (CSP = Korlát-Kielégítési Probléma)
  - $\mathcal{X}$ = B (0 és 1 Boole értékek)
    korlátok = ítéletkalkulusbeli relációk
    következtetési mechanizmus = MI SAT-módszerek (SAT — Boole kielégíthetőség)

# A CLP következtetés alapelvei

- A CLP következtetés
  - közege az ún. korlát-tár, amelyben a korlátok gyűlnek, egyre pontosabban közelítve a megoldást;
  - elemei az ún. primitív korlátok (a megengedett korlátok egy részhalmaza)
  - a korlát-tár mindig konzisztens, ellentmondás esetén visszalépés;
  - visszalépés esetén a korlát-tár is visszaáll a korábbi állapotba
  - a következtetés fajtái:
    - **teljes**, pl. CLP(R) lineáris esetben, CLP(B) — minden korlát bekerül a tárba;
    - **részleges**, pl. CLP(FD) — csak bizonyos egyszerű korlátok mennek a tárba, a többi, nem-primitív korlátok ágensként (démonként) várakoznak arra, hogy:
      a. primitív korláttá váljanak
      b. a tárat egy primitív korláttal bővíthessék (az ún. erősítés)

# A SICStus clp(Q,R) könyvtárak

- Alapelemek
  - Tartomány:
    `clpr`: lebegőpontos számok, `clpq`: racionális számok
  - Függvények:
    `+ - * / min max pow exp` (kétargumentumúak, `pow` ≡ `exp`),
    `+ - abs sin cos tan` (egyargumentumúak).
  - Korlát-relációk: `= =:= < > =< >= =\=` (`=` ≡ `=:=`)
  - Primitív korlátok (a korlát-tár elemei): lineáris kifejezéseket tartalmazó relációk
  - Megoldó algoritmus: lineáris programozási módszerek (Gauss elimináció, szimplex módszer)
- A könyvtár betöltése:
  `use_module(library(clpq))`, vagy `use_module(library(clpr))`
- A fő beépített eljárás
  - `{` *Constraint* `}`, ahol *Constraint* változókból és (egész vagy lebegőpontos) számokból a fenti műveletekkel felépített reláció, vagy ilyen relációknak a `,` operátorral képzett) konjunkciója.

# Példák a SICStus `clpq` könyvtárának használatára

```
| ?- use_module(library(clpq)).

| ?- {X=Y+4, Y=Z-1, Z=2*X-9}.                    % lineáris egyenlet
       X = 6, Y = 2, Z = 3 ?

| ?- {X+Y+9<4*Z, 2*X=Y+2, 2*X+4*Z=36}.          % egyenlőtlenség is lehet
       {X<29/5}, {Y= -2+2*X}, {Z=9-1/2*X} ?

| ?- {(Y+X)*(X+Y)/X = Y*Y/X+100}.               % lineárissá egyszerűsíthető
       {X=100-2*Y} ?

| ?- {(Y+X)*(X+Y) = Y*Y+100*X}.                 % így már nem...
       clpq:{2*(X*Y)-100*X+X^2=0} ?

| ?- {exp(X+Y+1,2) = 3*X*X+Y*Y}.                % nem lineáris...
       clpq:{1+2*X+2*(Y*X)-2*X^2+2*Y=0} ?

| ?- {exp(X+Y+1,2) = 3*X*X+Y*Y}, X=Y.           % így már igen...
       X = -1/4, Y = -1/4 ?

| ?- {2 = exp(8, X)}.                           % nem-lineárisak is megoldhatók
       X = 1/3 ?
```

# A SICStus `clpb` könyvtár

- Alapelemek:
  - **Tartomány:** logikai értékek (`1` és `0`, igaz és hamis)
  - **Függvények** (egyben korlát-relációk):

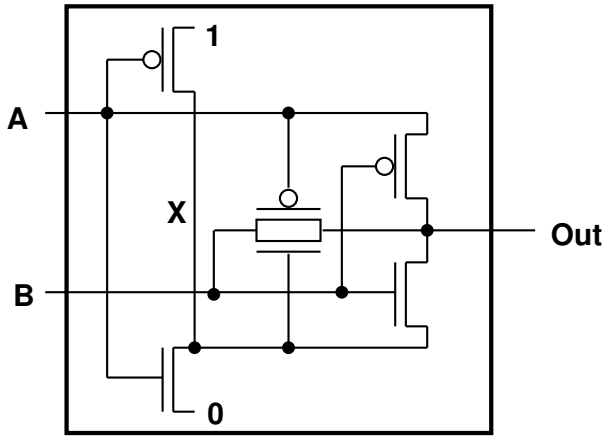
    | | |
    |---|---|
    | `~ P` | `P` hamis (*negáció*). |
    | `P * Q` | `P` és `Q` mindegyike igaz (*konjunkció*). |
    | `P + Q` | `P` és `Q` legalább egyike igaz (*diszjunkció*). |
    | `P # Q` | `P` és `Q` pontosan egyike igaz (*kizáró vagy*). |
    | `P =:= Q` | Ugyanaz mint `~(P # Q)`. |

  - **Constraint-megoldó algoritmus:** Boole-egyesítés.
- A `library(clpb)` könyvtár eljárásai
  - `sat(`*Kifejezés*`)`, ahol *Kifejezés* változókból, a `0 1` konstansokból és atomokból (ún. szimbolikus konstansok) a fenti müveletekkel felépített logikai kifejezés. Hozzáveszi *Kifejezés*t a korlát-tárhoz.
  - `labeling(`*Változók*`)`. Behelyettesíti a *Változók*at `0 1` értekekre, úgy, hogy a tár teljesüljön. Visszalépéskor felsorolja az összes lehetséges értéket.

# Példa a `clpb` könyvtár használatára: tranzisztoros áramkör verifikálása

```
n(D, G, S) :-           % Gate => Drain = Source
        sat( G*D =:= G*S).

p(D, G, S) :-           % ~ Gate => Drain = Source
        sat( ~G*D =:= ~G*S).

xor(A, B, Out) :-
        p(1, A, X),                 n(B, X, Out),
        n(0, A, X),                 p(A, B, Out),
        p(B, A, Out),               n(X, B, Out).
```

```
| ?- n(D, 1, S).                    S = D ?

| ?- n(D, 0, S).                    true ?

| ?- p(D, 0, S).                    S = D ?

| ?- p(D, 1, S).                    true ?

| ?- xor(a, b, X).                  sat(X=:=a#b) ?
```



# A SICStus `clpfd` könyvtár

- A `clpfd` könyvtár alapelemei
  - Tartomány: egészek (negatívak is!)
  - Függvények (aritmetika): `+ - * / ...`
  - Constraint-relációk

    **aritmetikaiak:** `#<, #>, #=<, #>=, #= #\=`

    **halmazműveletek:** *X* `in` *Halmaz*, pl. `X in 1..5`

    **logikai műveletek:** `#/\, #\/, #\` (negáció), `#<=>` (ekvivalencia), ...
  - egyszerű korlátok (korlát tár elemei): *X* `in` *Halmaz*
  - Constraint-megoldó algoritmus:

    **aritmetikaiak:** ún. intervallum-konzisztencia (csak a határokat szűkítik)

    **halmazműveletek:** teljes konzisztencia (ún. tartomány-konzisztencia)
- A tipikus CLP(FD) megoldási folyamat (forrás: CSP = Constraint Satisfaction Problems)
  - a változók tartományának megadása
  - korlátok felvétele
  - címkézés (visszalépéses keresés) — pl. a `labeling(Opciók, Változók)` könyvtári eljárás segítségével.

# Példa a `clpfd` könyvtár használatára: N királynő a sakktáblán

```
% A Qs lista N királynő biztonságos elhelyezését mutatja egy N*N-es sakktáblán:
% a lista i. eleme j ===> az i. királynőt az i. sor j. oszlopába kell helyezni.
queens(N, Qs):-             length(Qs, N), domain(Qs, 1, N), safe(Qs).

% safe(Qs): A Qs királynő-lista biztonságos.
safe([]).
safe([Q|Qs]):-              no_attack(Qs, Q, 1), safe(Qs).

% no_attack(Qs, Q, I): A Qs lista által leírt királynők egyike sem támadja a
% Q oszlopban levő királynőt, feltéve hogy Q és Qs távolsága I.
no_attack([],_,_).
no_attack([X|Xs], Y, I):- no_threat(X, Y, I), J is I+1, no_attack(Xs, Y, J).

% Az X és Y oszlopokban  I sortávolságra levő királynők nem támadják egymást.
no_threat(X, Y, I) :-      Y #\= X, Y #\= X-I, Y #\= X+I.
```

```
| ?- queens(4, Qs).
        Qs = [_A,_B,_C,_D], _A in 1..4, _B in 1..4, _C in 1..4, _D in 1..4 ?
| ?- queens(4, Qs), Qs = [1|_].
        Qs = [1,_A,_B,_C], _A in 3..4, _B in{2}\/{4}, _C in 2..3 ?
| ?- queens(4, Qs), Qs = [1|_], labeling([], Qs).
        no
| ?- queens(4, Qs), Qs = [2|_], labeling([], Qs).
        Qs = [2,4,1,3] ?
```

# Egy példasor: Lovagok és lókötők

- A feladat
  - Egy szigeten minden bennszülött lovag vagy lókötő.
  - A lovagok mindig igazat mondanak.
  - A lókötők mindig hazudnak.
  - Egy vagy több bennszülöttnek saját magukra vonatkozó kijelentése alapján meg kell határozni a bennszülött típusát.
  - Példa: Találkozunk két bennszülöttel Alfréd-dal és Bélával. Alfréd azt mondja: van köztünk lókötő. Milyen típusú Alfréd és Béla.
  - Irodalom: Raymond Smullyan: Mi a címe ennek a könyvnek?, A hölgy és a tigris, Typotex kiadó.
  - Továbbfejlesztés: a szigeten lehetnek normális emberek is, akik néha hazudnak, néha igazat mondanak.

# Lovagok és lókötők – A megoldás elvei

- Készítünk egy egyszerű formális nyelvet a bennszülöttek kijelentéseire, pl. `Alfréd mondja Alfréd = lókötő vagy Béla = lókötő`
- A bennszülöttek nevei (pl. `Alfréd`) Prolog változók, amelyek a `lovag` vagy `lókötő` értéket veszik fel.
- A nyelv egyetlen alap-relációja az =.
- Az összekötő jeleket (`mondja, és, vagy, nem`) Prolog operátornak deklaráljuk.
- Egy egyszerű Prolog programmal definiáljuk a “bennszülött logikát”, azaz a nyelv állításainak igazságértékét.
- A feladat: egy adott mondat esetén megkeresni azokat a változó-behelyettesítéseket, amelyekre a mondat a “bennszülött logika” szerint igaz lesz.

# Lovagok és lókötők: 1. változat (Prolog)

```
:- op(700, fy, nem).      :- op(900, yfx, vagy).
:- op(800, yfx, és).      :- op(950, xfy, mondja).

% Az A bennszülött mondhatja az Áll állítást.
A mondja Áll :- értéke(A mondja Áll, 1).

% értéke(Állítás, Érték): Állítás igazságértéke Érték (1 = igaz, 0 = hamis).
értéke(X = X, 1).
értéke(X = Y, 0) :-              különböző(X, Y).
értéke(lovag mondja M, E) :-     értéke(M, E).
értéke(lókötő mondja M, E) :-    értéke(nem M, E).
értéke(M1 és M2, E) :-           értéke(M1, E1), értéke(M2, E2),  E is E1 /\ E2.
értéke(M1 vagy M2, E) :-         értéke(M1, E1), értéke(M2, E2),  E is E1 \/ E2.
értéke(nem M, E) :-              értéke(M, E1),                   E is 1-E1.

% különböző(A, B): A és B különböző típusú bennszülöttek.
különböző(lovag, lókötő).        különböző(lókötő, lovag).

| ?- Alfréd mondja Alfréd = lókötő vagy Béla = lókötő.
        Béla = lókötő, Alfréd = lovag ? ; no

| ?- A mondja B = C.
        A = lovag, C = B ? ;
        A = lókötő, B = lovag, C = lókötő ? ;
        A = lókötő, B = lókötő, C = lovag ? ; no
```

# Lovagok és lókötők: 2., CLP(B) változat

(A bennszülöttek típusát numerikusan jelöljük: `lovag` →1,`lókötő` →0.)

```
:- use_module(library(clpb)).

:- op(700, fy, nem).      :- op(900, yfx, vagy).
:- op(800, yfx, és).      :- op(950, xfy, mondja).

A mondja Áll :- értéke(A mondja Áll, 1).

% értéke(Állítás, Érték): Az Állítás igazságértéke Érték.
értéke(X = Y, E) :-         sat((X =:= Y) =:= E).
értéke(X mondja M, E) :-    értéke(M, E0), sat((E0 =:= X) =:= E).
értéke(M1 és M2, E) :-      értéke(M1, E1), értéke(M2, E2), sat(E =:= E1*E2).
értéke(M1 vagy M2, E) :-    értéke(M1, E1), értéke(M2, E2), sat(E =:= E1+E2).
értéke(nem M, E) :-         értéke(M, E0), sat(E =:= ~E0).

| ?- Alfréd mondja Alfréd = 0 vagy Béla = 0.
        Béla = 0, Alfréd = 1 ? ; no
| ?- A mondja B mondja C mondja A = C.
        B = 1 ? ; no
| ?- A mondja B = C.
        sat(B=\=C#A) ? ; no
| ?- A mondja B = C, labeling([A,B,C]).
        A = 0, B = 1, C = 0 ? ; A = 0, B = 0, C = 1 ? ;
        A = 1, B = 0, C = 0 ? ; A = 1, B = 1, C = 1 ? ; no
```

# Lovagok és lókötők: 3., CLP(FD) változat

```
:- use_module(library(clpfd)).

:- op(700, fy, nem).      :- op(900, yfx, vagy).
:- op(800, yfx, és).      :- op(950, xfy, mondja).

A mondja Áll :- értéke(A mondja Áll, 1).

% értéke(Állítás, Érték): Az Állítás igazságértéke Érték.
értéke(X = Y, E) :-        X in 0..1, Y in 0..1, E #<=> (X #= Y).
értéke(X mondja M, E) :-   X in 0..1, értéke(M, E0), E #<=> (E0 #= X).
értéke(M1 és M2, E) :-     értéke(M1, E1), értéke(M2, E2), E #<=> E1 #/\ E2.
értéke(M1 vagy M2, E) :-   értéke(M1, E1), értéke(M2, E2), E #<=> E1 #\/ E2.
értéke(nem M, E) :-        értéke(M, E0), E #<=> #\ E0.

| ?- Alfréd mondja Alfréd = 0 vagy Béla = 0.
        Alfréd in 0..1, Béla in 0..1 ? ; no
| ?- Alfréd mondja Alfréd = 0 vagy Béla = 0, labeling([], [Alfréd,Béla]).
        Béla = 0, Alfréd = 1 ? ; no
| ?- A mondja B = C, labeling([], [A,B,C]).
        A = 0, B = 0, C = 1 ? ; A = 0, B = 1, C = 0 ? ;
        A = 1, B = 0, C = 0 ? ; A = 1, B = 1, C = 1 ? ; no
```

# Lovagok, lókötők (és normálisak): 4., CLP(FD) változat

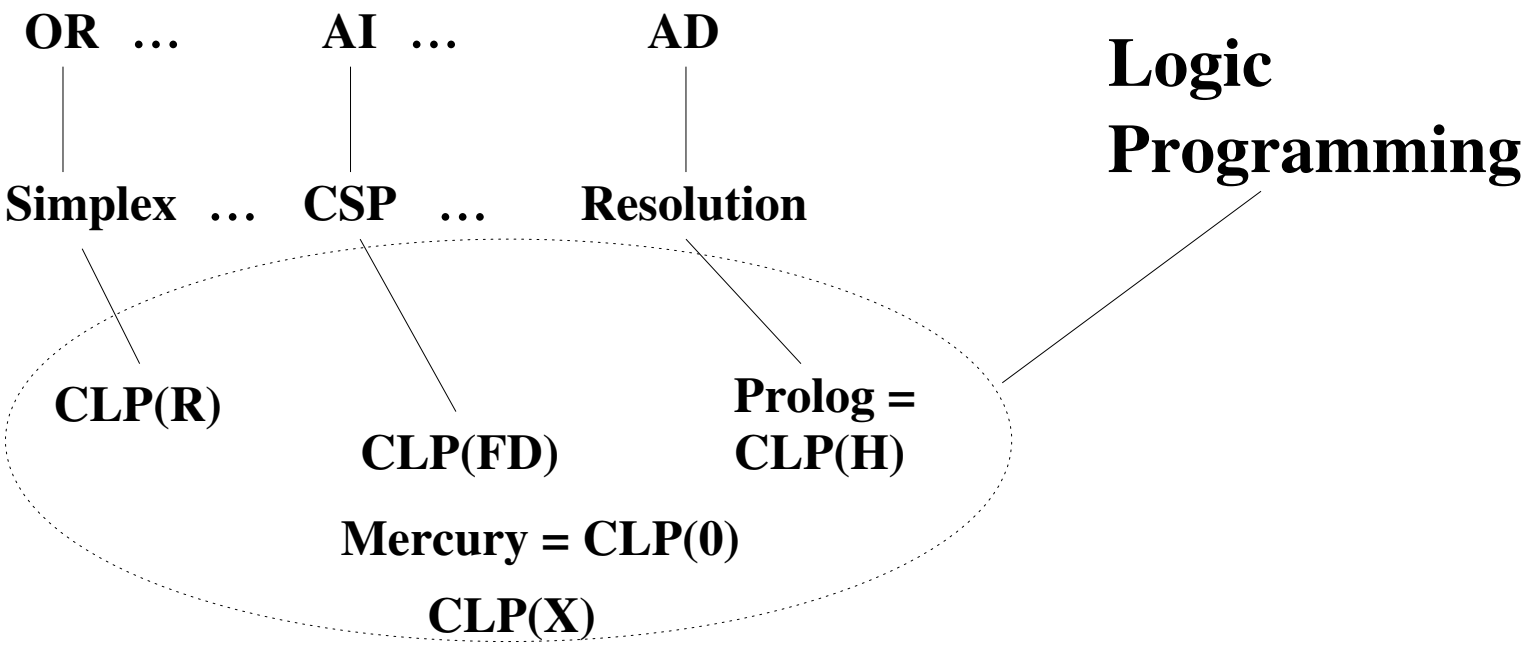
(A bennszülöttek típusa: `normális` →2, `lovag` →1, `lókötő` →0.)

```
:- use_module(library(clpfd)).

:- op(700, fy, nem).      :- op(900, yfx, vagy).
:- op(800, yfx, és).      :- op(950, xfy, mondja).

A mondja Áll :- értéke(A mondja Áll, 1).

% értéke(Állítás, Érték): Az Állítás igazságértéke Érték.
értéke(X = Y, E) :-        X in 0..2, Y in 0..2, E #<=> (X #= Y).
értéke(X mondja M, E) :-   X in 0..2, értéke(M, E0), E #<=> (X #= 2 #\/ E0 #= X).
értéke(M1 és M2, E) :-     értéke(M1, E1), értéke(M2, E2), E #<=> E1 #/\ E2.
értéke(M1 vagy M2, E) :-   értéke(M1, E1), értéke(M2, E2), E #<=> E1 #\/ E2.
értéke(nem M, E) :-        értéke(M, E0), E #<=> #\ E0.

% http://www.math.wayne.edu/~boehm/Probweek2w99sol.htm: We are given three
% people, A, B, C, one of whom is a knight, one a knave, and one a normal
% (but not necessarily in that order). They make the following statements.
%    A: I am normal, B: A is telling the truth, C: I am not normal
% What are A, B, and C?

| ?- A mondja A = 2, B mondja A = 2, C mondja nem C =2, all_different([A,B,C]),
     labeling([], [A,B,C]).
        A = 0, B = 2, C = 1 ? ; no
```

# CLP rendszerek a nagyvilágban

- Néhány implementáció
  - clp(R) — az első CLP(X) rendszer (Monash Univ, Australia, IBM és CMU)
  - CHIP — FD, Q és B (ECRC, Németo., Cosytec, Franciao.); CHARME (Bull); Decision Power (ICL)
  - Prolog III, Prolog IV (PrologIA, Marseille), Q (nem-lineáris is), B, FD, listák, intervallumok
  - ILOG solver (ILOG, Franciao.) — C++ könyvtár: R (nem-lineáris is), FD, halmazok
  - SICStus Prolog (SICS, Svédo.) — R/Q, FD, B
  - GNU Prolog (INRIA, Franciao.) — FD (C-re fordít)
  - Oz (DFKI, Németo.) — korlát alapú elosztott funkcionális nyelv.
- Kommerciális rendszerek (a fentiek között)
  - ILOG, CHIP, Prolog III–IV, SICStus
  - a szakma óriása: ILOG
    - szakterület: CLP + vizualizációs eszközök + szabályalapú eszközök
    - felvásárolta az egyik vezető operációkutatási céget, a CPLEX-et
    - 400 munkatárs 7 országban, 55M USD éves bevétel, NASDAQ-on jegyzett

# Mire használják a CLP rendszereket — néhány példa

- Ipari erőforrás optimalizálás
  - termék- és gépkonfiguráció
  - gyártásütemezés
  - emberi erőforrások ütemezése
  - logisztikai tervezés
- Közlekedés, szállítás
  - repülőtéri allokációs feladatok (beszállókapu, poggyász-szalag stb.)
  - repülő-személyzet járatokhoz rendelése
  - menetrendkészítés
  - forgalomtervezés
- Távközlés, elektronika
  - GSM átjátszók frekvencia-kiosztása
  - lokális mobiltelefon-hálózat tervezése
  - áramkörtervezés és verifikálás

# A CLP mint integrációs paradigma