

## 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.)

### Deklaratív programozás: tudnivalók

---

Követelmények DP-3

#### Honlap, levelezési lista

- Honlap: `<http://www.inf.bme.hu/~dp>`
- Levelezési lista: `<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.
- Ara kötetenként 600-800 Ft, terjedelemtől függően
- Elektronikus változata elérhető a honlapról (ps, pdf)

## KÖVETELMÉNYEK — TUDNIVALÓK

### Deklaratív programozás: tudnivalók (folyt.)

---

Követelmények DP-4

#### Fordító- és értelmezőprogramok

- SICStus Prolog (3.9, licenzzkötetes, aláírás ellenében jelszót adunk)
- Moscow SML (2.0, szabad szoftver)
- Mindkettő telepítve van a `<kempelen.inf.bme.hu>-n`
- Mindkettő letölthető a honlapról (linux, Win95/98/NT)
- Webes gyakorló felület készül (ld. honlap)
- Kézikönyvek HTML-változatban
- 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 (programozni a)!
- Hatékony (időimitt!), 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, leíróható 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: BMIE VIK, 2002. őszi félév

(Követelmények)

**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 feladatot jó megoldásáért 1-1 jutalompont

**Gyakorló feladatok**

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

Deklaratív programozás: BMIE VIK, 2002. őszi félév

(Követelmények)

**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 időkorlátban belüli futás esetén a 10 teszteset mindegyikére 0,5-0,5 pont, összesen max 5 pont, feléve, hogy legalább 4 teszteset sikeres,
  - a dokumentációra, a kód olvashatóságára, kommentezettségére max 2,5 pont.
  - tehát nyelvenként összesen max 7,5 pont szerezhető
- A NHF súlya az osztályzatban: 15% (100 pontból 15)

Deklaratív programozás: BMIE VIK, 2002. őszi félév

(Követelmények)

**Deklaratív programozás: félévközi követelmények (folyt.)****Nagyzárthelyi, pótzárthelyi (NZH, PZH)**

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

Deklaratív programozás: BMIE VIK, 2002. őszi félév

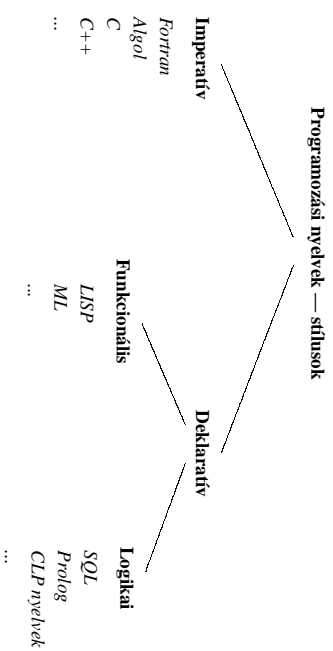
(Követelmények)

## Deklaratív programozás: vizsga

### Vizsga

- Szóbeli, felkészülés frásban
- Prolog, SML: több kisebb feladat, kétszer 35 pontért (programírás, -elemzés)
- Semmilyen jegyzet, segédlet nem használható
- A megatunlandó könyvtári függvények, ill. eljárások listáját előre megadjuk
- Ellenőrzünk a nagy házi feladat és a zárthelyi „hitelességét”
- 40%-os szabály (nyelvenként a max. részpontszám 40%-a kell az eredményességhez)
- Korábbi vizsgakérdések a honlapon találhatóak

## Programozási nyelvek osztályozása



## DEKLARATÍV ÉS IMPERATÍV PROGRAMOZÁS

## 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

Deklaratív programozás. BMÉ VIK, 2002. 654. fejev

(Deklaratív Programozás)

## A nagyszülő feladat — C nyelvű megoldás

Deklaratív és imperatív programozás DP-15

```

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

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

```

Deklaratív programozás. BMÉ VIK, 2002. 654. fejev

(Deklaratív Programozás)

## Példa — családi kapcsolatok

- Adatok

Egy gyerek–szülő kapcsolat, pl.

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

- A feladat:

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

Deklaratív programozás. BMÉ VIK, 2002. 654. fejev

(Deklaratív Programozás)

## A nagyszülő feladat — SML megoldás

Deklaratív és imperatív programozás DP-16

- 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

```

Deklaratív programozás. BMÉ VIK, 2002. 654. fejev

(Deklaratív Programozás)

## 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            Ciyakodó Henrik
Imre            Burgundi Gizella
Imre            Géza
Imre            Sarolt
SQL>
```

Deklaratív programozás: BMÉ VIK, 2002. 6.szá 146év

(Deklaratív Programozás)

## A deklaratív és imperatív megoldások összehasonlítása

- A keresési feladat megoldása
  - C nyelven: ciklussal
  - SQL-ben: beépített adatbázis-kereséssel
  - SML-ben: magasabbrendű függvénybe rejtett rekurzívval
  - 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)

Deklaratív programozás: BMÉ VIK, 2002. 6.szá 146év

(Deklaratív Programozás)

## 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',
        'Ciyakodó 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 = 'Ciyakodó Henrik' ? ;
NSz = 'Burgundi Gizella' ? ;
no

% Kik Géza unokái?
| ?- nagyszuloje(U, 'Géza').
U = 'Imre' ? ;
no
```

Deklaratív programozás: BMÉ VIK, 2002. 6.szá 146év

(Deklaratív Programozás)

## 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 egészset 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).
```

Deklaratív programozás: BMÉ VIK, 2002. 6.szá 146év

(Deklaratív Programozás)

## Bináris fák összegzése

- Egy bináris fa levéösszegének kiszámítása:
- egy csomópont esetén a két részfa levéösszegének 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);
    }
}

% SWI 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.
```

Deklaratív programozás: BME VIK, 2002. ősz 4. félév

(Deklaratív Programozás)

## Bináris fák összegzése — Prolog példafutás

Deklaratív és imperatív programozás DP-23

```
% 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.
%
```

Deklaratív programozás: BME VIK, 2002. ősz 4. félév

(Deklaratív Programozás)

## Bináris fák összegzése — SML példafutás

```
% mosm1
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();
%
```

Deklaratív programozás: BME VIK, 2002. ősz 4. félév

(Deklaratív Programozás)

## A funkcionális programozásról dióhéjban

Deklaratív és imperatív programozás DP-24

- **Alapszeme**
  - 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

Deklaratív programozás: BME VIK, 2002. ősz 4. félév

(Deklaratív Programozás)

## SML — előnyök és hátrányok

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

Deklaratív programozás: BMÉ VIK, 2002. 6.szái félév

(Deklaratív Programozás)

## Prolog — előnyök és hátrányok

Deklaratív és imperatív programozás DP-27

- 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: BMÉ VIK, 2002. 6.szái félév

(Deklaratív Programozás)

## A logikai programozásról dióhéjban

- Alapszeme
  - A program elemei logikai állításoknak felelnek meg, pl.:  $szuloje(U, N) :- szuloje(U, Sz), szuloje(Sz, N)$ . matematikai formája:  $UVVNSz(nagyszuloje(U, N) \leftarrow szuloje(U, Sz) \wedge szuloje(Sz, N))$
  - A program futása: dedukció (tételbizonyítási folyamat)
- A logikai programozás első megvalósítása: a Prolog nyelv
  - A logikai állítások egyszerűek, tekinthetők eljárásdefiniciónak is
  - A tételbizonyítási folyamat értelmezhető mint: mintaillesztéses eljárás hívás + visszalépéses keresés
  - Prolog = RDBMS + rekurzió + adatszerkeztűrák

Deklaratív programozás: BMÉ VIK, 2002. 6.szái félév

(Deklaratív Programozás)

## Deklaratív programozás — miért tanfűnk?

Deklaratív és imperatív programozás DP-28

- Új, magas szintű 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

Deklaratív programozás: BMÉ VIK, 2002. 6.szái félév

(Deklaratív Programozás)

## Egy példa: párbeszéd egy 50 soros Prolog programmal

```

/ ? - párbeszéd.
/ : Magyar legény vagyok én.
Felfogtam.
/ : Ki vagyok én?
Magyar legény
Boldog
/ : 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.

```

Deklaratív programozás: BMÉ VIK, 2002. őszi félév

(Deklaratív Programozás)

Bevezetés LP-31

## 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

Deklaratív programozás: BMÉ VIK, 2002. őszi félév

(Logikai Programozás)

## BEVEZETÉS A LOGIKAI PROGRAMOZÁSBA

### 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 kísérleti Prolog alkalmazás Magyarországon
1981	A japán 5. generációs projekt a logikai programozást választja
1982	A magyar MProlog az egyik első kereskedelmi forgalomban kerüliő 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
	Nagyhatókonyságú Prolog fordítóprogramok
	.....

Bevezetés LP-32

Deklaratív programozás: BMÉ VIK, 2002. őszi félév

(Logikai Programozás)



## 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\\_top.html](http://www.sics.se/ps/sicstus/sicstus_top.html)
  - GNU Prolog: <http://paulliac.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/cimem> belül)
  - Főlap: <0.html>
  - Prolog FAQ: <faq/prolog.faq>
  - Prolog Resource Guide: [faq/prg\\_1.faq](faq/prg_1.faq), [faq/prg\\_2.faq](faq/prg_2.faq)

Deklaratív programozás: BMÉ VIK, 2002. 6561 félév

(Logikai Programozás)

## Magyar nyelvű Prolog irodalom

- Farkas Zsuzsa, Futó Iván, Langer Tamás, Szeredi Péter:**  
Az MProlog programozási nyelv.  
Műszaki Könyvkiadó, 1989
- Márkus Zsuzsa:** Prologban programozni könnyű.  
Novotrade, 1988
- Futó Iván (szerk.):** Mesterséges intelligencia. (9.2. fejezet, Szeredi Péter)  
Aula Kiadó, 1999

Deklaratív programozás: BMÉ VIK, 2002. 6561 félév

(Logikai Programozás)

## Predikátumok, klózok

```

% két klózból álló predikátum definíciója, funktora: sum_tree/2
sum_tree(leaf(Val), Val).
sum_tree(node(Left, Right), S) :-
    sum_tree(Left, S1),
    sum_tree(Right, S2),
    S is S1+S2.

```

```

<Prolog program> ::= <predikátum> ...
<predikátum> ::= <klóz> ...
<klóz> ::= <tényállítás> ,_ |
        <szabály> ,_
<tényállítás> ::= <fej>
<szabály> ::= <fej> :- <törzs>
<törzs> ::= <cél> , ...
<cél> ::= <kifejezés>
<fej> ::= <kifejezés>

```

```

{azonos funktorú}

```

```

          fej \
          |   \
          |   | 1. klóz, tényállítás
          |   |
          |   | % cél
          |   | % cél
          |   | % cél
          /   /

```

## A PROLOG NYELV KÖZELÍTŐ SZINTAXISA

Deklaratív programozás: BMÉ VIK, 2002. 6561 félév

(Logikai Programozás)

## A Prolog szemléletmódjai

- A Prolog nyelv terminológiája többféle szemléletből, értelmezésből származik.

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

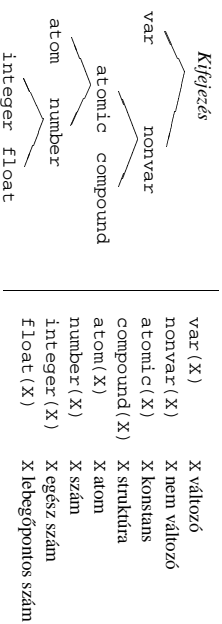
Deklaratív programozás. BMIE VIK, 2002. őszi félév

(Logikai Programozás)

Prolog szinaxis LP-39

## Kifejezések osztályozása

- Kifejezés-osztályok fasztruktúrája — osztályozó beépített eljárások



- Kifejezések funktora (principal functor)

- összetett kifejezés funktora = struktúranév/argumentumszám, pl. sum\_tree/2
- konstans funktora = konstans/0, pl. 'István' / 0
- változónak nincs funktora

Deklaratív programozás. BMIE VIK, 2002. őszi félév

(Logikai Programozás)

## Prolog kifejezések

```

% sun_tree(node(Left, Right), S) % összetett kif., funktora sun_tree/2
% |
% struktúranév | argumentum, változó
% |
% | argumentum, összetett kif.

<kifejezés> ::= <változó> |
              <konstans> |
              <összetett kifejezés>
<konstans> ::= <névkonstans> |
              <számkonstans>
<számkonstans> ::= <egész szám> |
                 <lebegőp. szám>
<összetett kifejezés> ::= <struktúranév> (<argumentum>,...)
<struktúranév> ::= <névkonstans>
<argumentum> ::= <kifejezés>
    
```

Deklaratív programozás. BMIE VIK, 2002. őszi félév

(Logikai Programozás)

Prolog szinaxis LP-40

## 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: !e8 1.e2

<változó> ::= <nagybetű> <alfanum>... |
            _ <alfanum>... |
            ' <névkar>... ' |
            <kisbetű> <alfanum>... |
            <tapadó jel>... | ; [ ] | {}

<névkar> ::= {tetszőleges nem ' és nem \ karakter} |
            \ <escape szekvencia>

<alfanum> ::= <kisbetű> | <nagybetű> | <számjegy> | _
<tapadó jel> ::= + | - | * | / | \ | $ | ^ | < | > | = | ' | ~ | : | . | ? | @ | # | &
<egész szám> ::= {előjeles vagy előjeltelen számjegy-sorozat}
<lebegőp. szám> ::= {belsőjeles tízedes pontot tartalmazó
                    számjegy-sorozat esetleges exponenssel}
    
```

Deklaratív programozás. BMIE VIK, 2002. őszi félév

(Logikai Programozás)

## Prolog adatstruktúrák, típusok

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

```
tree ≡ { leaf(i) | i ∈ integer } ∪ { node(l,r) | l,r ∈ 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 meghüségülési

Deklaratív programozás. BMIE VIK, 2002. 6564 félév

(Logikai Programozás)

Prolog szinaxis LP-43

## Szintaktikus édesfőzser: 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 zárójellezetenül
  - `y`: az adott oldalon állhat azonos prioritású operátor zárójellezetenül
- Tehát: `xFy` jobbról-balra, `yFx` balról-jobbra zárójellez.

Deklaratív programozás. BMIE VIK, 2002. 6564 félév

(Logikai Programozás)

## Megkülönböztetett és nem-megkülönböztetett úniók

- Megkülönböztetett únió:
  - véges sok, különböző funkcionál halmaz úniója
  - például: `:- type tree == {leaf(integer)} ∨ {node(tree, tree)}.`
  - a fenti típusban `leaf/1` és `node/2` a különböző funkcionok.
- 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 adattábrázolást jelent, pl. `node(5,node(3,2))`

Deklaratív programozás. BMIE VIK, 2002. 6564 félév

(Logikai Programozás)

Prolog szinaxis LP-44

## 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
      mod1 << >>
200 xFx **
200 xFy ^
200 Fy -?, \
```

### Egyéb beépített operátorok

```
1150 Fx dynamic multifile
      block meta_predicate
900 Fy spy nospy
550 xFy :
500 yFx #
500 Fx +3
```

1: is/term modban 200 xFx operátor  
2: is/term modban 500 Fx operátor  
3: is/term modban 200 Fy operátor

Deklaratív programozás. BMIE VIK, 2002. 6564 félév

(Logikai Programozás)

## Operátorok felhasználása

- Mire jők az operátorok?
  - aritmetikai eljárások kényelmes írására, pl.  $x \text{ is } (Y+3) \bmod 4$
  - aritmetikai kifejezések szimbolikus feldolgozására (pl. szimbolikus deriválás)
  - klózok leírására (`:- és ', ' is` operátor)
  - klózok átadhatók meta-eljárásoknak, pl. `asserta ( (P(X):-q(X),r(X)) )`
  - eljárásfejek, eljárás hívások olvashatóbbá tételére:
 

```
:- op(800, xfx, [nagyobb_szulofje, szulofje]).
```
  - Gy. `nagyobb_szulofje N :- Gy szulofje Sz, Sz szulofje 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.

Deklaratív programozás. BMÉ VIK, 2002. őszá félév

(Logikai Programozás)

Prolog szinaxis LP-47

## 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 ? ;
```

Deklaratív programozás. BMÉ VIK, 2002. őszá félév

(Logikai Programozás)

## Operátorok — szimbolikus deriválás

- Példa: egy formula természetes módon ábrázolható Prologban, pl.  $x^2y+y+1$ . A formula típusának leírásához nem-megkülönböztetett ünítőt kell használni: `n`

```
% :- type form == atom \ integer \ { form+form } \ \ { form-form } ...
```
- Példa formulakezelésre: szimbolikus derivált előállítás
 

```
% deriv(+Kif, +X, ?D): Kif-nek az X atom szerinti deriváltja D.
deriv(X, X, 1).
deriv(C, -, 0) :-
    number(C).
deriv(A, X, 0) :-
    atom(A), A \= X. % nem egyesíthető
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
```

Deklaratív programozás. BMÉ VIK, 2002. őszá félév

(Logikai Programozás)

## PROLOG SZEMANTIKA

## A Prolog deklaratív szemantikája

- Egy program és egy rá vonatkozó kérdés jelentése a következő:
  - Minden klóz egy logikai állításnak felel meg, pl.:  
 $naegyszuloje(U, N) :- szuloje(U, Sz), szuloje(Sz, N)$ .  
 logikai formája:  
 $\forall U \forall N \forall Sz (naegyszuloje(U, N) \leftarrow szuloje(U, Sz) \wedge szuloje(Sz, N))$
  - A fellet kérdéseknek (célsorozatnak) egy bizonyítandó állítás felel meg, pl.  
 logikai formája:  
 $\exists N (naegyszuloje('Imre', N))$
  - 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 valószínűsíthető meg:
  - az összes következmény nem biztos, hogy előállítható

Deklaratív programozás: BMÉ VIK, 2002. 56. oldal

(Logikai Programozás)

Prolog szemantika LP-51

## Prolog végrehajtási példa

```
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.
% 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)
% 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)
BIP > C=2, A is 3+C, write(A)
BIP > A is 3+2, write(A)
BIP > write(5)
% ===> 5
BIP > []
```

Deklaratív programozás: BMÉ VIK, 2002. 56. oldal

(Logikai Programozás)

## 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 eljárásírói mechanizmus,
  - visszalépéses mélységi keresés.
- A Prolog eljárásíró 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 ("tisztá" Prolog programokra)
  - a Prolog végrehajtás által előállított megoldás biztosan logikai következmény
  - de nem biztos, hogy minden következmény előáll (hiba, végtelen ciklus)

Deklaratív programozás: BMÉ VIK, 2002. 56. oldal

(Logikai Programozás)

Prolog szemantika LP-52

## A végrehajtás alapelemei: egyesítés

- Az eljárásíró és egy klózfej azonos alakra hozása, változók behelyettesítésével
- Példák
  - Bemenő paraméterátadás:
 

```
hívás: naegyszuloje('Imre', Nsz),
fej: naegyszuloje(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
```

Deklaratív programozás: BMÉ VIK, 2002. 56. oldal

(Logikai Programozás)

## Egyesítés: a behelyettesítés fogalma

- A behelyettesítés
  - Egy függvény, amely változókhöz 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  $K$  kifejezésre. pl.  $F(g(z, h), A, X)$   $\sigma = F(g(C, h), A, s(b, B))$
  - Két behelyettesítés kompozíciója (függvénykompozíció):
 
$$\sigma \otimes \theta = \{x \leftarrow x\sigma\theta \mid x \in D(\sigma)\} \cup \{x \leftarrow x\theta \mid x \in D(\theta) \setminus D(\sigma)\}$$
  - $\sigma$  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.

Deklaratív programozás. BMIE VIK, 2002. őszi félév

(Logikai Programozás)

## Egyesítési példák

- $A = \text{sum\_tree}(\text{leaf}(V), V), B = \text{sum\_tree}(\text{leaf}(5), S)$ 
  - (4)  $A$  és  $B$  neve és argumentumszáma megegyezik
    - (a)  $mgu(\text{leaf}(V), \text{leaf}(5)) = \{V \leftarrow 5\} = \sigma_1$
    - (b)  $mgu(V\sigma_1, S) = mgu(5, S) = \{S \leftarrow 5\} = \sigma_2$
  - tehát  $mgu(A, B) = \sigma_1 \otimes \sigma_2 = \{V \leftarrow 5, S \leftarrow 5\}$
- $A = \text{node}(\text{leaf}(X), T), B = \text{node}(T, \text{leaf}(3))$ 
  - (4)  $A$  és  $B$  neve és argumentumszáma megegyezik
    - (a)  $mgu(\text{leaf}(X), T) = \{T \leftarrow \text{leaf}(X)\} = \sigma_1$
    - (b)  $mgu(T\sigma_1, \text{leaf}(3)) = mgu(\text{leaf}(X), \text{leaf}(3)) = \{X \leftarrow 3\} = \sigma_2$
  - tehát  $mgu(A, B) = \sigma_1 \otimes \sigma_2 = \{T \leftarrow \text{leaf}(3), X \leftarrow 3\}$

Deklaratív programozás. BMIE VIK, 2002. őszi félév

(Logikai Programozás)

## 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  $\sigma = \theta$ .
  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  $A_1, \dots, A_N$  ill.  $B_1, \dots, B_N$ , 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  $\sigma = \sigma_1 \otimes \sigma_2 \otimes \sigma_3 \otimes \dots$
  5. Minden más esetben a  $A$  és  $B$  nem egyesíthető.

Deklaratív programozás. BMIE VIK, 2002. őszi félév

(Logikai Programozás)

## 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 ?

Deklaratív programozás. BMIE VIK, 2002. őszi félév

(Logikai Programozás)

## 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: `uni_fy_with_occurs_check/2`
- Kitejésztés (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(...)))) ) ?
| ?- uni_fy_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(...)))) ) ?
```

Deklaratív programozás. BMÉ VIK, 2002. 656. oldal

(Logikai Programozás)

Prolog szemantika LP-59

## 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 klózá, amelynek feje egyesíthető a hívással, és végrehajtjuk a redukciós lépést.
  3. Ha nincs egyesíthető fejtű klóz, vagy a beépített eljárás meghiúsul, akkor visszalépés következik
  4. Egyébként folytatjuk a végrehajtást 1.-től az új célsorozattal.

Deklaratív programozás. BMÉ VIK, 2002. 656. oldal

(Logikai Programozás)

## A végrehajtás alapelvei: redukciós lépés

- Redukciós lépés: egy célsorozat redukálása egy újabb célsorozattá egy klóz segítségével
- A redukciós lépés végrehajtása:
  - A klózt lemszóljuk, 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 klózfeje 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.
    - Sikertelen 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.

Deklaratív programozás. BMÉ VIK, 2002. 656. oldal

(Logikai Programozás)

Prolog szemantika LP-60

## Prolog végrehajtás: visszalépés

### • Visszalépés történetik:

- Ha egy beépített eljárás meghiúsul
- Ha egy felhasználói eljárás 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 klózból ugyanezen eljárás egy későbbi klózába kerül a vezérlés,
  - mély: egy már lefutott eljárás belsőjebe térünk vissza, újabb megoldást kérve.

Deklaratív programozás. BMÉ VIK, 2002. 656. oldal

(Logikai Programozás)



## Sekély visszalépés: példa

- ```
% :- type tree4 == integer \\/ {tree4--tree4}.
sum_tree4(Tree, Sum) :-
    integer(Tree), Sum = Tree.
sum_tree4(Left--Right, Sum) :-
    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 illeszhető
  - a redukciós lépés eredménye: `integer(5--3), S=5--3`
  - az első, beépített hívás megújításul, 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`

Deklaratív programozás. BME VIK, 2002. őszá félév

(Logikai Programozás)

## 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
```

Deklaratív programozás. BME VIK, 2002. őszá félév

(Logikai Programozás)

## 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 megfélelletés: `BOOL p(x) { return q(x) && r(x) || s(x); }`
- Diszjunkció, mint édesifőzzer
  - A fenti példa felírható a ; operátor (diszjunkció) segítségével:
 

```
p(X) :-
    ( q(X), r(Y)
    ; s(X)
    ).
```
  - A diszjunkción alakra hozáshoz szükséges lehet = /2 feltételek bevezetésére, ha a klózfejek nem egyformák.
  - A diszjunkciók (segéd)eljárás bevezetésével mindig kiküszöbölhetőek.

Deklaratív programozás. BME VIK, 2002. őszá félév

(Logikai Programozás)

## Egy összetettebb példa: polinom behelyettesítési értéka

- Formula: számokból és az 'x' névkonstansból '+' és '\*' operátorokkal felépülő kifejezés.
  - % :- type kif == {x} \\/ number \\/ {kif+kif} \\/ {kif\*kif}.
  - értéke(Kif, X, E): A Kif formula értéke E, az x=x behelyettesítéssel.
  - értéke(x, X, E) :-
 

```
E = X.
```
  - értéke(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.
```
  - | ?- értéke((x+1)\*x+x+2\*(x+x+3), 2, E).
 

```
E = 22 ? ;
no
```

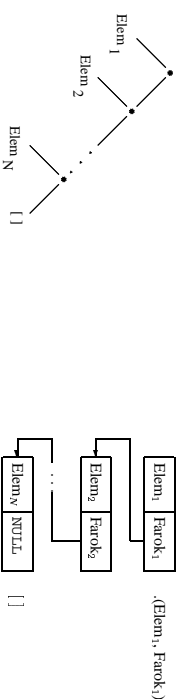
Deklaratív programozás. BME VIK, 2002. őszá félév

(Logikai Programozás)



## A Prolog lista-fogalma

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



Deklaratív programozás. BMÉ VIK, 2002. 656. oldal

(Logikai Programozás)

A Prolog lista LP-67

## Listák jelölése — szintaktikus édesítőszerek

- `[Fej|Farkok] ≡ .(Fej, Farkok)`
  - `[E1|em1, E2|em2, ..., EN|emN | Farkok] ≡ [E1|em1 | [E2|em2, ..., EN|emN | Farkok]]`
  - `[E1|em1, E2|em2, ..., EN|emN] ≡ [E1|em1, E2|em2, ..., EN|emN | []]`
- ```

? - [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|_] ? % nyílt végű
? - L = .(1,[2,3|[1]]).     => L = [1,2,3] ?
? - L = [1,2|. (3,[1])].    => L = [1,2,3] ?
? - [X|[3-Y/X|Y]] = .(A, [A-B,6]). => A=3, B=[6]/3, X=3, Y=[6] ?
    
```

Deklaratív programozás. BMÉ VIK, 2002. 656. oldal

(Logikai Programozás)

## 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).
append([], L2, L) :- L = L2.
append([X|L1], L2, L) :-
    append(L1, L2, L3), L = [X|L3].
    
```
- Helyhiány miatt az alábbi futás-listában az `append` szót `app`-ként rövidítettük.
- ```

> app([1,2,3],[4],A)
(2) > app([2,3],[4],B), A=[1|B]
(2) > app([3],[4],C), B=[2|C], A=[1|B]
(2) > app([],[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, ...)` komplexitáscsúcs futási ideje arányos `L1` hosszával.

Deklaratív programozás. BMÉ VIK, 2002. 656. oldal

(Logikai Programozás)

A Prolog lista LP-68

## 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

```

Deklaratív programozás. BME VIK, 2002. 6.szá. félév

(Logikai Programozás)

A Prolog lista LP-71

## append és revapp — listák gyűjtési iránya

```

● Prolog megvalósítás
append([], L, L).
append([X|L1], L2, [X|L3]) :-
    append(L1, 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 new1 = new link(p->elem);
    *lp = new1; lp = &new1->next;
  }
  *lp = list2;
  return list3;
}

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

```

Deklaratív programozás. BME VIK, 2002. 6.szá. félév

(Logikai Programozás)

## Listák megfordítása

```

● Naïv (négyzetes lépésszámú) megoldás
% rev(L, R): Az R lista az L megfordítása.
rev([], []).
rev([X|L], R) :-
    rev(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 definícióit.
● A könyvtár betöltése:
:- use_module(library(lists)).

```

Deklaratív programozás. BME VIK, 2002. 6.szá. félév

(Logikai Programozás)

A Prolog lista LP-72

## 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 helyértékű jegyet helyezi
a lista elejére (természetes sorrend), és egy másik szám/3 eljárást, amely a legkisebb
helyértékű jeggyel kezdi a listát (fordított sorrend).

● Számjegyek előállítás 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: L1 = L-L0 <-----> az L1 listát L0 elé fűzve kapjuk L-t.
szám(0, _, Jk, Jk).
szám(Sz, Alap, Jk0, Jk) :-
    Sz > 0, Sz1 is Sz//Alap, UtsoEgy is Sz mod Alap,
    szám(Sz1, Alap, [UtsoEgy|Jk0], Jk).

```

Deklaratív programozás. BME VIK, 2002. 6.szá. félév

(Logikai Programozás)

## 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, 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).

% 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).
```

- A kétféle írányú gyűjtés összehasonlítása

Deklaratív programozás. BME VIK, 2002. őszi félév

(Logikai Programozás)

A Prólogo lista LP-75

## 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 elejé fűzve
% kapjuk az L listát.
leaves2(@Int, L0, [Int|L0]).
leaves2(Left-Right, L0, L) :-
    leaves2(Right, L0, L1),
    leaves2(Left, L1, L).
```

Deklaratív programozás. BME VIK, 2002. őszi félév

(Logikai Programozás)

## 2000 tavaszi kis házi feladat — egyszerűsítés

- A fordított jegylistát gyűjtő eljárás egyszerűsíthető

- `fszám/4` minden hívása `fszám(_,_,[],_)` alakú.
- `fszám(Sz, A, [], Jk) => fszám2(Sz, A, Jk)`

- Az egyszerűsített program

```
% Fszám(Szám, Alap, Jegyek): A Szám >= 0 szám Alap > 1 alapú
% számrendszerben jobbról balra vett jegyeinek listája Jegyek.
fszám(0, '-', [0]).
fszám(Sz, Alap, Jk) :-
    Sz > 0, fszám2(Sz, Alap, Jk).

% Fszám2(Szám, Alap, Jk): A Szám >= 0 szám Alap > 1 alapú
% számrendszerben jobbról balra vett jegyeinek listája Jk.
fszám2(0, '-', []).
fszám2(Sz, Alap, [UtsoJegy|Jk]) :-
    Sz > 0, Sz1 is Sz//Alap, UtsoJegy is Sz mod Alap,
    fszám2(Sz1, Alap, Jk).
```

Deklaratív programozás. BME VIK, 2002. őszi félév

(Logikai Programozás)

A Prólogo lista LP-76

## Bináris fa és lista összegezése 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-hoz adva S-t ad.
```

```
sum_list1([], S0, S0).
```

```
sum_list1([X|L], S0, S) :-
```

```
    S1 is S0+X,
```

```
    sum_list1(L, S1, S).
```

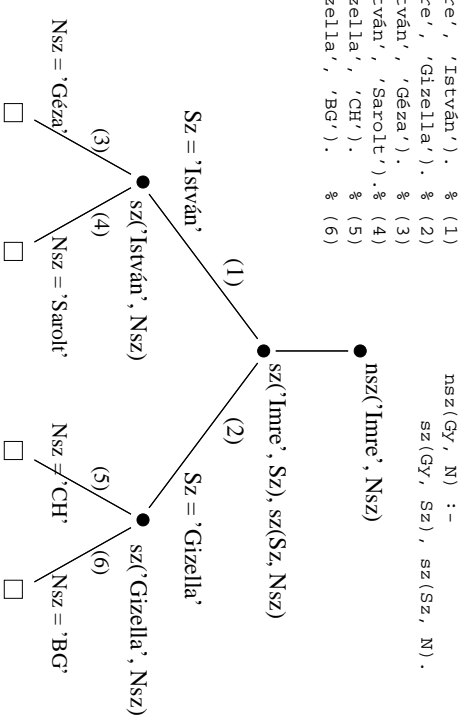
Deklaratív programozás. BME VIK, 2002. őszi félév

(Logikai Programozás)

### Prolog végrehajtási példa

```

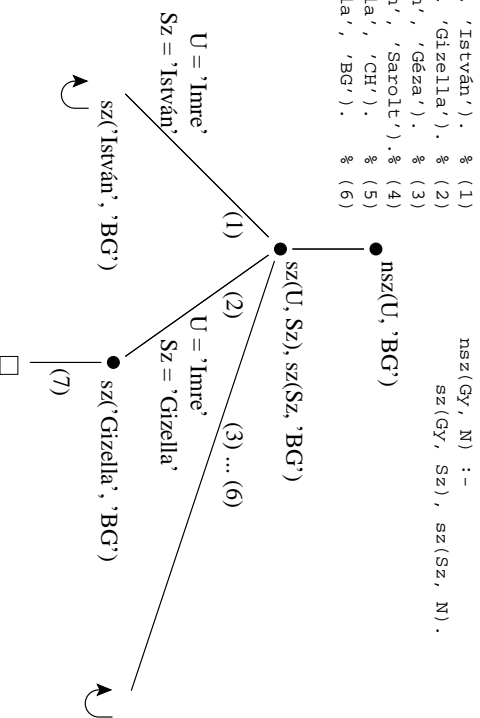
sz('Imre', 'István'). % (1)
sz('Imre', 'Gizella'). % (2)
sz('István', 'Géza'). % (3)
sz('István', 'Sarolt'). % (4)
sz('Gizella', 'CH'). % (5)
sz('Gizella', 'BG'). % (6)
    
```



### 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)
    
```



### A Prolog végrehajtási algoritmus

1. (Kezdeti beállítások:) A verem üres, CS := célsorozat
2. (Beépített eljárások:) Ha CS első célja beépített akkor hajtsuk végre.
  - a. Ha sikertelen => 6. lépés.
  - b. Ha sikeres, elvégezzük a behelyettesítéseket, CS-ből elhagyjuk az első hírvást => 5. lépés.
3. (Klópszámláló kezdtérkezeve:) I = 1.
4. (Redukciós lépés:) CS első hívásához tartozó eljárásdefinicióban N klóz van.
  - a. Ha I > N => 6. lépés.
  - b. Redukciós lépés az I-edik klóz és a CS célsorozat között.
  - c. Ha sikertelen, akkor I := I+1 => 4. lépés.
  - d. Ha I < N (nem utolsó), akkor vereméljük <CS, I>-t.
  - e. CS := a redukciós lépés eredménye
5. (Siker:) Ha CS üres, akkor sikeres vég, egyébként => 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 => 4. lépés.

## Viszálépékes keresés — egy aritmetikai példa

- Példa: „jó” számok keresése
- A feladat: keressük meg azokat a kétfegyű 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(U): U egy decimális számjegy.
dec(0) .
dec(U) :- dec1(U) .

% 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 .
```

Deklaratív programozás. BMIE VIK, 2002. őszai félév

(Logikai Programozás)

## Viszálépékes 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) .

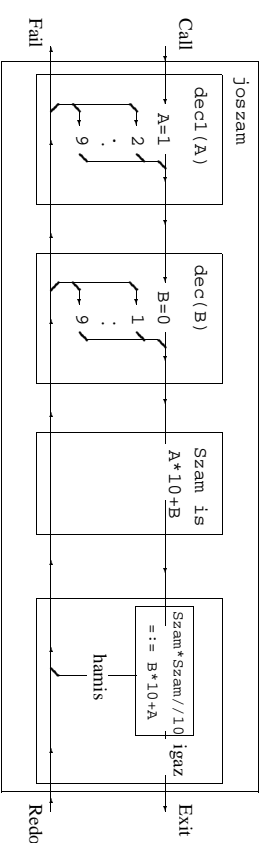
| ?- between(1, 2, _X), between(3, 4, _Y), Z is 10*_X+_Y.
Z = 13 ? ;
Z = 14 ? ;
Z = 23 ? ;
Z = 24 ? ;
no
```

Deklaratív programozás. BMIE VIK, 2002. őszai félév

(Logikai Programozás)

## Prolog végrehajtás — a 4-kapus doboz modell

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



Deklaratív programozás. BMIE VIK, 2002. őszai félév

(Logikai Programozás)

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

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

| ?- member(1, L).

⇒ X = 2 ? ; X = 3 ? ; X = 3 ? ; no

**Lista elemnévé tesz, végtelen választás:**  
| ?- member(1, L).

⇒ L = [1|\_A] ? ; L = [\_A,1|\_B] ? ;

L = [\_A,\_B,1|\_C] ? ; ...

Deklaratív programozás. BMIE VIK, 2002. őszai félév

(Logikai Programozás)

## 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.

## 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 funkora:
    - 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 részalmazát.
    - Futáskor lényegében konstans idő alatt választunk a részalmazok közül.
    - *Fontos:* ha egyetlen a részalmaz, nem hozunk létre választási pontot!

## A végrehajtás optimalizálása: indexelés

1. *Kezdeti beállítások:* A verem üres, CS := célsorozat
2. *Beepített eljárások:* Ha CS első célja beepített akkor hajtsuk végre. (...)
3. *Klózstímláló kezdtérekészve:* 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 ⇒ 6. lépés.
  - b. Redukciós lépés az *indexelési* lista I-edik klóza és a CS célsorozat között.
  - c. Ha sikertelen, akkor I := I+1 ⇒ 4. lépés.
  - d. Ha I < N (nem utolsó), akkor venniük <CS, I>-t.
  - e. CS := a redukciós lépés eredménye
5. *Siker:* Ha CS üres, akkor sikeres vég. egyébként ⇒ 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 ⇒ 4. lépés.

## 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 funkora 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 letűrás előtt megszünteti.
    - p(s(0), Y) választási pontot hagy a letűrásakor.

## Tömrő és minta-kifejezések, lista-miniták, nyílt végű listák

- Tömrő (ground) kifejezés: változói nem tartalmazó kifejezés
- Minta: egy általában nem nem tömrő kifejezés, mindazon kifejezéseket „képviselet”, amelyek belőle változó-behelyettesítéssel előállhatnak
- Lista-minta: listát (is) képviseelő minta.
- Nyílt végű lista: olyan lista-minta, amely bármilyen hosszú listát is képviselet.
- Zárt végű lista: olyan lista(-minta), amely egyféle hosszú listát képviselet.

Zárt v.	Milyen listákat képvisel	Nyílt v.	Milyen listákat képvisel
[X]	egyelemű	X	teszt/ölleges
[X, Y]	kételemű	[X Y]	nem üres (legalább 1 elemű)
[X, X]	két egyforma elemből álló	[X, Y Z]	legalább 2 elemű
[X, 1, Y]	3 elemből áll, 2. eleme 1	[a, b Z]	legalább 2 elemű, elemei: a, b, ...

„Biztonságos” a futás, azaz végleges 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.

Deklaratív programozás. BMÉ VIK, 2002. 6564 félév

(Logikai Programozás)

## Variációk appendre 1. — Három lista összerűzése

Visszalpétes keresés Problémán LP-91

```
% 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,...,1001],[1,2,3],[1], L)` 103 helyett 203 lépés!
- Szűszedésre nem alkalmas — végtelen választási pontot hoz létre (véges a keresési tér, ha az 1. és 3. argumentum legalább egyike zárt végű lista.)

Szűszedésre is alkalmas, hatékony változat

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

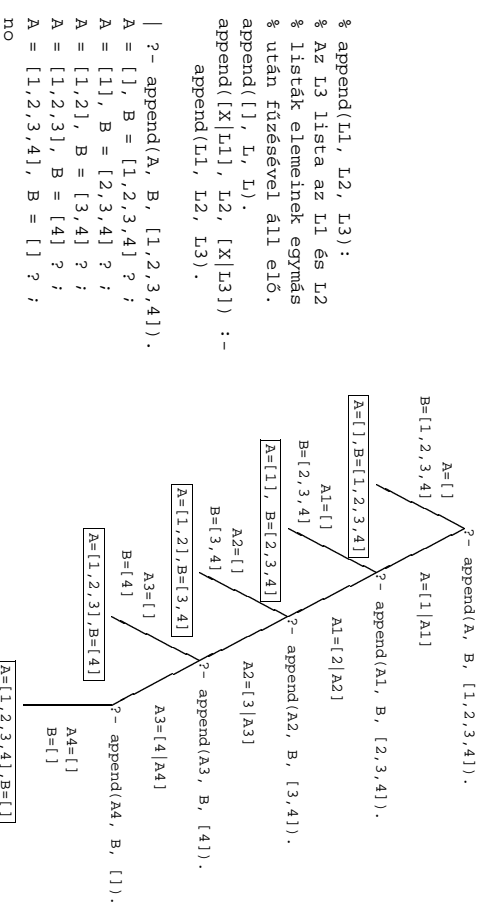
Az első `append/3` hívás nyílt végű listát állít elő:

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

Deklaratív programozás. BMÉ VIK, 2002. 6564 félév

(Logikai Programozás)

## Listák szétbontása az append/3 segítségével



Deklaratív programozás. BMÉ VIK, 2002. 6564 félév

(Logikai Programozás)

## Variációk appendre 2. — lista folytonos része

Visszalpétes keresés Problémán LP-92

```
% 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 = [0,1,2,3,4,...,10]`, `L2 = [0,1,2,3,4,10]`  
x100000
- `csublist(L2, L123)`: 570 msec
- `check_csublist(L2, L123)`: 430 msec

Deklaratív programozás. BMÉ VIK, 2002. 6564 félév

(Logikai Programozás)

## Mintakeresés append /3-mal

- Párban előforduló elemek

```
% párban(Lista, Elem): A Lista számlistájának 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 részlista 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
```

Deklaratív programozás. BMÉ VIK, 2002. 6561 félév

(Logikai Programozás)

Legalapvetőbb beépített eljárások LP-95

## Aritmetikai beépített eljárások

- `X is Kif1, Kif2`: A `Kif1` aritmetikai kifejezés értékét egyesít `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 (`=` :=  $\Rightarrow$  egyenlő, `\=` :=  $\Rightarrow$  nem-egyenlő).

- Ha `Kif1, Kif2` valamelyike nem aritmetikai kifejezés  $\Rightarrow$  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
```

Deklaratív programozás. BMÉ VIK, 2002. 6561 félév

(Logikai Programozás)

## LEGALAPVETŐBB BEÉPÍTETT ELJÁRÁSOK

### Programfejlesztési beépített eljárások

- `consult(File)` vagy `consult(File)`: A `File` állományban levő programot beolvassa és értelmezendő alakban eltárolja. (`File = user`  $\Rightarrow$  terminálról olvas.)
- `listing` vagy `listing(Predikátum)`: Az értelmezendő alakban eltárolt összes ill. adott nevű predikátumokat ki listázza.

- `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 CFT 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.
>
```

Legalapvetőbb beépített eljárások LP-96

### Programfejlesztési beépített eljárások

Deklaratív programozás. BMÉ VIK, 2002. 6561 félév

(Logikai Programozás)



## 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`, `false`: 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
```

Deklaratív programozás. BMÉ VIK, 2002. 656. oldal

(Logikai Programozás)

Nagyobb példák — negáció, feltételes szerkezet LP-99

## 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 járatral!

```
& 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 járatral.
útszakasz(Kezet, Cél, H) :-
(   járat(Kezet, Cél, H)
;   járat(Cél, Kezet, H)
).

```

Deklaratív programozás. BMÉ VIK, 2002. 656. oldal

(Logikai Programozás)

## NAGYOBB PÉLDÁK — NEGÁCIÓ, FELTÉTELES SZERKEZET

Nagyobb példák — negáció, feltételes szerkezet LP-100

## 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, 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
| ?-
```

Deklaratív programozás. BMÉ VIK, 2002. 656. oldal

(Logikai Programozás)

## 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.
útvonal_2(0, Hová, Hová, Kizártak, Út, 0) :-
    reverse(Kizártak, Út).
útvonal_2(N, Honnan, Hová, Kizártak, Út, H) :-
    N > 0, NI is N-1,
    útszakasz(Honnan, Közben, H1),
    \+ member(Közben, Kizártak),
    útvonal_2(NI, 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

```

Deklaratív programozás. BME VIK, 2002. őszi félév

(Logikai Programozás)

## Súlyozott gráf ábrázolása listával

Nagyobb példák — negáció, feltételes szerkezet

LP-103

- 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)]).

```

Deklaratív programozás. BME VIK, 2002. őszi félév

(Logikai Programozás)

## A meghívásúlasos negáció (NF — Negation by Failure)

- $A \setminus H$  hívás beépített meta-eljárás (vö.  $\setminus$  — nem bizonyítható)
    - végrehajtja a hívás hívást,
    - ha hívás sikeresen fut le, akkor meghívásul,
    - egyébként (behelyettesítés nélküli) sikerül.
    - $\setminus + H$  jelentése:  $\neg \exists X(H)$ , ahol  $X$  a  $H$ -ban a hívás pillanatában behelyettesíthető 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 ?

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

```

Deklaratív programozás. BME VIK, 2002. őszi félév

(Logikai Programozás)

## Ismétlődésmentes útvonal keresése listával ábrázolt gráfban

Nagyobb példák — negáció, feltételes szerkezet

LP-104

```

:- 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], 0).
útvonal_3(N, Gráf, Honnan, Hová, [Honnan|Út], H) :-
    N > 0, NI is N-1,
    select(Él, Gráf, Gráf1),
    él_végpontok_hossz(Él, Honnan, Közben, H1),
    útvonal_3(NI, 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

```

Deklaratív programozás. BME VIK, 2002. őszi félév

(Logikai Programozás)

## Egyítható 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 egyíthatója E.
egyhat(x, 1).
egyhat(K1F, E) :-
    number(K1F), E = 0.
egyhat(K1+K2, E) :-
    egyhat(K1, E1),
    egyhat(K2, E2),
    E is E1+E2.
egyhat(K1*K2, E) :-
    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
```

Deklaratív programozás. BMÉ VIK, 2002. 6561 félév

(Logikai Programozás)

Nagyobb példák — negáció, feltételes szerkezet IP-107

## 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
    ),
    (....).
```

Deklaratív programozás. BMÉ VIK, 2002. 6561 félév

(Logikai Programozás)

## Egyítható 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
    ).
```

Deklaratív programozás. BMÉ VIK, 2002. 6561 félév

(Logikai Programozás)

Nagyobb példák — negáció, feltételes szerkezet IP-108

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

- Az egyébként rész elhagyható, alapértelmezése: fail.

Deklaratív programozás. BMÉ VIK, 2002. 6561 félév

(Logikai Programozás)

## Feltételes kifejezés — példák

```

● Faktoriális
% fakt(+N, ?F): NI = F.
fakt(N, F) :-
  ( N = 0 -> F = 1 % N = 0, F = 1
  ; N > 0, NI is N-1, fakt(NI, F1), F is N*F1
  ).

```

● Jelentése azonos a sima diszjunkciós alakkal (lásd komment), de amál 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
  ).

```

Deklaratív programozás. BME VIK, 2002. őszá félév

(Logikai Programozás)

Nagyobb példák — negáció, feltételes szerkezet LP-111

## 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-kei 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
% egyikeivel 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).

```

Deklaratív programozás. BME VIK, 2002. őszá félév

(Logikai Programozás)

## Feltételes kifejezés és negáció

●  $A \setminus +$  Felt negáció kiváltható a ( Felt -> Fail ; true ) feltételes kifejezéssel.

● Példa: ellenőrizzik, 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
  ).

```

●  $A \setminus =$  beépített eljárás jelentése: az argumentumok nem egyesíthetők:  
 $X \setminus = Y :- - \setminus + X = Y.$

● A nem-eleme példa-eljárás rekurzív megvalósítása:

```

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

```

Deklaratív programozás. BME VIK, 2002. őszá félév

(Logikai Programozás)

## A PROLOG SZINTAXIS

## A Prolog szintaxis összefoglalása

- A Prolog szintaxis alapelvei
  - Minden programlelen kifejezés!
  - A szükséges összekötő jelek (', ', '!', ':-', '-->): szabványos operátorok.
  - A beolvasott kifejezést funkcionál alapján osztályozzuk:
    - *kérdés*:  $? - C\acute{E}1$ .
    - *CÉ1t* lefuttatja, és a változó-behelyettesítéseket kiírja (ez az alapértelmezés az ún. top-level interaktív felületen).
    - *paramcs*:  $:- C\acute{E}1$ .
    - A *CÉ1t* csendben lefuttatja. Pl. deklaráció (operátor, ...) elhelyezésére.
    - *szabály*:  $F\acute{e}j :- T\acute{o}rzs$ .
    - A szabályt felveszi a programba.
    - *nyelvtani szabály*:  $F\acute{e}j \rightarrow T\acute{o}rzs$ .
    - Prolog szabálytálá alakítja és felveszi (lásd a DCG nyelvtan).
    - *tényállitás*: *Minden egyéb kifejezés*.
    - Üres törzst szabályként felveszi a programba.

Deklaratív programozás. BMIE VIK, 2002. őszi félév

(Logikai Programozás)

A Prolog szintaxis LP-115

## Szintaktikus édesítőszerek — összefoglalás, gyakorlati tanácsok

- Operátoros kifejezések alapstruktúra alakra hozása
  - Zárójeljeztük be a kifejezést, az operátorok prioritása és fájája alapján, például  $-a+b*2 \Rightarrow ((-a)+(b*2))$ .
  - Hozzuk az operátoros kifejezéseket alapstruktúra alakra:
    - $(A \text{ Inf } B) \Rightarrow \text{Inf}(A, B)$ ,  $(\text{Pref } A) \Rightarrow \text{Pref}(A)$ ,  $(A \text{ Postf}) \Rightarrow \text{Postf}(A)$
    - Példa:  $((-a)+(b*2)) \Rightarrow -(a) + *(b, 2) \Rightarrow +(-(a), *(b, 2))$ .
  - Trükkös esetek:
    - A vesszőtől névként idezni kell: pl.  $(pp, (qq,rr)) \Rightarrow ', '(pp, '(qq,rr))$ .
    - $- Szám \Rightarrow$  negatív számkonstans, de  $- Egy\acute{e}b \Rightarrow$  prefix alak.
    - Példa.  $-1+2 \Rightarrow +(-1, 2)$ , de  $-a+b \Rightarrow +(-(a), b)$ .
    - $N\acute{e}v(\dots) \Rightarrow$  struktúrákifejezés;
    - $N\acute{e}v(\dots) \Rightarrow$  prefix operátoros kifejezés. Példák:
      - $-(1, 2) \Rightarrow -(1, 2)$  (változatlan), de
      - $-(1, 2) \Rightarrow -(', '(1, 2))$ .

Deklaratív programozás. BMIE VIK, 2002. őszi félév

(Logikai Programozás)

## A Prolog nyelv-változatok

- A SICStus rendszer két üzemmódja
  - *iso*: Az ISO Prolog szabványnak megfelelő.
  - *si*: *cttus*: Korábbi változatokkal kompatibilis.
  - Állítás: `set_prolog_flag(Language, Mod)`.
  - Különbségek:
    - szintaxis-részletek, pl. a `0x1ff` szám-alak csak ISO módban,
    - beírtott eljárások viselkedésének kisebb eltérései.
    - az eddig ismertett eljáráások hatása lényegében nem változik.

Deklaratív programozás. BMIE VIK, 2002. őszi félév

(Logikai Programozás)

A Prolog szintaxis LP-116

## Szintaktikus édesítőszerek — listák, egyebek

- Listák alapstruktúra alakra hozása
  - Farok-megadás betoldása.
    - $[1, 2] \Rightarrow [1, 2|[ ]]$ .  $[[X|Y]|[ ]] \Rightarrow [[X|Y]|[ ]]$
    - Vessző (ismétel) kiküszöbölése  $[Elem1, Elem2, \dots] \Rightarrow [Elem1|[Elem2, \dots]]$ .
    - $[1, 2|[ ]] \Rightarrow [1|[2|[ ]]]$
    - $[1, 2, 3|[ ]] \Rightarrow [1|[2, 3|[ ]]] \Rightarrow [1|[2|[3|[ ]]]]$
    - Struktúrákifejezéssé alakítás:  $[Fej|Farok] \Rightarrow \cdot(Fej, Farok)$ .
    - $[1|[2|[ ]]] \Rightarrow \cdot(1, \cdot(2, [ ]))$ ,  $[[X|Y]|[ ]] \Rightarrow \cdot((X, Y), [ ])$
  - Egyéb szintaktikus édesítőszerek:
    - Karakterkód-jelölés: `0'Kar`.
    - $0'a \Rightarrow 97$ ,  $0'b \Rightarrow 98$ ,  $0'c \Rightarrow 99$ ,  $0'd \Rightarrow 100$ ,  $0'e \Rightarrow 101$
    - Füzér (string): `"xyz..."`  $\Rightarrow$  az `xyz...` karakterek kódját tartalmazó lista
    - `"abc" \Rightarrow [97, 98, 99]`, `" "`  $\Rightarrow [ ]$ , `"e" \Rightarrow [101]`
    - Kaposos zárójeljezés: `{Kif}`  $\Rightarrow \{ \{Kif\}$  (egy `{}` név, egy argumentumú struktúra — `a {}` jelpár egy önálló lexikai elem, egy névkonstans).
    - Bináris, hexa, s1b alak (csak *iso* módban), pl. `0b101010`, `0x1a`.

Deklaratív programozás. BMIE VIK, 2002. őszi félév

(Logikai Programozás)

## 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 }

Deklaratív programozás. BMÉ VIK, 2002. 6564 félév

(Logikai Programozás)

A Prolog szintaxis LP-119

## 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 ⟩

Deklaratív programozás. BMÉ VIK, 2002. 6564 félév

(Logikai Programozás)

## 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 ⟩ | ⟨ füzét ⟩
  - | ⟨ név ⟩ | ⟨ szám ⟩ | ⟨ változó ⟩

Deklaratív programozás. BMÉ VIK, 2002. 6564 félév

(Logikai Programozás)

A Prolog szintaxis LP-120

## 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 ( füzét ) " jelek közé zárt karakter sorozat, 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 ]

Deklaratív programozás. BMÉ VIK, 2002. 6564 félév

(Logikai Programozás)

## A Prolog lexikai elemei I. (ismétlés)

- (név)
  - kisbetűvel kezdődő alfanumerikus jelsorozat (ebben megengetve kis- és nagybetűi, számjegyeket és aláhúzásjelet);
  - egy vagy több ún. speciális jelből (+-\* / \ \$ ^ <> = ' ~ : . ? @ # & ) álló jelsorozat;
    - az őnmagában 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.

Deklaratív programozás. BMÉ VIK, 2002. 656. oldal

(Logikai Programozás)

A Prolog szinaxis LP-123

## Megjegyzések és formázó-karakterek

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

Deklaratív programozás. BMÉ VIK, 2002. 656. oldal

(Logikai Programozás)

## 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 i so módban)
  - karakterkód-konstans 0' c alakban, ahol c egyetlen karakter
- (lebegőpontos szám)
  - mindenképpen tartalmaz tízedespontot
  - mindkét oldalán legalább egy (decimális) számjeggyel
  - e vagy E betűvel jelzett esetleges exponens

Deklaratív programozás. BMÉ VIK, 2002. 656. oldal

(Logikai Programozás)

## TÍPUSOK PROLOGBAN

## Tipusok leírása Prologban

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

```
{ str(T1, ..., Tn) } ≡ { str(e1, ..., en) | e1 ∈ T1, ..., en ∈ Tn }, n ≥ 0
```

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

```
{szemely(atom,atom,integer)} \ {atom-atom} \ atom
```
- Egy típusleírás elnevezhető (kommentben): `% :- type tnév == tLeírás.`

```
% :- type t1 == {atom-atom} \ atom.,
% :- type ember == {ember-atom} \ {semmi}.
```
- Megkülönböztetett únó: csupa különböző funktori összetett típus únóija. Egyszerűsített jelölés:
 

```
:- type T == { S1 } \ / .. \ / { Sn }. ⇒ :- type T ---> S1 ; ... ; Sn.
% :- type ember ---> ember-atom; semmi.
% :- type egészlista ---> []; [integer|egészlista].
```

Deklaratív programozás. BMIE VIK, 2002. 6564 félév

(Logikai Programozás)

Tipusok Prologban IP-127

## Predikátum-deklarációk

- Predikátumtípus-deklaráció
 

```
:- pred {eljárásnév}({típusazonosító},...)
```
- Példák:
 

```
:- pred member(T, list(T)).
:- pred append(list(T), list(T), list(T)).
```
- Predikátumtípus-deklaráció (Nem kötelező, több is megadható)
 

```
:- mode {eljárásnév}({módaazonosító},...) ahol {módaazonosító} ::= in | out.
```
- Példák:
 

```
:- mode append(in, in, in). % ellenőrzésére
:- mode append(in, in, out). % két lista összerűzésére
:- mode append(out, out, in). % egy lista szétűzésére
```
- Vegyes típus- és móddeklaráció
 

```
:- pred {eljárásnév}({típusazonosító}:::{módaazonosító},...)
```
- Példa:
 

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

Deklaratív programozás. BMIE VIK, 2002. 6564 félév

(Logikai Programozás)

## Tipusok 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) % KeyT és ValueT típusú % == list(pair(KeyT, ValueT)). % 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. únó}.
<megkülönb. únó> ::= <konstruktor> ; ...
<konstruktor> ::= <névkonstans> | (<struktúránév>)( <típusleírás> ), ... )
<típusleírás> ::= <típusnév> | (<típusváltozó> | <típusleírás> ) \ / <típusleírás> | { <típusnév> } ( <típusleírás> ), ... )
<típusazonosító> ::= <típusnév> | <típusnév> ( <típusváltozó> ), ... )
```

Deklaratív programozás. BMIE VIK, 2002. 6564 félév

(Logikai Programozás)

Tipusok Prologban IP-128

## 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

Deklaratív programozás. BMIE VIK, 2002. 6564 félév

(Logikai Programozás)



## SML-Prolog áttekzés: 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, ismét érték  
 minta: csak fordítási időben értelmes  
 egyszerű minták:  $x::x::x$  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 ismerten érték  
 minta: teljes jogú adastruktúra  
 összetett minták, pl.  $[X, X|Xs]$   
 kétirányú mintaillesztés  
 többértelmű klózválasztás  
 több eredmény (nemdeterminizmus)  
 többirányú használat  
 (pl. összetekő és szétszedő append)  
 struktúra (rekord)  
 konjunktív, segéd-változóval

Deklaratív programozás. BMÉ VIK, 2002. őszá félév

(Logikai Programozás)

SML-Prolog áttekzés LP-131

## SML-Prolog áttekzés: 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  
 rekurzív  
 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.
```

típusatlan nyelv  
 rekurzív, ritkábban magasabbrendű predikátum  
 visszalépéses ciklus  
 (pl. két lista közös eleme)  
 meghívás, kivétel

SML-Prolog áttekzés LP-132

## Összefoglalás: Prolog programok szemantikája

- Prolog program jelentése = milyen válaszokat (behelyettesítéseket) kapunk egy cél futtatásakor.
- Procedurális szemantika — az ismeretett végrehajtási, egyesítési algoritmus.
- Deklaratív szemantika:
  - program: logikai állítások (klózek, 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. :- ( )

Deklaratív programozás. BMÉ VIK, 2002. őszá félév

(Logikai Programozás)

Deklaratív programozás. BMÉ VIK, 2002. őszá félév

(Logikai Programozás)

## Ismétlés: A Prolog végrehajtási mechanizmus, dióhéjban

- (Kezdet:) Ha célsorozat üres → sikeres lefutás.
- (Folytatás:) Keresünk az **első** céllal egyesíthető klózfejet (a klózból friss másolatot képezve, felültölti 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 klóztörzsen 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 tetejétől), és új egyesíthető fejű klóz keresésével folytatjuk a (Folytatás)-nál.

Deklaratív programozás. BMÉ VIK, 2002. 6561 félév

(Logikai Programozás)

SML-Prolog áttekzés LP-133

## 4. fejezet: Prolog programozási módszerek

- Az előző előadás-blokk (jegyzetbeli 3. fejezet) célja volt:
  - a Prolog nyelv alapjainak bemutatása.
  - a logikailag „tiszta” résznyelvre koncentrálnia.
- 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.

Deklaratív programozás. BMÉ VIK, 2002. 6561 félév

(Logikai Programozás)

## 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 meghúsnul.

Deklaratív programozás. BMÉ VIK, 2002. 6561 félév

(Logikai Programozás)

SML-Prolog áttekzés LP-136

## Prolog programozási módszerek: tartalomjegyzék

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

Deklaratív programozás. BMÉ VIK, 2002. 6561 félév

(Logikai Programozás)

## 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:
```

  - Végrehajtuk 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:
  - Diszjunktiónak 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 diszjunktív másik ágával folytatjuk.)

Deklaratív programozás. BMÉ VIK, 2002. 564. oldal

(Logikai Programozás)

## A KERESÉSI TÉR SZŰKÍTÉSE

A keresési tér szűkítése LP-139

### Feltételes szerkezet: választási pontok a feltételben

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

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

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

Deklaratív programozás. BMÉ VIK, 2002. 564. oldal

(Logikai Programozás)

### A vágó eljárás

A keresési tér szűkítése LP-140

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

```
első_poz_elem([_|_], EP) :- !, EP > 0, !.
első_poz_elem([X|_], 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 „átrahatlan”
  - (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)

Deklaratív programozás. BMÉ VIK, 2002. 564. oldal

(Logikai Programozás)

## Példák a vágó eljárás használatára

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

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

% pozitívvalak(+L, -P): P az L pozitív elemekből áll.
pozitívvalak([], []).
pozitívvalak([_E|_], [E|Pk]) :-
    E > 0, !,
    pozitívvalak(Ek, Pk).
pozitívvalak([_E|_], Pk) :-
    /* \+_E > 0, */ pozitívvalak(Ek, Pk).
```

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

Deklaratív programozás. BMÉ VIK, 2002. őszi félév

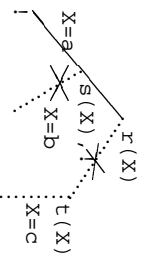
(Logikai Programozás)

## 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.
% a vágót tartalmazó példa futása
:- r(X), write(X), fail.
```



A keresési tér szűkítése LP-143

Deklaratív programozás. BMÉ VIK, 2002. őszi félév

(Logikai Programozás)

## A vágó definíciója

### Segédfogalom

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

### A vágó végrehajtása:

- mindig sikerül: és a végrehajtás adott állapottól visszatele egészen a szülő céljig, 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)
```

Deklaratív programozás. BMÉ VIK, 2002. őszi félév

(Logikai Programozás)

## A diszjunktív feltételes szerkezet megvalósítása a vágó segítségével

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

```
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 `fel t` részekben vágó nem lehet, ISO módban lehet, de hatásköre (szülője) a `fel t` rész.
- Az akkor részekben lehet vágó. Ennek hatásköre, a `->` nyílból generált vágóval ellentétben, a teljes `p` predikátum (ilyenkor a Prolog megvalósítás egy speciális, ún. távolbható vágó használ).
- Vágót rendkívül ritkán szükséges feltételes szerkezetben szerepeltetni.

A keresési tér szűkítése LP-144

Deklaratív programozás. BMÉ VIK, 2002. őszi félév

(Logikai Programozás)

## 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([_|_], Last) :-
  ( L = [] -> Last = E
  ; last(L, Last)
  ).

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

Deklaratív programozás. BMIE VIK, 2002. őszi félév

(Logikai Programozás)

## Feltételes szerkezetek

A keresési tér szikítése LP-147

**Feltételes szerkezet — példa**

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

### Általános alak

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

## Diszjunktív feltételes szerkezet

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

### Általános alak

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

Deklaratív programozás. BMIE VIK, 2002. őszi félév

(Logikai Programozás)

## A vágás első alapesete — klóz mellett való elkötelezés

- A klóz mellett elkötelezés általában egyszerű feltételes szerkezetet jelent.
 

```
szülő :- feltétel, !, akkor.
szülő :- egyébként.
```
- A vágó szikségletlené 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 ≡ a ∧ b ∨ ¬a ∧ c
```
- A vágó által kiváltott negált feltélt célszerű kommentként jelezni:
 

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

Deklaratív programozás. BMIE VIK, 2002. őszi félév

(Logikai Programozás)

## Feltételes szerkezetek és fejillesztés

A keresési tér szikítése LP-148

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

```
% a vágó előttt 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 alapszabály**a:

- A kimenő paraméterek értékadását mindig a vágó után végezzük!

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

- Ez nemcsak általánosanban 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 :-)

Deklaratív programozás. BMIE VIK, 2002. őszi félév

(Logikai Programozás)

## A bevezető példáknak a vágás alapszabályát betartó változata

```
% fakt(+N, ?F) : N1 = 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([_], Last) :- !, Last = E.
last([_|L], E) :- last(L, E).

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

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

Deklaratív programozás. BMÉ VIK, 2002. 6561 félév

(Logikai Programozás)

A keresési tér szűkítése LP-151

## A vágás második alapesete — első megoldásra való megszorítás

Mikor használjuk az első megoldásra megszorító vágót?

- behelyettesítést nem okozó, eldöntendő kérdés esetén;
- feladat-specifikus 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) :-
    út_vonal(N, A, B, Hossz), Hossz >= Min, !.
```

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

Deklaratív programozás. BMÉ VIK, 2002. 6561 félév

(Logikai Programozás)

## 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 */.
```

Deklaratív programozás. BMÉ VIK, 2002. 6561 félév

(Logikai Programozás)

A keresési tér szűkítése LP-152

## Feladat-specifikus optimalizálás

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

```
% Az L lista első eleme H-szor ismétlődik
% a lista kezdőszeleteként.
kezdethossz(L, H) :-
    L = [_|_], append(Ek, Farok, L),
    \+ Farok = [_|_], !,          % 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
```

Deklaratív programozás. BMÉ VIK, 2002. 6561 félév

(Logikai Programozás)





## 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/1-ban szereplő vezérlési szerkezetek lényegében ugyanolyan fűnek, mint az interpretált (consul-t-al 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
```

Deklaratív programozás. BMÉ VIK, 2002. 6564 félév

(Logikai Programozás)

## 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
    ),
    !.
```

Deklaratív programozás. BMÉ VIK, 2002. 6564 félév

(Logikai Programozás)

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

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

A call/1 viszonylag költséges: egy 1414 hosszú lista megfordítása msec-vel (kb. 1 millió append hívás), minden append körül egy felesleges call/1-al 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     |

Deklaratív programozás. BMÉ VIK, 2002. 6564 félév

(Logikai Programozás)

## Példa: magasabbrendű reláció definíciója

- 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 szóközi

| ?- _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 növe:
   forall(append(_, [A,B|_], _L), A < B).

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

Deklaratív programozás. BMÉ VIK, 2002. 6564 félév

(Logikai Programozás)



## Determinizmus

- Egy eljáráshívás **determinisztikus**, ha (legfeljebb) egyféleképpen sikerülhet.
- Egy eljáráshívásnak egy sikeres végrehajtása **determinisztikusan futott le**:
  - ha nem hagyott választási pontot a híváshoz tartozó részében, 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(1, X).          % det. hívás,
p(2, b).          | 1 1 Exit: p(1,a)    % det. lefutás
p(3, b).          | ?- p(Y, a).         % det. hívás,
                  | ?- p(Y, b), Y > 2. % nemdet. lefutás
                  | 1 1 Exit: p(1,a)  % det. hívás
                  | 1 1 Exit: p(2,b)  % nemdet. lefutás
                  | 1 1 Exit: p(3,b)  % det. lefutás
    
```

## DETERMINIZMUS ÉS INDEXELÉS

### 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(Norvar, Y)` hívás nem hoz létre választási pontot:

```

p(1, a).          | p(X, Y) :-
p(2, b).          |   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 fejt-argumentum alapján indexel (first argument indexing).
- Az indexelés alapja az első fejt-argumentum külső funkora:
  - 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 részhalmozát.
  - Futáskor lényegében konstans idő alatt választunk a részhalmozak közül.
- **Fontos:** ha egyetlen a részhalmoz, 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:
 

|                                  |                            |
|----------------------------------|----------------------------|
| <pre>{(1) (2) (3) (4) (5)}</pre> | ha $A$ változó;            |
| <pre>{(1) (2)}</pre>             | ha $A = 0$ ;               |
| <pre>{(2) (3) (4)}</pre>         | ha $A$ fő funktora $s/1$ ; |
| <pre>{(2) (5)}</pre>             | ha $A = 9$ ;               |
| <pre>{(2)}</pre>                 | 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.

Deklaratív programozás. BMÉ VIK, 2002. 6561 félév

(Logikai Programozás)

## Indexelés — további tudnivalók

- Indexelés és aritmetika
  - Az indexelés nem foglalkozik aritmetikai vizsgálatokkal.
  - Pl. az  $N = 0$  és  $N > 0$  feltételek nem „zárták ki” egymást.
  - Az alábbi  $\text{fakt}/2$  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 (funktorok: `'[]' / 0` ill. `'[]' / 2`).
  - A két klóz sorrendje nem érdekes (feltéve, hogy zárt listával hívjuk az első pozíción) — de azért tegyük a leálló klóz mindíg előre.

Deklaratív programozás. BMÉ VIK, 2002. 6561 félév

(Logikai Programozás)

## Struktúrák, változók a fejarumentumban

- Azonos funktorú struktúrák az első fejarumentumban:
  - Ha a klózek szétválasztásához szükség van az első (struktúra) argumentum részére is, akkor érdemes segédeljárást bevezetni.
  - Például  $p/2$  és  $q/2$  ekvivalens, de  $q(\text{Nonvar}, Y)$  determinisztikusan fut le!
 

|                                                          |                                                           |                                              |
|----------------------------------------------------------|-----------------------------------------------------------|----------------------------------------------|
| <pre>p(0, a) . p(s(0), b) . p(s(1), c) . p(9, z) .</pre> | <pre>q(0, a) . q(s(x), Y) :- q(s(1), c) . q(9, z) .</pre> | <pre>q_segged(0, b) . q_segged(1, c) .</pre> |
|----------------------------------------------------------|-----------------------------------------------------------|----------------------------------------------|
- Fejlesztés kiváltása egyenlőséggel (vö. SML réteglal minta)
  - Az indexelés figyelembe veszi a törzs elején szereplő egyenlőséget:
 

```
p(X, ...) :- X = K1f, ... esetén K1f funktora szerint indexel.
```
  - Példa: lista hosszának reciprokra, 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.
```

Deklaratív programozás. BMÉ VIK, 2002. 6561 félév

(Logikai Programozás)

## 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) .
```

Deklaratív programozás. BMÉ VIK, 2002. 6561 félév

(Logikai Programozás)



## 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. személynegyűestre 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).
```

Deklaratív programozás. BMÉ VIK, 2002. 6564 félév

(Logikai Programozás)

## Jobbrekurzív listaösszeg — akkumulátorpár segítségével

Jobbrekurzió és akkumulátorok LP-175

- 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.

Deklaratív programozás. BMÉ VIK, 2002. 6564 félév

(Logikai Programozás)

## 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 kinni 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.

Deklaratív programozás. BMÉ VIK, 2002. 6564 félév

(Logikai Programozás)

## Az akkumulátorok használata

Jobbrekurzió és akkumulátorok LP-176

- Az akkumulátorokkal általában 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: `VálL0`; közbülső értékek: `VálL1`, ..., `VálLn`; végérték: `VálL`.
- A Prolog akkumulátorpár nem más mint a funkcionális programozásból ismert gyűjtőargumentum és a függvény eredményének együttese.

Deklaratív programozás. BMÉ VIK, 2002. 6564 félév

(Logikai Programozás)

## 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).
```

Előrebocsított megjelyzés: 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

```
% suml2(+L, +S0, ?S, +Q0, ?Q): S-S0 = $\Sigma$  Li, Q-Q0 = $\Sigma$  Li*Li
suml2([], S, S, Q, Q).
suml2([X|L], S0, S, Q0, Q):-
    S1 is S0+X, Q1 is Q0+X*X, suml2(L, S1, S, Q1, Q).
```

- Többszörös akkumulátorok összevonása

```
% suml2(+L, +S0/Q0, ?S/Q): S-S0 = $\Sigma$  Li, Q-Q0 = $\Sigma$  Li*Li
suml2([], SQ, SQ).
suml2([X|L], S0/Q0, SQ):-
    S1 is S0+X, Q1 is Q0+X*X, suml2(L, S1/Q1, SQ).
```

Deklaratív programozás. BMÉ VIK, 2002. őszi félév

(Logikai Programozás)

## Korábbi listakezelő predikátumok

- A revapp mint akkumuláló eljárás

```
% revapp(Xs, L0, L): Xs megfordítását L0 elé fűzve 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 (különbségi lista): az a lista amelyet úgy kapunk, hogy L végétől elhagyjuk L0-t (előfelétél: L0 szuffixuma L-nek).

- Az append is tekinthető akkumuláló eljárásnak (a 2. és 3. arg. felcserélő). Az akkumulálás: az L0 elejétől sorra elhagyjuk Xs elemeit, végül marad L.

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

Deklaratív programozás. BMÉ VIK, 2002. őszi félév

(Logikai Programozás)

## 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 követető N db b-ből áll.
anbn(N, L) :-
    an(N, a, AN),
    an(N, b, BN),
    append(AN, BN, 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|L1]) :-
    N > 0,
    N1 is N-1,
    an(N1, A, L0, L1).
```

## $a^n b^n$ 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 követető N db b-ből áll.
anbn(0, L0, L) :- !, L = L0.
anbn(N, L0, [a|L1]) :-
    N > 0,
    N1 is N-1,
    anbn(N1, [b|L0], L1).
```

- A második klóz nem jobbrekurzív változata

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

Deklaratív programozás. BMÉ VIK, 2002. őszi félév

(Logikai Programozás)

Deklaratív programozás. BMÉ VIK, 2002. őszi félév

(Logikai Programozás)

**$a^n b^n$  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'); // előlről
    a = &(*a)->next; // hátra épít
    b = new link('b', b); // hátról előre épít
  }
  *a = b; return l;
}
```

Deklaratív programozás. BMÉ VIK, 2002. 6564 félév

(Logikai Programozás)

Jobbkurzió és akkumulátorok LP-183

**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)
  ).
```

Deklaratív programozás. BMÉ VIK, 2002. 6564 félév

(Logikai Programozás)

**Összetettebb adatszerkezetűk akkumulálása**

## ● Az adatszerkezet:

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

## ● A fa csomópontjaiban tároljuk a számértékeket a levelek nem tárolnak információt.

## ● Egészszek 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)) ?
```

Deklaratív programozás. BMÉ VIK, 2002. 6564 félév

(Logikai Programozás)

Jobbkurzió és akkumulátorok LP-184

**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
```

Deklaratív programozás. BMÉ VIK, 2002. 6564 félév

(Logikai Programozás)

## 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
```

Deklaratív programozás. BMÉ VIK, 2002. 6561 félév

(Logikai Programozás)

Imperatív programok átirása LP-187

## 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).

Deklaratív programozás. BMÉ VIK, 2002. 6561 félév

(Logikai Programozás)

## IMPERATÍV PROGRAMOK ÁTÍRÁSA PROLOGBA

## A hatv C függvénynek megfelelő Prolog eljárás

- A függvény eredménye a reláció utolsó arg.-a: `hatv(+A, +H, ?E): AH = E.`
  - A ciklusnak segédeljártás felül meg: `hatv(+A0, +H0, +E0, ?E): A0H0 * E0 = 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* felül 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;
```

Imperatív programok átirása LP-188

Deklaratív programozás. BMÉ VIK, 2002. 6561 félév

(Logikai Programozás)



## A C ciklus és a Prolog eljárás kapcsolata

- A ciklust megvalósító Prolog eljárás minden pontján minden C változónak megfélelelhető 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; argumentumában az egyes C változóknak pillanatnyilag megfélelelt Prolog változóval.
- A C ciklus **invariánsa** nem más mint a Prolog eljárás fejkommentje, a példában:
 

```
% natv(+A0, +H0, +E0, ?E): A0^H0 * E0 = E.
```

Deklaratív programozás. BMÉ VIK, 2002. 6561 félév

(Logikai Programozás)

## Ciklikus programok helyességének bizonyítása

- A ciklusokat „fel kell vágni” egy **ciklus-invariánssal**, 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 natv(int a0, unsigned h0) /* utófeltétel: natv(a0, h0) = a0^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^k&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 */
}
```

Imperatív programok ábrása LP-191

Deklaratív programozás. BMÉ VIK, 2002. 6561 félév

(Logikai Programozás)

## 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:  $x = \text{mfok}_k\text{gyok}(a, b, c)$ 
  - előfeltételek:  $b^2 - 4*a*c \geq 0$ ,  $a \neq 0$
  - utófeltétel:  $a*x^2 + b*x + c = 0$
  - a program:
 

```
double mfok_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.

Deklaratív programozás. BMÉ VIK, 2002. 6561 félév

(Logikai Programozás)

## 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 megfélelelt 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).
```

Imperatív programok ábrása LP-192

Deklaratív programozás. BMÉ VIK, 2002. 6561 félév

(Logikai Programozás)

## Fibonacci sorozat — Prolog stílusban

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

```

fib(N, F) :-
    fib(N, 0, 1, F).
% 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;
% }
fib(_, F0, _, F0).

```

Deklaratív programozás: BMÉ VIK, 2002. őszi félév

(Logikai Programozás)

## Keresési feladat Prologban — felsorolás vagy gyűjtés?

Megoldások gyűjtése és felsorolása LP-195

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

|                                                                                                                                                                                                                                                 |                                                                                                                                                                                                                                                          |
|-------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------|----------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------|
| <pre> % 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([_ L], [P Pk]) :-     páros_elemei(L, Pk). </pre> | <pre> % 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([_ L], P) :-     % _X akár páros, akár páratlan     % folytatjuk a felsorolást:     páros_eleme(L, P). </pre> |
| <pre> % egyszerűbb megoldás: páros_eleme2(L, P) :-     member(P, L), P mod 2 =:= 0. </pre>                                                                                                                                                      |                                                                                                                                                                                                                                                          |

Deklaratív programozás: BMÉ VIK, 2002. őszi félév

(Logikai Programozás)

## MEGOLDÁSOK GYŰJTÉSE ÉS FELSOROLÁSA

### Mi a közös a felsoroló és gyűjtő megoldásokban?

Megoldások gyűjtése és felsorolása LP-196

- Keresünk meg a közös részt a páros\_elemei és páros\_eleme eljárásokban!
- Mindkétőben 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([_|L], P, L).

```
- A köv\_páros eljárásra épülő gyűjtő és felsoroló eljárások:
 

|                                                                                                                                                                                                     |                                                                                                                                                                    |
|-----------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------|--------------------------------------------------------------------------------------------------------------------------------------------------------------------|
| <pre> % 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(_, []). </pre> | <pre> % 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)     ). </pre> |
|-----------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------|--------------------------------------------------------------------------------------------------------------------------------------------------------------------|

Deklaratív programozás: BMÉ VIK, 2002. őszi félév

(Logikai Programozás)

## A gyűjtő és felsoroló sémák összehasonlítása

- A páros elemeket gyűjtő ill. felsoroló eljárások alapján adjunk meg egy általános sémát a kétféle eljárásfűpusral!
- 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)
    ).
```

Deklaratív programozás. BME VIK, 2002. 564. oldal

(Logikai Programozás)

## Fennsíkok felsorolása — 2., hatékony megoldás

Megoldások gyűjtése és felsorolása LP-199

- 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)
    ).
```

Deklaratív programozás. BME VIK, 2002. 564. oldal

(Logikai Programozás)

## Egy összetettebb példa: fennsíkok felsorolása

- Egy listában fennsíkokat 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 kezdőpozícióik.
- fennsík(L, F, H): Az L listában az F (1-től számozott) pozíción egy H hosszú fennsík van.

- Egy gyorsprogramozási módszerrel készült (Prolog beker) megoldás:

|                                                                                                                                                                                                                                                                                 |                                                                                                                                                                                                                                                                                                                                       |
|---------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------|---------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------|
| <pre>fennsík0(L, F, H) :-     fennsík0(L, F, H) :-         Reste = [E,E _],         append(Eljeje, Reste, L),         \+ last(Eljeje, E),         length(Eljeje, F0), F is F0+1,         kezdet_hossz(Testing, H).     % kezdet_hossz/2 defíniációját     % lásd korábban</pre> | <pre>fennsík1(L, F, H) :-     fennsík1(L, F, H) :-         Reste = [E,E _],         append(Eljeje, Reste, L),         \+ last(Eljeje, E),         length(Eljeje, F0), F is F0+1,         % kezdet_hossz/2 kifejtve:         ( append(Ek, Farok, Reste),           \+ Farok = [E _] -&gt;             length(Ek, H)           ).</pre> |
|---------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------|---------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------|

Deklaratív programozás. BME VIK, 2002. 564. oldal

(Logikai Programozás)

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

Megoldások gyűjtése és felsorolása LP-200

- 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 elejétől a maximális számú
% E-vel azonos elemet leahagyva marad L, a leahagyott 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).
```

Deklaratív programozás. BME VIK, 2002. 564. oldal

(Logikai Programozás)

## 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_eleme(L, Pk), member(P, Pk).
```

Természetesen ez így nem hatékony!

- Gyűjtés felsorolásból: a megoldásgyűjtő beépített eljárások segítségével, pl.

```
páros_eleme(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 eljárásnévként 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):

$Lista = \{ Gyűjtő \text{ másolat} \mid (\exists X \dots Z) Cél \text{ igaz} \}$

ahol  $X, \dots, Z$  a `findall` hívásban levő szabad változók (azaz olyan, ahívás pillanatában behelyettesíthetően 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 eljárásnévként értelmezi, meghívja;

- összegyűjti a megoldásokat (a `Gyűjtő`-t és a szabad változók behelyettesítését);

- 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):

$Lista = \{ Gyűjtő \mid Cél \text{ igaz} \}$ ,  $Lista \neq []$ .

## A bagof megoldásgyűjtő eljárás (folyc)

- Explicit kvantorok
  - `bagof(Gyűjtő, V1 ^ ... ^ Vn ^ Cél, Lista)` alakú hívása a `V1, ..., Vn` változókat egzisztenciálisan követhet tekintni, nem sorolja fel.
  - jelentése:  $Lista = \{ Gyűjtő \mid (\exists V1, \dots, Vn) Cél igaz \} \neq []$ .
  - | ?- `gráf(_G), bagof(B, A^member(A-B, _G), VegP).`  
 $\implies VegP = [b,c,c,d,d] ? ; no$
- Egymásba ágyazott gyűjtések
  - szabád változók esetén a `bagof` nendeterminisztikus lehet, így skatulyázható:
    - % *A G irányított gráf Fokszámlistája FL:*
    - %  $FL = \{ A-N \mid N = \{ \{ V \mid A-V \in G \} \}$
    - Fokszámai(G, FL) :-*
    - `bagof(A-N, V, member(A-V, G), Vks),`  
`length(Vk, N)`  
`), FL).`
    - | ?- `gráf(_G), fokszámai(_G, FL).`  
 $\implies FL = [a-2,b-2,c-1] ? ; no$

Deklaratív programozás. BMÉ VIK, 2002. 6561 félév

(Logikai Programozás)

1P-207

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

Megoldásgyűjtő beépített eljárások

- az eljárás végrehajtása:
  - ugyanaz mint `bagof(Gyűjtő, Cél, L0)`, `sort(L0, Lista)`,
  - it `sort/2` egy univerzális rendező eljárás (lásd később), amely
  - az eredménylistát rendezzi (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).`  $\implies Pk = [a,b,c,d] ? ; no$

Deklaratív programozás. BMÉ VIK, 2002. 6561 félév

(Logikai Programozás)

## A bagof megoldásgyűjtő eljárás (folyc)

- Fokszámlista hatékonyabb előállítása
  - a vezérlési szerkezeteket célszerű elkerülni a meta-argumentumokban
  - segédeljárás bevezetésével a kvantor is szükségletlené válik:
    - % *Az A pont foka a G irányított gráfban N, N>0.*
    - `pont_foka(A, G, N) :-`  
`bagof(V, member(A-V, G), Vks), 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).`  $\implies L = [] ? ; no$
  - | ?- `bagof(X, (between(1, 5, X), X<0), L).`  $\implies no$
  - | ?- `findall(S, member(S, [f(X,X),g(X,Y)]), L).`  
 $\implies L = [f(_A,_A),g(_B,_C)] ? ; no$
  - | ?- `bagof(S, member(S, [f(X,X),g(X,Y)]), L).`  
 $\implies L = [f(X,X),g(X,Y)] ? ; no$
- A `bagof/3` logikailag tisztább mint a `findall/3`, de időigényesebb!

Deklaratív programozás. BMÉ VIK, 2002. 6561 félév

(Logikai Programozás)

## META-LOGIKAI ELJÁRÁSOK

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

- A Prolog kifejezések pillanatnyi behelyettesíthetőségi állapotát vizsgáló eljárások (értelmszerűen sorrendfüggetlek):
  - kifejezések osztályozása (1)
 

```
| ?- var(X) /* X változó? */, X = 1. => X = 1
| ?- X = 1, var(X). => no
```
  - kifejezések rendezése (4)
 

```
| ?- X @< 3 /* X megelőzői 3-t? */, X = 4. => X = 4
| % a változók megelőzőik a nem változó kifejezéseket
| ?- X = 4, X @< 3. => no
```
- Prolog kifejezéseket szétszedő vagy összerakó eljárások:
  - (struktúra) kifejezés  $\iff$  név és argumentumok (2)
 

```
| ?- X = f(alma,körte), X = .. L => L = [f,alma,körte]
```
  - atomok és számok  $\iff$  karaktereik (3)
 

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

Deklaratív programozás. BMÉ VIK, 2002. 6564 félév

(Logikai Programozás)

**Oszályozó eljárások: elágaztatás behelyettesíthetőség 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) :-
    % length(?L, +I0, -I1):
    % Az L lista I-I0 hosszú.
    length([], I, I).
length([_|L], I0, I) :-
    !, !s I0+1,
    length(L, I1, I).

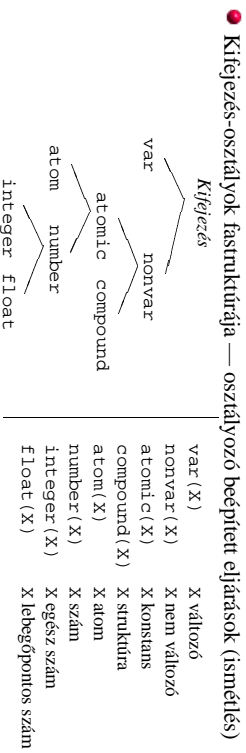
% length(?L, +I0, -I1):
% Az L lista I-I0 hosszú.
length([], I, I).
length([_|L], I0, I) :-
    !, !s I0+1,
    length(L, I1, I).

% length(?L, Len) => Len = 2 ? ; no
% length([1,2], 3) => no
% length(L, 3) => L = [_A,_B,_C] ?;no
% length(L, Len) => L = [], Len = 0 ? ;
% L = [_A], Len = 1 ? ; L = [_A,_B], Len = 2 ?
```

Deklaratív programozás. BMÉ VIK, 2002. 6564 félév

(Logikai Programozás)

**Kifejezések osztályozása**



- Kifejezés-osztályok fastruktúrája — osztályozó beépített eljárások (ismétlés)
  - `simple(X)`: X nem összetett (konstans vagy változó);
  - `ground(X)`: X römör, azaz nem tartalmaz behelyettesíthetlen változót.
- Az osztályozó eljárások használata — példák
  - `var, nonvar` — többirányú eljárásokban a különböző irányok elágaztatása
  - `number, atom, ...` — nem-megkülönböztetett úniók feldolgozása (pl. szimbolikus deriválás)

Deklaratív programozás. BMÉ VIK, 2002. 6564 félév

(Logikai Programozás)

**Struktúrák szétszedése és összerakása: az univ eljárás**

- Az `univ` eljárás hívási mintái:
  - `+Kif = .. ?Lista`
  - `-Kif = .. +Lista`
- Az eljárás jelentése: Igaz, ha
  - `Kif = Fun(A1, ..., An)` és `Lista = [Fun, A1, ..., An]`, ahol `Fun` egy atom és `A1, ..., An` 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']
```

Deklaratív programozás. BMÉ VIK, 2002. 6564 félév

(Logikai Programozás)



## 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 (argumentumában 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]
```

Deklaratív programozás. BMÉ VIK, 2002. őszi félév

(Logikai Programozás)

## Az *univ* alkalmazása: ismétlődő sémák összevonása

- A feladat: egy szimbolikus aritmetikai kifejezésben a kiértékelhető (infix) részkifejezések helyettesítése az énékükkel.

### • 1. megoldás, *univ* nélkül:

```
% Az X szimbolikus kifejezés egyszerűsítése EX.
egysz0(X, EX) :-
    atom(c(X), I, 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), I, 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
```

Deklaratív programozás. BMÉ VIK, 2002. őszi félév

(Logikai Programozás)

## 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)
```

Deklaratív programozás. BMÉ VIK, 2002. őszi félév

(Logikai Programozás)

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

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

```
egysz(X, EX) :-
    atom(c(X), I, 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 *több* kifejezésre:

```
egysz1(Kif, EKif) :-
    Kif =.. [M|ArgL], egysz1_lista(ArgL, EArgL), EKif0 =.. [M|EArgL],
    % catch(:Goal, _:_KCl): ha Goal kivételre dob, KCl-t futtatja:
    catch(EKif is EKif0, _, EKif = EKif0).
egysz1_lista([], []).
egysz1_lista([K|Ks], [E|EKs]) :-
    egysz1(K, E), egysz1_lista(Ks, EKs).
| ?- egysz1(f(1+2+a, exp(3,2), a+1+2), E). => E = f(3+a,9.0,a+1+2)
```

Deklaratív programozás. BMÉ VIK, 2002. őszi félév

(Logikai Programozás)



## Univ alkalmazása általános kifejezés-bejárásra: kirratás

- A feladat: egy tetszőleges kifejezés kirratása úgy, hogy
  - a kétfargumentumú operátorok zárójelzet infix formában,
  - minden más alap-struktúra alakban jelenjék meg.

```

Ki(KiF) :-
    compound(KiF), !, KiF =.. [Func, A1|ArgL],
    ( % kétfargumentumú kifejezés, funktora infix operátor
      ArgL = [A2], current_op(_, Kind, Func), infix_fafta(Kand),
    -> write('(', Ki(A1),
      write(' ', write(Func), write(' ', Ki(A2), write(')')
      ; write(Func),
        write('(', Ki(A1), arglistaki(ArgL), write(')')
      ),
    Ki(KiF) :- write(KiF).

% infix_fafta(F): F egy infix operátorfafta.
infix_fafta(xfx), infix_fafta(xfy). infix_fafta(yfx).

% Az [A1,...,An] listát ",A1,...,An" alakban kirrja.
arglistaki([]).
arglistaki([_|_]) :- write(' ', Ki(A), arglistaki(A)).

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

Deklaratív programozás. BME VIK, 2002. 6.szá. félév

(Logikai Programozás)

## Általános kifejezés-bejárás univ-val: változómentesítés

- A változómentesítés egy saját megvalósítása:

```

% A Term kifejezésben levő változókat '$mymar(I)' sdb.
% struktúrákkal helyettesíti be, I = N0, ... N-1.
numbervarsI(Term, N0, N) :-
    var(Term), !,
    Term = '$mymar'(N0), N is N0+1.
numbervarsI(Term, N0, N) :-
    Term =.. [_|Args],
    numbervarsI_list(Args, N0, N).

% numbervarsI_list(L, N0, N): Az L listában levő változókat
% '$mymar(I)' sdb. struktúrákkal helyettesíti be, I = N0, ... N-1.
numbervarsI_list([], N, N).
numbervarsI_list([_|As], N0, N) :-
    numbervarsI(A, N0, N1), numbervarsI_list(As, N1, N).

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

Deklaratív programozás. BME VIK, 2002. 6.szá. félév

(Logikai Programozás)

## 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-al való kirratáskor változónévként (A, B...) jelennek meg.
  - A write\_term(KiF, Opciók) beépített eljárás kirrja 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), n1,
      write_term(_K, [quoted(true),numbervars(false)]), n1.
====>
[f(A),g(B),A]
[f('$VAR'(0)),g('$VAR'(1)),$VAR'(0)]
N = 2
    
```

- A feladat: elkészítenődő egy numbervarsI/3 eljárás, amely '\$VAR' helyett '\$mymar' funktort használ.

Deklaratív programozás. BME VIK, 2002. 6.szá. félév

(Logikai Programozás)

## numbervarsI egy alkalmazása

### Két kifejezés azonosossága

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

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

% azonos2/2 és azonos/2 teljeseen ekvivalens.
% \+ \+ X : csakkor sikeres amikor X, de változóbehe lyettesítést nem okoz
azonos2(X, Y) :-
    \+ \+ (numbervarsI(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 ?
    
```

Deklaratív programozás. BME VIK, 2002. 6.szá. félév

(Logikai Programozás)

## Univ alkalmazása: részkiefezések keresése

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

```
% Kif_szám( PKif, ?N, ?Kiv): Kif Kiv kiválasztójú része az N szám.
Kif_szám(X, N, Kiv) :-
    number(X, I, 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, XI),
    Kif_szám(XI, N, Kiv).
```

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

Deklaratív programozás. BMÉ VIK, 2002. 564. oldal

(Logikai Programozás)

## 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).
```

- 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 ?
```

Deklaratív programozás. BMÉ VIK, 2002. 564. oldal

(Logikai Programozás)

## 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  $c_1c_2\dots c_n$  karakterekből áll, akkor KódLista-t egyesíti a  $[k_1, k_2, \dots, k_n]$  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
```

Deklaratív programozás. BMÉ VIK, 2002. 564. oldal

(Logikai Programozás)

## 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 ízes számrendszerbeli alakja a KódLista karakterkód-listának felel meg.
- Végrehajtása:
  - Ha Szám adott (bemenő), és a  $c_1c_2\dots c_n$  karakterekből áll, akkor KódLista-t egyesíti a  $[k_1, k_2, \dots, k_n]$  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'a|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! :-)
```

Deklaratív programozás. BMÉ VIK, 2002. 564. oldal

(Logikai Programozás)

**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 kifejezőesztályba 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 rendszertífűgő.
  4. Ha  $X$  és  $Y$  lebegőpontos vagy egész szám, akkor  $X \prec Y \Leftrightarrow X < Y$ .
  5. Ha  $X$  és  $Y$  név, akkor sorrendjünk megegyezik a lexikografikus (abc) sorrenddel.
  6. Ha  $X$  és  $Y$  struktúrák:
    - 6.1. Ha  $X$  és  $Y$  aritása ( $\equiv$  argumentumszáma) különböz,  $X \prec Y \Leftrightarrow X$  aritása kisebb mint  $Y$  aritása.
    - 6.2. Egyébként ha a rekordok neve különböz,  $X \prec Y \Leftrightarrow X$  neve  $\prec Y$  neve.
    - 6.3. Egyébként (azonos név, azonos aritás) bahról az első nem azonos argumentum dönt.
- (A SICStus Prologban kiterjesztésként megengedett végtelen (ciklikus) kifejezésekre a fenti rendezés nem érvényes.)

Deklaratív programozás. BMÉ VIK, 2002. 654. oldal

(Logikai Programozás)

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

Meta-logikai eljárások IP-227

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

% class/+-T, -C): A T kifejezés a C-edik kifejezőesztályba 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)
        % 2. szabály
    ).
```

Deklaratív programozás. BMÉ VIK, 2002. 654. oldal

(Logikai Programozás)

**Kifejezések összehasonlítása — beépített eljárások**

- Két tetszőleges kifejezés összehasonlítását végző eljárások:

| hívás         | igaz, ha                                   |
|---------------|--------------------------------------------|
| KiF1 == KiF2  | KiF1 $\neq$ KiF2 $\wedge$ KiF2 $\neq$ KiF1 |
| KiF1 \== KiF2 | KiF1 $\prec$ KiF2 $\vee$ KiF2 $\prec$ KiF1 |
| KiF1 @< KiF2  | KiF1 $\prec$ KiF2                          |
| KiF1 @=< KiF2 | KiF2 $\neq$ KiF1                           |
| KiF1 @> KiF2  | KiF2 $\prec$ KiF1                          |
| KiF1 @>= KiF2 | KiF1 $\neq$ 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)
```

Deklaratív programozás. BMÉ VIK, 2002. 654. oldal

(Logikai Programozás)

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

Meta-logikai eljárások IP-228

```
% 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 sorozatú argumentumai
% lexikografikusan megelőzik S2 azonos sorozatú 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).
```

Deklaratív programozás. BMÉ VIK, 2002. 654. oldal

(Logikai Programozás)

## EGYENLŐSÉGJELŰK — ÖSSZEHOGLALÁS

- $U = V$ :  $U$  egyesítendő  $V$ -vel.  
Soha sem jelez hibát.  
| ?-  $X = 1+2$ .     $\Rightarrow X = 1+2$   
| ?-  $3 = 1+2$ .     $\Rightarrow$  no
- $U = V$ :  $U$  azonos  $V$ -vel.  
Soha sem jelez hibát és soha sem helyettesít be.  
| ?-  $X = 1+2$ .     $\Rightarrow$  no  
| ?-  $3 = 1+2$ .     $\Rightarrow$  no  
| ?-  $+(1,2)=1+2$      $\Rightarrow$  yes
- $U := V$ :  $U$  az  $U$  és  $V$  aritmetikai kifejezések értéke megegyezik.  
Hibát jelez, ha  $U$  vagy  $V$  nem (tömör) aritmetikai kifejezés.  
| ?-  $X := 1+2$ .     $\Rightarrow$  **hiba**  
| ?-  $1+2 := X$ .     $\Rightarrow$  **hiba**  
| ?-  $2+1 := 1+2$ .     $\Rightarrow$  yes  
| ?-  $2.0 := 1+1$ .     $\Rightarrow$  yes
- $U$  is  $V$ :  $U$  egyesítendő a  $V$  aritmetikai kifejezés értékével.  
Hiba, ha  $V$  nem (tömör) aritmetikai kifejezés.  
| ?-  $2.0$  is  $1+1$ .     $\Rightarrow$  no  
| ?-  $X$  is  $1+2$ .     $\Rightarrow X = 3$   
| ?-  $1+2$  is  $X$ .     $\Rightarrow$  **hiba**  
| ?-  $3$  is  $1+2$ .     $\Rightarrow$  yes  
| ?-  $1+2$  is  $1+2$ .     $\Rightarrow$  no
- $(U = . . V$ :  $U$  „szétszedetje” a  $V$  lista)  
| ?-  $1+2 = . . X$ .     $\Rightarrow X = [+1, 2]$   
| ?-  $X = . . [f, 1]$ .     $\Rightarrow X = f(1)$

Deklaratív programozás: BMÉ VIK, 2002. őszi félév

(Logikai Programozás)

Egyenlőségjelűk — összehajlás IP-231

## A Prolog nem-egyenlőség jellegű beépített eljárásai

- A nem-egyenlőség jellegű eljárások soha sem helyettesítenek be változót!  
| ?-  $X \neq 1+2$ .     $\Rightarrow$  no  
| ?-  $+(1,2) \neq 1+2$ .     $\Rightarrow$  no
- $U \neq V$ :  $U$  nem egyesíthető  $V$ -vel.  
Soha sem jelez hibát.  
| ?-  $X \neq 1+2$ .     $\Rightarrow$  yes  
| ?-  $3 \neq 1+2$ .     $\Rightarrow$  yes  
| ?-  $+(1,2) \neq 1+2$      $\Rightarrow$  no
- $U \neq V$ :  $U$  és  $V$  aritmetikai kifejezések értéke különbözők.  
Hibát jelez, ha  $U$  vagy  $V$  nem (tömör) aritmetikai kifejezés.  
| ?-  $X \neq 1+2$ .     $\Rightarrow$  **hiba**  
| ?-  $1+2 \neq X$ .     $\Rightarrow$  **hiba**  
| ?-  $2+1 \neq 1+2$ .     $\Rightarrow$  no  
| ?-  $2.0 \neq 1+1$ .     $\Rightarrow$  no

Deklaratív programozás: BMÉ VIK, 2002. őszi félév

(Logikai Programozás)

## A Prolog egyenlőség-szerű beépített eljárásai

|       | Egyesítés | Azonosság | Aritmetika |
|-------|-----------|-----------|------------|
| $U$   | $U = V$   | $U = V$   | $U = V$    |
| 1     | no        | yes       | yes        |
| a     | no        | yes       | error      |
| $1+2$ | yes       | no        | yes        |
| $1+2$ | no        | yes       | no         |
| $1+2$ | no        | yes       | yes        |
| 3     | no        | yes       | no         |
| $X$   | $X=1+2$   | no        | error      |
| $X$   | $X=Y$     | no        | error      |
| $X$   | yes       | yes       | error      |

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

Deklaratív programozás: BMÉ VIK, 2002. őszi félév

(Logikai Programozás)

## Modulok defínitása SICStus Prolog nyelven

- A SICStus Prolog modulfogalmának jellemzői:
  - Minden modul külön állományba kell kerülnjön.
  - Az állomány első programemele egy modul-parames kell legyen:
 

```
:- module( ModulNév, [ExpFunktor1, ExpFunktor2, ...]).
```
  - *ExpFunktor* = az exportálandó eljárás funktora (név/argumentumszám)
- Példa:
 

```
:- module(platek, [fennsík/3]).           % platek 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(plateo).                  % a fenti modul betöltése
:- use_module(library(lists), [last/2]). % csak last/2 importált
```
- Modulvalíthkált hívási formát: *Modul:Hívás* a *Modul*-ban futtatja *Hívás*-t.
- A modulfogalom nem szigorú: *platek:első\_fennsík(...)* meghívható!

Deklaratív programozás. BME VIK, 2002. őszi félév

(Logikai Programozás)

Modularitás IP-235

## Meta-eljárások modularizált programban

- Eljárások átadása paraméterként modulközi hívásban gondot okozhat *m1.pl* állomány:
 

```
:- module(m1, [kétyszer/1]).
% :- meta_predicate kétyszer(:), (*).
kétyszer(X) :-
  X, X.
p :- write(ba).
```
- *m2.pl* állomány:
 

```
:- module(m2, [q/0,r/0]).
% :- use_module(m1).
q :- kétyszer(p).
r :- kétyszer(m2:p).
p :- write(ba).
```
- Futtatás:
 

```
| ?- [m1,m2].
| ?- q.      => ba ba
| ?- r.      => ba ba
```
- Automatikuss modul-kvalifikáció meta-predikátum deklarációival:
 

Ha *m1.pl*-ben elhagyjuk a (\*)-gal jelzett sor előtti % kommentjelet, akkor

```
| ?- q.      => ba ba!
```

Deklaratív programozás. BME VIK, 2002. őszi félév

(Logikai Programozás)

## Meta-predikátum deklaráció, modulnév-kiterjesztés

- Meta-predikátum deklaráció
  - Formája:
 

```
:- meta_predicate (eljárásnév)(módspec1), ..., (módspecn), ...
% (módspecj) lehet ':", '+', '-', '?',
% A ':' mód azt jelzi, hogy az adott argumentumot betöltéskor ún. modulnév-kiterjesztésnek kell alávetni. (A többi mód hatása azonos, be-kimenő irányt jelezhetünk segítségével.)
```
  - Egy *Kif* kifejezés modulnév-kiterjesztése a következő átalakítási jelenti:
 

ha *KIF:M:X* alakú, vagy egy olyan változó, amely az adott eljárás fejében meta-argumentum pozíción szerepel, akkor változatlanul hagyjuk:

    - egyébként helyettesítjük *curMod:Kif*-fel (*curMod* a kurrens modul).
  - Példa folyt. (th. az *m1*-beli *kétyszer* meta-predikátumnak deklarált!):
 

```
:- module(m2, [négyezer/1,q/0]).
% :- use_module(m1).
q :- kétyszer(p).
% tárolt alak:
% :- meta_predicate négyezer(:).
négyezer(X) :- kétyszer(X), kétyszer(X).      => változatlan
```

Deklaratív programozás. BME VIK, 2002. őszi félév

(Logikai Programozás)

## Magasabbrendű eljárások — listakezelés

- Magasabbrendű (vagy meta-eljárás) egy eljárás,
  - ha eljárásként értelmezi egy vagy több argumentumát
  - pl. `call/1`, `findall/3`, `\+ /1`, `setb`.
- 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]
```

Deklaratív programozás. BMÉ VIK, 2002. őszi félév

(Logikai Programozás)

Magasabbrendű eljárások LP-239

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

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

```
% 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 függvényvel!)

```
% 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).

Deklaratív programozás. BMÉ VIK, 2002. őszi félév

(Logikai Programozás)

## Részlegesen paraméterezett eljárásnévadások

- 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 elemekre az  $x \rightarrow x^2 + Pr + Q$  hozzárendelést alkalmazó eljárás:

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

```
másodfokú_képek(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_képek(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.

Deklaratív programozás. BMÉ VIK, 2002. őszi félév

(Logikai Programozás)



## Részlegesen paraméterezett eljáráshívások — segédesszközök

- 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 definiálандók:

```
:- 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).
```

...

Deklaratív programozás. BMÉ VIK, 2002. 6561 félév

(Logikai Programozás)

Magasbrendű eljárások 1P-243

## Rekurzív meta-eljárások — foldl és foldr

- `foldl(+Xs, :Pred, +Y0, -Y): Y0-dól indulva, az Xs elemekre 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).`  $\implies$  `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-dól indulva, az Xs elemekre 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).  $\implies$  E = 321
```

Deklaratív programozás. BMÉ VIK, 2002. 6561 félév

(Logikai Programozás)

## 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 elemekre 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).  $\implies$  L = [1,4,9,16]
| ?- map([1,2,3,4], másodfokú_képe(2,1), L).  $\implies$  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`.

Deklaratív programozás. BMÉ VIK, 2002. 6561 félév

(Logikai Programozás)

## 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:
    - :- dynami c(*E1* járásnév/*Argumentumszám*).
 (ha van klóza a programban, akkor az első előt — ilyenkor kötelező);
  - futási időben, adatháziskezelő beépített eljárással
- Adatháziskezelő eljárások („adatházis” = a program klózáinak összessége):
  - klóz felvétele első, utolsó helyre: `asserta/1`, `assertz/1`
  - klóz törlése (illesztéssel, többszörösen sikerülhet): `retract/1`
  - klóz lekérdezése (illesztéssel, többszörösen sikerülhet): `clause/2`

Deklaratív programozás. BMÉ VIK, 2002. 6564 félév

(Logikai Programozás)

## Klóz törlése: `retract/1`

Dinamikus adatháziskezelés IP-247

- `retract(:@Klóz)`
  - A Klóz klóz-kifejezésből megállapítja a predikátum funktorát.
  - Az adott predikátum klózái sora megpróbálja illeszteni Klóz-zal.
  - Ha az illesztés sikerül, akkor kitöri a klózt és sikeresen lefut.
  - Viszsalépés esetén folytatja a keresést (illeszt, töröl, sikerül, stb.)
- Példa (folytatás):
  - | ?- listing(p), retract((p(2,\_):-\_)), listing(p), fail. => no
- A futás kimenete:
 

|                         |                         |                         |
|-------------------------|-------------------------|-------------------------|
| <code>p(2, 0) :-</code> | <code>p(1, A) :-</code> | <code>p(1, A) :-</code> |
| <code>p(1, A) :-</code> | <code>q(A).</code>      | <code>q(A).</code>      |
| <code>p(A).</code>      | <code>p(2, A) :-</code> |                         |
| <code>p(2, A) :-</code> | <code>r(A).</code>      |                         |
| <code>r(A).</code>      |                         |                         |

Deklaratív programozás. BMÉ VIK, 2002. 6564 félév

(Logikai Programozás)

## 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* klózként 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).
```

Deklaratív programozás. BMÉ VIK, 2002. 6564 félév

(Logikai Programozás)

## Alkalmazási példa — egyszerűsített `findall`

Dinamikus adatháziskezelés IP-248

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

```
:- dynami c(megoldás/1).

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

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

  - | ?- findall(Y, (member(X, [1,2,3]), Y is X\*X), ML). => ML = [1,4,9]

Deklaratív programozás. BMÉ VIK, 2002. 6564 félév

(Logikai Programozás)

## Klóz lekérdezése: clause/2

- clause(:@Fej, ?Törzs)
- A Fej alapján megállapítja a predikátum funkcionát.
- Az adott predikátum klóznai sorra megpróbálja illeszteni a Fej :- Törzs kifejezéssel (tényállítási esetén Törzs = true).
- Ha az illesztés sikerült, akkor sikeresen lefut.
- Viszsalépés esetén folytatja a keresést (illeszt, sikerül, sítb.)

- 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		

Deklaratív programozás, BMÉ VIK, 2002. őszai félév

(Logikai Programozás)

## Nyomkövető interpreter - példafutás

Dinamikus adathéztáskészítés LP-251

```
:- 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).
```

<pre> - ?- 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([], [b], [b]) exit: app([], [b], [b]) exit: app([c], [b], [c,b]) exit: app([c], [b,c], [b], [c,b,c,b]) L = [b] ?</pre>	<pre>  ?- 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([], [b], [b]) exit: app([], [b], [b]) exit: app([c], [b], [c,b]) exit: app([c], [b,c], [b], [c,b,c,b]) L = [b] ?</pre>
--	---

Deklaratív programozás, BMÉ VIK, 2002. őszai félév

(Logikai Programozás)

## A clause eljárás alkalmazása: egyszerű nyomkövető interpreter

- Az alábbi interpreter csak „tisztia”, 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 áthataladását a Port kapun D bekezdésű nyomkövetéssel mutatja.
trace(G, D, Port) :-
    /*D szókózt ír ki:*/ tab(D),
    write(Port), write(' '), write(G), nl.
```

Deklaratív programozás, BMÉ VIK, 2002. őszai félév

(Logikai Programozás)

## NYELVTANI ELEMZÉS PROLOGBAN

## Egy egyszerű nyelvtani elemzési példa

- Bináris számok nyelvtana
 

```

<szám> ::= <számjegy> <számmaradék>
<számmaradék> ::= <számjegy> <számmaradék> | ε
<számjegy> ::= 0 | 1
      
```
- Ugyanez DCG (Define Clause Grammar) jelöléssel:
 

```

szám --> <számjegy> számmaradék.
számmaradék --> <számjegy> számmaradék | ".
számjegy --> "0" | "1".
      
```
- A definitív klóz nyelvtan (DCG):
  - egy általános nyelvtani formalizmus,
  - amely egyszerűen Prologra fordítható,
  - a legelső Prolog rendszer része (bár a szabványnak nem).

Deklaratív programozás. BME VIK, 2002. 6564 félév

(Logikai Programozás)

## A DCG szabályok lefordított alakja

Nyelvtani elemzés Prologban LP-255

- A korábbi DCG példa:
 

```

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

Deklaratív programozás. BME VIK, 2002. 6564 félév

(Logikai Programozás)

## 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 „lelemezhető” egy <szám>, marad L ha
% L0-ról lelemezhető egy <számjegy>, marad L1, és
% L1-ről lelemezhető egy <számmaradék>, marad L.
      
```
- Általánosan: az adott nem-terminálisnak megfelelő jelsorozatot „lelemezve” (lehagyva) egy L0 lista elejétől marad egy L lista.
- Terminális szintűmunk esetén egyetlen elemet kell leahagyni a listáról, erre szolgál a ‘C’/3 beépített eljárás. Definiója: ‘C’(L0, X, L) :- L0 = [X|L]. (A SICStus fordító a ‘C’/3 hívást ténylegesen a fenti egyenlőséggel helyettesíti.)
- A „lelemezés” tulajdonképpen akkumulációs folyamat, ahol az elemi akkumulációs lépés: egy terminális leahagyása a lista elejétől (‘C’/3).

Deklaratív programozás. BME VIK, 2002. 6564 félév

(Logikai Programozás)

## Vezérlési szerkezetek DCG szabályokban

Nyelvtani elemzés Prologban LP-256

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

```

% Lelemezhető számjegyek egy MAXIMÁLIS (esetleg üres) listája.
számmaradék -->
    ( számjegy -> számmaradék
    ; []
    ).
% Vigyázat: [] helyett true nem jó!
% Ugyanez vágóvával
számmaradék --> számjegy, !, számmaradék.
számmaradék --> [].
% Az utóbbi Prolog alakja:
számjegy(L0, L1), !, számmaradék(L1, L).
számmaradék(L0, L) :-
    L = L0.
      
```
- ?- számmaradék("102", L). => L = "2" ; no

Deklaratív programozás. BME VIK, 2002. 6564 félév

(Logikai Programozás)

## 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],
    {decimális_jegy_kódja(K)} .
% K a következő terminális
% Prolog hívás
% K egy számjegy kódja.
decimális_jegy_kódja(K) :-
    K >= 0'0, K < 0'9.
```

- A fenti DCG szabály Prolog megfeleltője:

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

Deklaratív programozás. BMÉ VIK, 2002. 6564 félév

(Logikai Programozás)

## A DCG nyelvtani szabályok szerkezete — összetoglalás

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

```
p(A,...) -->
    q0(A,...), ..., [X], qn(C,...), ...
    {cél}, ..., qn(D,...) .
p(A,...,L0,L):-
    q0(B,...,L0,L1), ..., 'C'(Ln-1, X, Ln), q(C,...,Ln,Ln+1),...,
    cél, ..., qn(D,...,Ln,L) .
```

Deklaratív programozás. BMÉ VIK, 2002. 6564 félév

(Logikai Programozás)

## Az elemző kiegészítése argumentumokkal

- Egy DCG szabály az elemzéssel párhuzamosan további (kimerő) argumentum(ok)ban felépítheti a kiellenzött dolog „jelentését”, pl. egy elemzési fát, vagy annak egy kiértékelését.

- Példa: szám elemzése és értékének kiszámítása:

```
% Lelemezhető egy Sz értékű decimális számjegy-sorozat
szám(Sz) --> számjegy(U), számmaradék(U, Sz) .
% Lelemezhető számjegyek egy esetleg üres lista, amelynek
% az eddigi leellenzött Sz0-val együtt vett értéke Sz.
számmaradék(Sz0, Sz) -->
    számjegy(U), !, {Sz1 is Sz0*10+U}, számmaradék(Sz1, Sz) .
számmaradék(Sz0, Sz0) --> [] .
```

```
% Lelemezhető egy U értékű számjegy.
számjegy(U) --> [K], {decimális_jegy_kódja(K), U is K-0'0} .
| ? - szám(Sz, "102 56", L) . => L = " 56", Sz = 102; no
```

- A számmaradékok DCG szabály Prolog alakja:

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

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

Deklaratív programozás. BMÉ VIK, 2002. 6564 félév

(Logikai Programozás)

## 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 lelemezhető 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 elemel!
```

- 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) --> [] .
...
```

Deklaratív programozás. BMÉ VIK, 2002. 6564 félév

(Logikai Programozás)

```

:- use_module(library(lists)).

% mondat(Alany, Áll, L0, L): L0-L kiemelezhető egy Alany alanyból és Áll
% állítmányból álló mondatká. 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 létege az Ige.
én_te("én", "vagyok").
én_te("te", "vagy").

% én_te_perm(Ki, Ige, Áll, L0, L): L0-L kiemelezhető egy Ki
% névmásból, Ige igealakból és Áll állítmányból álló mondatká.
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)
    ).

```

Deklaratív programozás. BME VIK, 2002. 6564 félév

(Logikai Programozás)

## Egy nagyobb DCG példa: „természetes” nyelvű beszélgetés

```

% :- 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, [])
    -> fejdolgoz(Mondás)
    ; write('Nem értem\n'), fail
    ),
    Mondás = un, !.

% menet(Mondás, L0, L): Az L0-L kiemelt 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").

```

Deklaratív programozás. BME VIK, 2002. 6564 félév

(Logikai Programozás)

## Példa: „természetes” nyelvű beszélgetés — szavak elemzése

```

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

% illik(Szó, 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 " ?" jektől)
betű(K) --> [K], {\+ member(K, " .?")}.

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

```

Deklaratív programozás. BME VIK, 2002. 6564 félév

(Logikai Programozás)

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

Nyelvtani elemzés Prologban IP-263

```

% :- dynamic tudom/2.
% fejdolgoz(Mondás): fejdolgozza a felhasználói érkező Mondás üzenetet.
fejdolgoz(un) :-
    write('Én is.\n').
fejdolgoz(kijelent(Alany, Áll)) :-
    assertz(tudom(Alany, Áll)),
    write('Fel fogtam.\n').
fejdolgoz(kérdez(Alany)) :-
    tudom(Alany, _), !,
    válasz(Alany).
fejdolgoz(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(_).

```

Deklaratív programozás. BME VIK, 2002. 6564 félév

(Logikai Programozás)

## Beszélgétes 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
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.

```

Deklaratív programozás. BMÉ VIK, 2002. őszi félév

(Logikai Programozás)

## 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|L1].

```
  - 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).

```

Deklaratív programozás. BMÉ VIK, 2002. őszi félév

(Logikai Programozás)

„Hagyományos” beépített eljárások LP-268

## Aritmetikai beépített eljárások

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

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

Függvény jelölések			
<code>abs/1</code>	<code>exp/1</code>	<code>floor/1</code>	<code>sign/1</code>
<code>atan/1</code>	<code>float/1</code>	<code>log/1</code>	<code>sin/1</code>
<code>ceiling/1</code>	<code>float_fractional_part/1</code>	<code>max/2</code>	<code>sqrt/1</code>
<code>cos/1</code>	<code>float_integer_part/1</code>	<code>round/1</code>	<code>truncate/1</code>

Deklaratív programozás. BMÉ VIK, 2002. őszi félév

(Logikai Programozás)

## „HAGYOMÁNYOS” BEÉPÍTETT ELJÁRÁSOK



## 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`,  $(= / 2)$  szerinti azonos elemek ismétlődését kiszűrve).
- Lista kulcs szerinti rendezése: `keysort(@L, ?S)`
  - Az `L` argumentum `KeyValuePair` alakú kifejezések listája.
  - Az eljárás jelentése: az `S` lista az `L` lista `KeyValuePair` szerinti szabványos (`@<` általi) rendezése, ismétlődéseket nem szűr.

Deklaratív programozás. BMÉ VIK, 2002. 656. oldal

(Logikai Programozás)

## Kifejezések kírása — felhasználó vezérelte formázás

- `print(@X)`: Alapértelmezésben azonos `writeln`-al. Ha a felhasználó definiál egy `portray/1` eljárást, akkor a rendszer minden a `print`-tel kinyomtatandó rész kifejezésre meghívja `portray`-t. Ennek sikere esetén feltehető, hogy a kírás megtörtént, megújítás esetén maga írja ki a rész kifejezést.
- A rendszer a `print` eljárást használja a változó-behelyettesítések és a nyomkövetés kírásárai
- `portray(@K1F)` (felhasználó által definiálható ún. *kampó eljárás*): Igaz, ha `K1F` kifejezést a Prolog rendszernek nem kell kírnia (és ekkor maga a `portray` kell, hogy elvégezze a kírását).
- 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]
	?

„Hagyományos” beépített eljárások IP-271

Deklaratív programozás. BMÉ VIK, 2002. 656. oldal

(Logikai Programozás)

## Kifejezések kírása

- `write(@X)`: Kírja `X`-et, ha szükséges operátorokat, zárójeleket használva.
- `writeln(@X)`: Mint `write(X)`, csak gondoskodik, hogy szükség esetén az atomok idézőjelek közé legyenek téve.
- `writeln_canonical(@X)`: Mint `writeln(X)`, csak operátorok nélkül, minden struktúra szabványos alakban jelenik meg.
- `writeln_term(@X, +Opciók)`: Az Opciók opciólista szerint kírja `X`-et.
- `Format(@Formatum, @Adatlista)`: A `Formatum`-nak megfelelő módon kírja `Adatlista`-t. A formázójelek alakja: `~(szám esetleg)\(formázójel)`.

?- write('Helló világ').	⇒ Helló világ
?- writeln('Helló világ').	⇒ 'Helló világ'
?- writeln_canonical('**_-'%).	⇒ -(*, '%')
?- writeln_canonical([1,2]).	⇒ ', '(1, '(2, [1])
?- writeln_term([1,2,3], [max_depth(2)]).	⇒ [1,2 ...]
?- format('X~s ---~3d s', [[0'j,0'6],3245]).	⇒ X=j6 --- 3.245 s

Deklaratív programozás. BMÉ VIK, 2002. 656. oldal

(Logikai Programozás)

## Karakterek kírása és beolvasása

- `put_code(+Kód)`: Kírja az adott kódi karaktert.
- `tab(+N)`: Kír `N` szökőti feléve, hogy `N > 0`.
- `nl`: Kír egy soremelést.
- `get_code(?Kód)`: Beolvass 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-től.
rd_line(L) :-
    peek_code(0'\n), !, get_code(_), L = [].
rd_line([C|_]) :-
    get_code(C), rd_line(L).
```

?- rd_line(L), tab(20), member(X, L), put_code(X), tab(1), fail ; nl.	Hel l o w o r l d !
---	---------------------

„Hagyományos” beépített eljárások IP-272

Deklaratív programozás. BMÉ VIK, 2002. 656. oldal

(Logikai Programozás)



## Példa: számbel olvasás

```
% számba(Szám) : a Szám szám következik az input-folyamban.
számba(Szám) :-
    számjegy(Érték),
    számba(Érték, Szám).

% Az eddig beolvasott Szám0-val együtt az input-folyamban következő
% szám értéke Szám.
számba(Szám0, Szám) :-
    számjegy(E), I,
    SzámI is Szám0*10+E,
    számba(SzámI, Szám).
számba(Szám, Szám).

% Érték értékű számjegy következik.
számjegy(Érték) :-
    peek_code(Kar),
    Kar >= 0'0, Kar = < 0'9,
    get_code(_),
    Érték is Kar - 0'0.

| ?- számba(X), get_code(_), számba(Y).
| : 123 456
    => X = 123, Y = 456
```

Deklaratív programozás. BMÉ VIK, 2002. 6.szá. félév

(Logikai Programozás)

## 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 ismerteket összes be- és kiviteli eljárásnak van egy eggyel több argumentumú változata, amelyek első argumentuma a csatorna. Ezek: `write/2`, `writeln/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`.

Deklaratív programozás. BMÉ VIK, 2002. 6.szá. félév

(Logikai Programozás)

## Kifejezések beolvasása

- `read(?Kif)`: Beolvass egy ponttal lezárt kifejezést és egyesíti Kif-fel. (File végénel Kif = `end_of_file`.)
- `read_term(?Kif, +Opciók)`: Mint `read/1`, de az Opciók opciólistát is figyelembe vesz.
- Példa — bocsúsínálta programbeolvasó:
 

<pre>consult_body :-     repeat,         read(Term),         ( Term = end_of_file -&gt; true         ; assertz(Term), fail         ),         !.</pre>	<pre>  ?- listing(lp/11). p(A) :-     q(A),     r(A). yes</pre>
--	---

Deklaratív programozás. BMÉ VIK, 2002. 6.szá. félév

(Logikai Programozás)

## Egy egyszerűbb be- és kiviteli szervezés: DECIO 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 állománynevet egyesíti FileNév-vel.
- `seen, told`: Lezárja a jelenlegi beviteli/kiviteli csatornát.
- Példák — nagyon egyszerű `consult` variánsok:
 

<pre>consult_decio_style(File) :-     seeing(Old), see(File),     repeat,         read(Term),         ( Term = end_of_file         -&gt; seen         ; assertz(Term), fail         ),         !,         see(Old).</pre>	<pre>consult_with_streams(File) :-     open(File, read, S),     repeat,         read(S, Term),         ( Term = end_of_file         -&gt; close(S)         ; assertz(Term), fail         ),         !.</pre>
---	--

Deklaratív programozás. BMÉ VIK, 2002. 6.szá. félév

(Logikai Programozás)

## 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 Minta-val.
  - Ha ez sikeres, meghívja a Hibaág-at.
  - Ellenkező esetben továbbadja a hiba-kifejezést, hogy a további körülvévő `catch` eljárások esetleg elkaphassák azt.

Deklaratív programozás: BME VIK, 2002. 6.szái félév

(Logikai Programozás)

## Programfejlesztési eljárások (folytatás)

- `statistics`: Különféle statisztikákat ír ki az aktuális kimenetre.
- `statistics(?FaJta, ?Érték)`: Érték a FaJta fájtajú mennyiség értéke.
  - Példa: `statistics(runtime, E) => E=[Tdiff, T], Tdiff az előző lekérdezés óra, T a rendszerindítás óta eltelt idő, ezredmásodpercben.`
- `break`: Egy új interakciós szintet hoz létre.
- `abort`, `halt`: Kilép a legkülső interakciós szintre ill. a Prolog rendszertől.
- `trace`: Elindítja az interaktív nyomkövetést.
- `debug, zip`: Elindítja a szelektív nyomkövetést, csak spion-pontokra áll meg. (A zip mód gyorsabb, de nem gyűjt annyit 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.

„Hagyományos” beépített eljárások LP-279

Deklaratív programozás: BME VIK, 2002. 6.szái félév

(Logikai Programozás)

## Programfejlesztési beépített eljárások (SICStus specifikusak)

- `set_prolog_flag(+Jelző, @Érték)`: Jelző értékét 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(:@File), [:@File, ...]`: Betölti a File(ok)at, interpretált alakban.
- `compile(:@File)`: Betölti a File(ok)at, lefordított alakot hozva létre.
- `listing`: Kijűz az összes interpretált eljárást az aktuális kimenetre.
- `listing(@EljárásSpec)`: Kijűz a megnevezett interpretált eljárásokat.
- Itt és később: EljárásSpec — név vagy funktor, esetleg modul-kvalifikációval ellátva, ill. ezek listája, pl. `listing(p), listing([m:q,p/1])`.

Deklaratív programozás: BME VIK, 2002. 6.szái félév

(Logikai Programozás)

## FEJLETTBBI NYELVI ÉS RENDSZERLEMEK

## 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 adatastruktúrákra, valamint hozzáférési algoritmusokat ezek kezelésére. Nehézkesebb, veszélyesebb, de jóval hatékonyabb mint az előző megoldás. Összetett adatok adásvételére is jó. (SWI, SICStus)

Deklaratív programozás. BMÉ VIK, 2002. 6564 félév

(Logikai Programozás)

Foljebbn nyelv i és rendszervelemek IP-283

## 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 `sp1fr` (link foreign resource) eszköz segítségével:
 

```
sp1fr ixkeys ixtest.pl +c ixkeys.c
```

Deklaratív programozás. BMÉ VIK, 2002. 6564 félév

(Logikai Programozás)

## Külső nyelvi interfész — példa

- A példa a `library` (load) megvalósításából származik.
- A C nyelven megírandó eljárás Prolog hívási alakja:
 

```
index_keys(+Spec, +Kif, -Kulcs, -Szám)
```
- A megírandó 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 *kvonathól* képzett lista. A kvonath lényegében az argumentum funktorra, azzal az eltéréssel, hogy a konstansok kivonata maga a konstans, struktúrák esetén pedig a struktúra neve és az ariása külön kerül a kvonath-listába.

Deklaratív programozás. BMÉ VIK, 2002. 6564 félév

(Logikai Programozás)

Foljebbn nyelv i és rendszervelemek IP-284

## Külső nyelvi interfész — a C kód (ixkeys.c állomány)

```
#include <stdio.h>
#define NA -1 /* not applicable */
#define NI -2 /* instantiateladness */
long ixkeys(SP_term_ref spec,
            SP_term_ref term, SP_term_ref list)
{
    unsigned long sname, tname, plus;
    int sarlity, tarlity, i;
    long ret = 0;
    SP_term_ref arg = SP_new_term_ref();
    tmp = SP_new_term_ref();
    SP_get_functor(spec, &sname, &sarlity);
    SP_get_functor(term, &tname, &tarlity);
    if (sname != tname || sarlity != tarlity)
        return NA;

    for (i = sarlity; 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, &tarlity);
                SP_put_integer(tmp, (long)tarlity);
                SP_cons_list(list, tmp, list);
                SP_put_atom(arg, tname);
                break;
            case SP_TYPE_LIST:
                SP_cons_list(list, arg, list); ++ret;
        }
    }
    return ret;
}
```

Deklaratív programozás. BMÉ VIK, 2002. 6564 félév

(Logikai Programozás)

## 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 eltarolja 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 értéket.
```

```
bb_delete(Kulcs, Érték)
```

```
Előhívja Érték-be a Kulcs értéket, majd kitérni.
```

Deklaratív programozás. BME VIK, 2002. őszi félév

(Logikai Programozás)

Fejletubb nyelvi és rendszervelemek LP-287

## Fejlett vezérlési lehetőségek SICStusban: Blokk-deklarációk

- Példa:

```
:- block P(-, ?, -, ?, ?).
```

Jelentése: ha az első és a harmadik argumentum is behelyettesíthetően változó (blokkolási feltétel), akkor a `p` hívás feltéggesszerűdik.

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).
```

Deklaratív programozás. BME VIK, 2002. őszi félév

(Logikai Programozás)

## Hasznos lehetőségek SICStus Prolog-ban (folytatás)

- Visszaléptethető módon változatható kifejezések

```
create_mutable(Adat, ValTklf)
Adat kezdőértékkel létrehoz egy új változatható kifejezést, ez lesz ValTklf. Adat nem lehet üres változó.
```

```
get_mutable(Adat, ValTklf)
```

```
Adat-ba előveszi ValTklf pillanatnyi értékét.
```

```
update_mutable(Adat, ValTklf)
```

A ValTklf változatható kifejezés új értéke Adat lesz. Ez a változtatás visszaléptéskor visszacsinalódik. Adat nem lehet üres változó.

- Takarító eljárás

```
call_cleanup(Hivas, Tiszto)
```

Meghívja `call` (Hivas)-t és ha az véglegesen befejezte futását, meghívja `Tiszto`-t. Egy eljárás akkor fejezte be véglegesen a futását, ha további alternatívák nélkül sikerült, meghívásait vagy kivételt dobott.

Deklaratív programozás. BME VIK, 2002. őszi félév

(Logikai Programozás)

Fejletubb nyelvi és rendszervelemek LP-288

## Blokk-deklarációk (folytatás)

- Generál-és-ellenőriz típusú programok gyorsítása

- általában nem hatékonyak (pl megírja\_1), mert túl sok visszaléptést használnak

- korutinszervezéssel a generáló és ellenőrző rész "automatikusan" összefésülhető

- ehhez az ellenőrző részt előre tenni és megfélelően blokkolni

- Korutinszervezésre építő programok

- Példa: egyszerűsített Hamming feladat

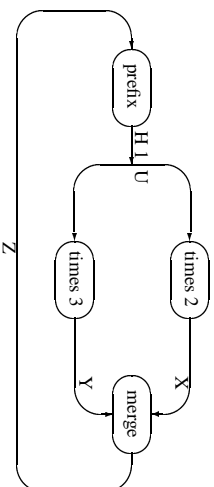
- keressük a  $2^i * 3^j$  ( $i \geq 1, j \geq 1$ ) alakú számok közül az első  $N$  darabot nagyság szerint rendezve.

- "stream-and-parallelism" közelítésmódot használva korutinszervezéssel egyszerűen lehet megoldani

Deklaratív programozás. BME VIK, 2002. őszi félév

(Logikai Programozás)

## 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([], _, []).
```

Deklaratív programozás. BMÉ VK, 2002. 564. fejev

(Logikai Programozás)

Folietumb nyelvű és rendszervelemek LP-291

## Korlatszerző eljárások

- freeze(X, Hivas)

Hivast felfüggeszti mindaddig, amíg X behelyettesíten változó.

- frozen(X, Hivas)

Az X változó miatt felfüggesztett hívás(ok)ot egyesít Hivas-sal.

- dif(X, Y)

X és Y nem egyesíthető. Mindaddig felfüggesztődik, amíg ez el nem dönthető.

- call\_residue(Hivas, Maradék)

Hivast végrehajtja, és ha a sikeres lefutás után maradnak felfüggesztett hívások, akkor azokat visszaszítja Maradékban. Pl.

```

| ?- call_residue(dif(X, f(Y)), Maradék).
    => Maradék = [[X]-(prolog:dif(X, f(Y)))]
| ?- call_residue(dif(X, f(Y)), X=f(Z)), Maradék.
    => X = f(Z), Maradék = [[Y,Z]-(prolog:dif(f(Z), f(Y)))]
  
```

Deklaratív programozás. BMÉ VK, 2002. 564. fejev

(Logikai Programozás)

## 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).
```

Deklaratív programozás. BMÉ VK, 2002. 564. fejev

(Logikai Programozás)

Folietumb nyelvű és rendszervelemek LP-292

## SICStus könyvtárak

- Könyvtár betöltése

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

- A legfontosabb könyvtárak

- arrays Logaritmusos elérési idejű kiterjeszhető tömbök megvalósítását tartalmazza.

- assoc AVL fák segítségével valósfija meg az „asszociációs listák”, azaz véges Prolog kifejezeshalmazokon definiált kiterjeszhető leképezések fogalmát.

- atts tetszőleges attributumokat enged a Prolog változókhoz rendelni, ezeket

- tárolórekeszként és a Prolog egyesítési mechanizmusának módosítására is engedni használni.

- heaps A bináris kazal (heap) fogalmát valósfija meg, amely főként prioritásos sorok (priority queue) megvalósítására használható.

- lists Biztosítja a listakezelő alapműveleteket.

- terms Különböző kiterjeszkezelő 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.

Deklaratív programozás. BMÉ VK, 2002. 564. fejev

(Logikai Programozás)

- random Egy véletelenszám-generátort tartalmaz.
- system Különféle operációsrendszer-szolgálatások elérését biztosítja.
- trees Az arrays könyvtárhoz hasonló, de nem-kiterjeszhető logaritmikus elérési idejű tömbfogalmat valósít meg, bináris fák segítségével (kicsit hatékonyabb mint az arrays könyvtár).
- ugraphs Irányított és irányítatlan gráf fogalmat valósít meg, élcimkék nélkül.
- wgraphs Olyan irányított és irányítatlan gráf fogalmat valósít meg, ahol minden él egy egérszértékű súllyal rendelkezik.
- sockets A socket-ek kezelésére szolgáló eljárásokat biztosít.
- linda/ client és linda/ server Linda-szerű processzorkommunikációs eszközöket ad.
- bdb Felhasználói által definiált többszörös indexelési lehetővé tevő, Prolog kifejezések lemezen való tárolására szolgáló adatbázis-rendszer.
- clpb Boolean-értékekre vonatkozó feltétel-megoldó (constraint solver).
- clpq és clpr Feltétel-megoldó a Q (racionális számok) ill. R (valós számok) tartományán.
- clpfd Végés tartományokra vonatkozó feltétel-megoldó (constraint solver).
- tcltk A Tcl/Tk nyelv és eszközkészlet elérését biztosítja.
- gauge Prolog programok a profilrozására szolgáló, a tcltk -n alapuló grafikus interfészsel rendelkező eszköz.

Deklaratív programozás. BME VIK, 2002. őszai félév

(Logikai Programozás)

- charso Karakter sorozatból olvasó ill. abba rő be- és kiviteli eljárások gyűjteménye.
- timeout Lehetőséget ad arra, hogy célok futási idejét korlátozzuk.
- xref A nyomkövetés és a program-analízis segítségével használható keresztreferencia készítő program.

## ÚJ IRÁNYZATOK A LOGIKAI PROGRAMOZÁSBAN

Deklaratív programozás. BME VIK, 2002. őszai félév

(Logikai Programozás)

### Új irányzatok a logikai programozásban — kitekintés

Új irányzatok a logikai programozásban LP-296

- Bevezetés a Logikai Programozásba c. jegyzet 6. fejezete:
  - Párhuzamos megvalósítások
  - Az Andorra-I rendszer rövid bemutatása
  - A Mercury nyagyhakékonyságú LP megvalósítás
  - CLP (Constraint Logic Programming)
- Az utolsó két témával foglalkozik a „**Nagyhakékonyságú logikai programozás**” c. választható tárgy (általában őszi félévben)
- Rövid ízelítőként áttekinjtük a korlát-logikai programozás (CLP) témakörét.
- Constraint = megszorítás, kényszer, korlátozás, korlát, ...
- A továbbiakban a „constraint” angol kifejezésre a „korlát” fordítást használjuk

Deklaratív programozás. BME VIK, 2002. őszai félév

(Logikai Programozás)







## Példák a SICStus clpbq könyvtárának használatára

- | ?- use\_module(library(clpbq)).
- | ?- {X=Y+4, Y=Z-1, Z=2\*X-9}.  
X = 6, Y = 2, Z = 3 ?  
% lineáris egyenlet
- | ?- {X+Y+9<4\*Z, 2\*X=X+2, 2\*X+4\*Z=36}.  
{X<29/5}, {Y= -2+2\*X}, {Z=9-1/2\*X} ?  
% egyenlőtlenség is lehet
- | ?- {(Y+X)\*(X+Y)/X = Y+Y/X+100}.  
{X=100-2\*Y} ?  
% lineárisá egyszerűsíthető
- | ?- {(Y+X)\*(X+Y) = Y+Y+100\*X}.  
clpbq:{2\*(X+Y)-100\*X+X^2=0} ?  
% Így már nem...
- | ?- {exp(X+Y+1, 2) = 3\*X\*X+Y\*Y}.  
clpbq:{1+2\*X+2\*(Y\*X)-2\*X^2+2\*Y=0} ?  
% nem lineáris...
- | ?- {exp(X+Y+1, 2) = 3\*X\*X+Y\*Y}, X=Y.  
X = -1/4, Y = -1/4 ?  
% Így már igen...
- | ?- {2 = exp(8, X)}.  
X = 1/3 ?  
% nem-lineárisak is megoldhatók

Deklaratív programozás. BMÉ VIK, 2002. 564. oldal

(Logikai Programozás)

## A SICStus clpb könyvtár

- Alapellemek:

- **Tartomány:** logikai értékek (1 és 0, igaz és hamis)
- **Függvények** (egyben korlát-relációk):
  - $\sim P$  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 (*kitáró vagy*).
  - $P := Q$  Ugyanaz mint  $\sim(P \# Q)$ .

- **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 műveletekkel felépített logikai kifejezés. Hozzáveszi *Kifejezést* a korlát-tárhoz.
- **labeling** (*Változók*). Behelyettesíti a *Változókat* 0 1 értékekre, úgy, hogy a tár teljesüljön. Visszalépéskor felsorolja az összes lehetséges értéket.

Deklaratív programozás. BMÉ VIK, 2002. 564. oldal

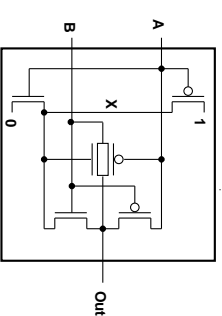
(Logikai Programozás)

## 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),
    p(0, A, X),
    p(B, A, Out),
    n(B, X, Out),
    p(A, B, Out),
    n(X, B, Out).
```



## A SICStus clpfd könyvtár

- A clpfd könyvtár alapelmei

- 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(Options, Változók) könyvtári eljárás segítségével.

Deklaratív programozás. BMÉ VIK, 2002. 564. oldal

(Logikai Programozás)

Deklaratív programozás. BMÉ VIK, 2002. 564. oldal

(Logikai Programozás)

## 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):-
    % 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 lefekt 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 sor távolságra levő királynők nem támadják egymást.
no_threat(X, Y, I) :- Y #\= X, Y #\= X-I, Y #\= X+I.
| ?- queens(4, Qs).
| ?- queens(4, Qs), Qs = [1|_].
| ?- queens(4, Qs), Qs = [1,_B,_C] in 3..4, _B in 2}\{4}, _C in 2..3 ?
no
| ?- queens(4, Qs), Qs = [2|_], labeling([], Qs).
Qs = [2,4,1,3] ?
```

Deklaratív programozás. BMÉ VIK, 2002. 654. félév

(Logikai Programozás)

## Lovagok és lókötékek – A megoldás elvei

Kortár-logikai programozás – rövid áttekintés LP-307

- Készítünk egy egyszerű formális nyelvet a bennszülöttek kijelentéseire, pl. Alfréd mondja Alfréd = lóköttő vagy Béla = lóköttő
- A bennszülöttek nevei (pl. Alfréd) Prolog változók, amelyek a Lovag vagy Lóköttő értéket veszik fel.
- A nyelv egyetlen alaprelációja 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.

Deklaratív programozás. BMÉ VIK, 2002. 654. félév

(Logikai Programozás)

## Egy példasor: Lovagok és lókötékek

- A feladat
  - Egy szigeten minden bennszülött lovag vagy lóköttő.
  - A lovagok mindig igazat mondanak.
  - A lóköttők mindig hazudnak.
  - Egy vagy több bennszülötnek saját magukra vonatkozó kijelentése alapján meg kell határozni a bennszülött típusát.
  - Példa: Találkozunk két bennszülöttel Alfréd-dal és Bélával. Alfréd azt mondja: van közöttünk lóköttő. Milyen típusú Alfréd és Béla.
  - Irodalom: Raymond Smullyan: Mi a címe ennek a könyvnek?, A hölgy és a tigris, Typotex kiadó.
  - További fejlesztés: a szigeten lehetnek normális emberek is, akik néha hazudnak, néha igazat mondanak.

Deklaratív programozás. BMÉ VIK, 2002. 654. félév

(Logikai Programozás)

## Lovagok és lókötékek: 1. változat (Prolog)

Kortár-logikai programozás – rövid áttekintés LP-308

```
:- 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 = Y, 1).
értéke(X = Y, 0) :-
    értéke(Állítás M, E) :-      különböző(X, Y).
    értéke(Lovag mondja M, E) :- értéke(M, E).
    értéke(lóköttő 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, lóköttő).
különböző(lóköttő, lovag).

| ?- Alfréd mondja Alfréd = lóköttő vagy Béla = lóköttő.
    Béla = lóköttő, Alfréd = lovag ? ; no

| ?- A mondja B = C.
    A = lovag, C = B ? ;
    A = lóköttő, B = lovag, C = lóköttő ? ;
    A = lóköttő, B = lóköttő, C = lovag ? ; no
```

Deklaratív programozás. BMÉ VIK, 2002. 654. félév

(Logikai Programozás)

## Lovagok és lókörtök: 2., CLP(B) változat

(A bennszilbíttek típusát numerikusan jelöljük: lovag → 1, lókörtő → 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 = 1, C = 1 ? ;
    A = 1, B = 0, C = 0 ? ; A = 1, B = 1, C = 1 ? ; no
```

Deklaratív programozás. BMÉ VIK. 2002. 564. oldal

(Logikai Programozás)

## Lovagok és lókörtök: 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
```

Deklaratív programozás. BMÉ VIK. 2002. 564. oldal

(Logikai Programozás)

## Lovagok, lókörtök (és normálisak): 4., CLP(FD) változat

Kortár-logikai programozás – rövid áttekintés LP-311

(A bennszilbíttek típusa: normális → 2, lovag → 1, lókörtő → 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/ProblemeK2w99sol.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
```

Deklaratív programozás. BMÉ VIK. 2002. 564. oldal

(Logikai Programozás)

## CLP rendszerek a nagyvilágban

Kortár-logikai programozás – rövid áttekintés LP-312

- 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, Svédó) — R/Q, FD, B
  - GNU Prolog (INRIA, Franciao.) — FD (C-re fordít)
  - Oz (DFKI, Németo) — kortár alapú elosztott funkcionális nyelv.
- Kommerciális rendszerek (a fentiek között)
  - ILOG, CHIP, Prolog III-IV, SICStus
  - a szakma óriása: ILOG
  - szakterület: CLP + vizualizációs eszközök + szabványalapú eszközök
  - felvásárolta az egyik vezető operációkutatási céget, a CPLEX-et
  - 400 munkatárs 7 országban, 55M USD éves bevétel, NASDAQ-on jegyzett

Deklaratív programozás. BMÉ VIK. 2002. 564. oldal

(Logikai Programozás)

## 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

