

Deklaratív programozás

Hanák Péter

hanak@inf.bme.hu

Irányítástechnika és Informatika Tanszék

Szeredi Péter

szeredi@cs.bme.hu

Számítástudományi és Információelméleti Tanszék

KÖVETELMÉNYEK, TUDNIVALÓK



Deklaratív programozás: tudnivalók

- Honlap, levelezési lista

- Honlap: `<http://dp.iit.bme.hu>`

- Levlista: `<http://www.iit.bme.hu/mailman/listinfo/dp-l>`.

A listatagoknak szóló levelet a `<dp-l@www.iit.bme.hu>` címre kell küldeni.

Csak a feliratkozottak levele jut el moderátori jóváhagyás nélkül a listatagokhoz.

- Jegyzet

- Szeredi Péter, Benkő Tamás: Deklaratív programozás. Bevezetés a logikai programozásba (1000 Ft)

- Elektronikus változata elérhető a honlapról (ps, pdf)

- A nyomtatott változat **KORLÁTOZOTT SZÁMBAN** megvásárolható a SZIT tanszék V2 épületbeli titkárságán a V2.104 szobában, Bazsó Lászlónénál, 10:30-12:00 (hétfő-péntek) és 13:30-15:30 (hétfő-csütörtök).

- Kellő számú további igény esetén megszervezzük az újrayomtatást.

Deklaratív programozás: tudnivalók (folyt.)

Fordító- és értelmezőprogramok

- SICStus Prolog — 4.1.2 verzió (licenz az ETS-en keresztül kérhető)
- Erlang (szabad szoftver)
- Mindkettő letölthető a honlapról (linux, Win95/98/NT)
- Webes Prolog gyakorló felület az ETS-ben (ld. honlap)
- Kézikönyvek HTML-, ill. PDF-változatban
- Más programok: SWI Prolog, Gnu Prolog
- emacs-szövegszerkesztő Erlang-, ill. Prolog-módban (linux, Win95/98/NT/XP/Vista/7)
- Eclipse fejlesztői környezet SICStushoz (béta-teszt): SPIDER 0.0.20

Deklaratív programozás: félévközi követelmények

Nagy házi feladat (NHF)

- Programozás mindkét nyelven (Prolog, Erlang)
- Mindenkinek önállóan kell kódolnia (programoznia)!
- Hatékony (időlimit!), jól dokumentált („kommentezett”) programok
- A két programhoz közös, 5–10 oldalas fejlesztői dokumentáció (TXT, $\text{T}_{\text{E}}\text{X}/\text{L}\text{A}\text{T}_{\text{E}}\text{X}$, HTML, PDF, PS; de nem DOC vagy RTF)
- Kiadás legkésőbb a 6. héten, a honlapon, letölthető keretprogrammal
- Beadás a 12. héten; elektronikus úton (ld. honlap)
- A beadáskor és a pontozáskor külön-külön tesztorozatot használunk (nehézségben hasonlókat, de nem azonosakat)
- Azok a programok, amelyek megoldják a tesztesetek 80%-át *létraversenyen* vesznek részt (hatékonyság, gyorsaság plusz pontokért)

Deklaratív programozás: félévközi követelmények (folyt.)

Nagy házi feladat (folyt.)

- A beadási határidőig többször is beadható, csak az utolsót értékeljük
- Pontozása mindkét nyelvből:
 - helyes (azaz jó eredményt időkorláton belül adó) futás esetén a 10 teszteset mindegyikére 0,5-0,5 pont, összesen max. 5 pont
 - a dokumentációra, a kód olvashatóságára, kommentezettségére max. 2,5 pont
 - tehát nyelvenként összesen max. 7,5 pont szerezhető
- A NHF súlya az osztályzatban: 15% (a 100 pontból 15)
- A megajánlott jegy előfeltétele, hogy a hallgató nagy házi feladata mindkét nyelvből bejusson a létraversenybe (minimum 80%-os teljesítmény)

Deklaratív programozás: félévközi követelmények (folyt.)

Kis házi feladatok (KHF)

- 3 feladat Prologból is, Erlang-ból is
- Beadás elektronikus úton (ld. honlap)
- Egy KHF beadása érvényes, ha minden tesztesetre lefut
- Kötelező a KHF-ek legalább 50%-ának érvényes beadása, és legalább egy érvényes KHF beadása mindkét nyelvből
- Minden feladat jó megoldásáért 1-1 jutalompont jár

Gyakorlatok

- Kéthetente 2 órás gyakorlatok
- Kötelező részvétel a gyakorlatok 70 %-án (pontosabban n gyakorlat esetén legalább $\lfloor 0.7n \rfloor$ gyakorlaton)
- További Prolog gyakorlási lehetőség az ETS rendszerben (gyakorló feladatok, lásd honlap)

Konzultációk

- Rendszeres konzultációs lehetőség

Deklaratív programozás: félévközi követelmények (folyt.)

Nagyzárthelyi, pótzárthelyi (NZH, PZH, PPZH)

- A zárthelyi kötelező, semmilyen jegyzet, segédlet nem használható!
- 40%-os szabály (nyelvenként a maximális részpontszám 40%-a kell az eredményességhez).
- Az NZH az órarendben előírt héten, a PZH az utolsó oktatási hetekben lesz
- A PPZH-ra indokolt esetben a pótlási időszakban egyetlen alkalommal adunk lehetőséget
- Az NZH anyaga az addig előadott tananyag.
- A PZH, ill. a PPZH anyaga azonos az NZH anyagával
- A zárthelyi súlya az osztályzatban: 15% (a 100 pontból 15)

Az aláírás megszerzésének feltételei (összefoglalás)

- Részvétel a gyakorlatok legalább 70%-án
- Zárthelyi sikeres megírása, azaz mindkét nyelvből legalább 40%-os eredmény elérése
- A 6 kis házi közül legalább 3 érvényes beadása úgy, hogy mindkét nyelvből legalább egy érvényes kis házi van

Deklaratív programozás: vizsga

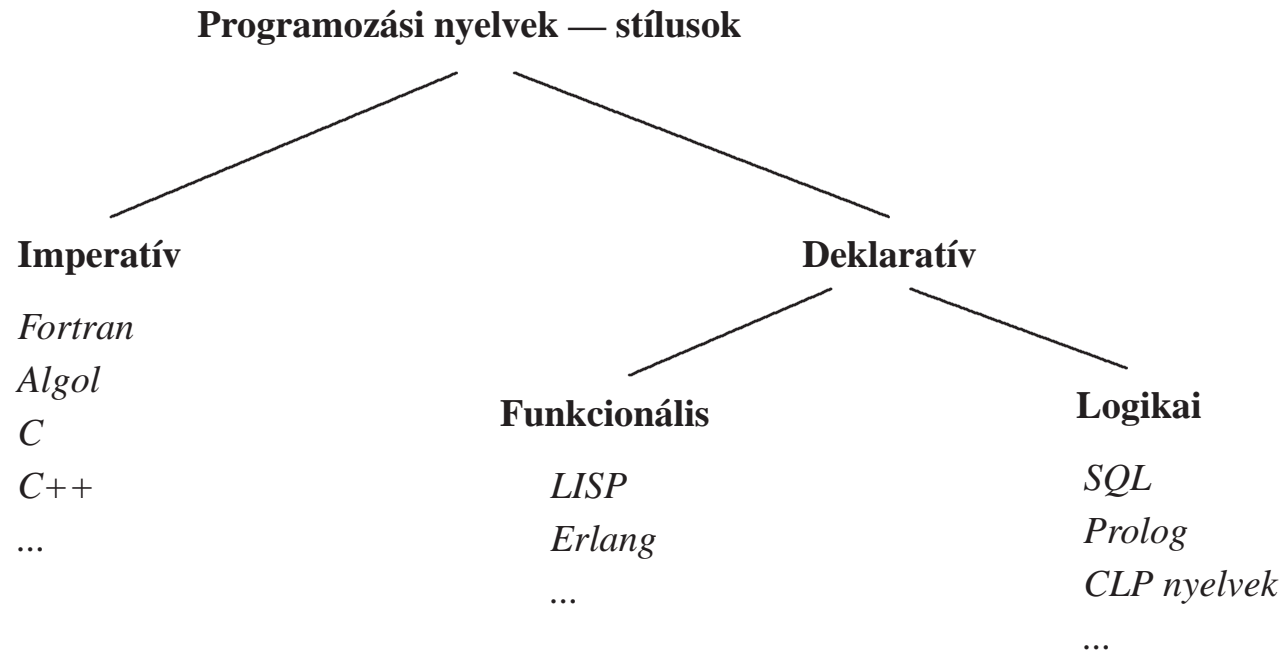
Vizsga

- Feltétel: aláírás a jelen félévben vagy korábban (de a TVSZ által előírt időn belül)
- A vizsga szóbeli, felkészülés írásban
- Prolog, Erlang: több kisebb feladat (programírás, -elemzés) kétszer 35 pontért
- A vizsgán szerzhető max. 70 ponthoz adjuk hozzá a félévközi munkával szerzett pontokat: ZH: max. 15 pont, NHF: max. 15 pont, továbbá a pluszpontokat (KHF, létraverseny)
- A vizsgán semmilyen jegyzet, segédlet nem használható, de lehet segítséget kérni
- 40%-os szabály (nyelvenként a max. részpontszám 40%-a kell az eredményességhez)
- Elővizsga a pótlási héten – minden, a tárgyból vizsgára bocsátható hallgató jelentkezhet
- A vizsgaidőszak első hetében (azaz még decemberben) is tartunk egy vizsgát
- Megajánlott vizsgajegy
 - Alapfeltételek: aláírás; NHF beadása; NHF „megvédése” az elővizsgán
 - A jó (4) jegy feltétele: a nagy házi feladat mindkét nyelvből bejut a létraversenybe
 - A jeles (5) jegy feltétele: legalább 40%-os eredmény a létraversenyen, mindkét nyelvből

BEVEZETÉS A LOGIKAI PROGRAMOZÁSBA



Programozási nyelvek osztályozása



Deklaratív programozási nyelvek

- A funkcionális nyelvek alapja a matematika függvényfogalma
- A logikai nyelvek alapja a matematika relációfogalma
- Közös tulajdonságaik
 - Deklaratív szemantika – a program jelentése egy matematikai állításként olvasható ki.
 - Deklaratív változó \equiv matematikai változó – *egy ismeretlen értéket jelöl, vö. egyszeres értékadás*
- Jelmondat
 - MIT és nem HOGYAN (WHAT rather than HOW): a *megoldás módja* helyett inkább a megoldandó *feladat leírását* kell megadni
 - A gyakorlatban mindkét szemponttal foglalkozni kell — kettős szemantika:
 - deklaratív szemantika — MIT (milyen feladatot) old meg a program;
 - procedurális szemantika — HOGYAN oldja meg a program a feladatot.

A logikai programozás alapgondolata

- Logikai programozás (LP):
 - Programozás a matematikai logika segítségével
 - egy logikai program nem más mint **logikai állítások halmaza**
 - egy logikai **program futása** nem más mint **következtetési folyamat**
 - De: a logikai következtetés óriási keresési tér bejárását jelenti
 - szorítsuk meg a logika nyelvét
 - válasszunk egyszerű, ember által is követhető következtetési algoritmusokat
 - Az LP máig legelterjedtebb megvalósítása a **Prolog = Programozás logikában (Programming in logic)**
 - az elsőrendű logika egy erősen megszorított résznyelve az ún. **definit-** vagy **Horn-klózik** nyelve,
 - végrehajtási mechanizmusa: **mintaillesztéses** eljáráshíváson alapuló **visszalépéses** keresés.

Az előadás LP részének áttekintése

- **1. blokk:** A Prolog nyelv alapjai
 - Logikai háttér
 - Szintaxis
 - Végrehajtási mechanizmus
- **2. blokk:** Prolog programozási módszerek
 - A legfontosabb beépített eljárások
 - Fejlettebb nyelvi és rendszerelemek
- Kitekintés: Új irányzatok a logikai programozásban

A Prolog/LP rövid történeti áttekintése

- 1960-as évek Első 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 forgalomba kerülő Prolog megvalósítás
- 1983 Egy új fordítási modell és absztrakt Prolog gép (WAM) megjelenése (D H D Warren)
- 1986 Prolog szabványosítás kezdete
- 1987–89 Új logikai programozási nyelvek (CLP, Gödel stb.)
- 1990–... Prolog megjelenése párhuzamos számítógépeken
Nagyhatékonyságú Prolog fordítóprogramok
.....

Információk a logikai programozásról

- A legfontosabb Prolog megvalósítások:

- SWI Prolog: <http://www.swi-prolog.org/>

- SICStus Prolog: <http://www.sics.se/sicstus>

- GNU Prolog: <http://pauillac.inria.fr/~diaz/gnu-prolog/>

- Hálózati információforrások:

- The WWW Virtual Library: Logic Programming:

<http://www.afm.sbu.ac.uk/logic-prog>

- CMU Prolog Repository:

(a <http://www.cs.cmu.edu/afs/cs/project/ai-repository/ai/lang/prolog/> címen belül)

- Főlap: [0.html](#)

- Prolog FAQ: [faq/prolog.faq](#)

- Prolog Resource Guide: [faq/prg_1.faq](#), [faq/prg_2.faq](#)

Magyar nyelvű Prolog irodalom

Farkas Zsuzsa, Futó Iván, Langer Tamás, Szeredi Péter:

Az MProlog programozási nyelv.

Műszaki Könyvkiadó, 1989

jó bevezetés, sajnos az MProlog beépített eljárásai nem szabványosak.

Márkus Zsuzsa: Prologban programozni könnyű.

Novotrade, 1988

mint fent

Futó Iván (szerk.): Mesterséges intelligencia. (9.2 fejezet, Szeredi Péter)

Aula Kiadó, 1999

csak egy rövid fejezet a Prologról

Peter Flach: Logikai Programozás. Az intelligens következtetés példákon keresztül.

Panem — John Wiley & Sons, 2001

jó áttekintés, inkább elméleti érdeklődésű olvasók számára

English Textbooks on Prolog

- Logic, Programming and Prolog, 2nd Ed., by Ulf Nilsson and Jan Maluszynski, Previously published by John Wiley & Sons Ltd. (1995)
Downloadable as a pdf file from <http://www.ida.liu.se/~ulfni/lpp>
- Prolog Programming for Artificial Intelligence, 3rd Ed., Ivan Bratko, Longman, Paperback - March 2000
- The Art of PROLOG: Advanced Programming Techniques, Leon Sterling, Ehud Shapiro, The MIT Press, Paperback - April 1994
- Programming in PROLOG: Using the ISO Standard, C.S. Mellish, W.F. Clocksin, Springer-Verlag Berlin, Paperback - July 2003

PROLOG: EGY KIS GYAKORLATI BEMUTATÁS



Példafeladatok

- Szimbolikus feldolgozás: deriválás
- Adatstruktúrák: bináris fák
- Aritmetika: faktoriális
- Adatbáziskezelés: családi kapcsolatok
- Logikai feladványok: lovagok és lóköttők

Klasszikus szimbolikuskifejezés-feldolgozás: deriválás

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

```
% deriv(Kif, D): Kif-nek az x szerinti deriváltja D.
```

```
deriv(x, 1).
```

```
deriv(C, 0) :- number(C).
```

```
deriv(U+V, DU+DV) :- deriv(U, DU), deriv(V, DV).
```

```
deriv(U-V, DU-DV) :- deriv(U, DU), deriv(V, DV).
```

```
deriv(U*V, DU*V + U*DV) :- deriv(U, DU), deriv(V, DV).
```

```
| ?- deriv(x*x+x, D).
```

```
⇒ D = 1*x+x*1+1 ? ; no
```

```
| ?- deriv((x+1)*(x+1), D).
```

```
⇒ D = (1+0)*(x+1)+(x+1)*(1+0) ? ; no
```

```
| ?- deriv(I, 1*x+x*1+1).
```

```
⇒ I = x*x+x ? ; no
```

```
| ?- deriv(I, 0).
```

```
⇒ no
```

A Prolog végrehajtási mechanizmusa dióhéjban

- A Prolog eljárásos szemléletben
 - Egy eljárás: azon klózik összesége, amelyek fejének neve és argumentumszáma azonos.
 - Egy klóz: $Fej :- Törzs$, ahol $Törzs$ egy célsorozat
 - Egy célsorozat: C_1, \dots, C_n , célok (eljáráshívások) sorozata, $n \geq 0$
- Végrehajtás: adott egy program és egy futtatandó célsorozat
 - Redukciós lépés:
 - a célsorozat *első* tagjához keresünk egy vele *egyesíthető* klózfejet,
 - az egyesítéshez szükséges *változó-behelyettesítéseket* elvégezzük,
 - az első célt helyettesítjük az adott klóz törzsével
 - Egyesítés: két Prolog kifejezés azonos alakra hozása változók behelyettesítésével, a lehető legáltalánosabb módon
 - Keresés:
 - a redukciós lépésben a klózikat a felírás sorrendjében (felülről lefele) nézzük végig,
 - ha egy cél több klózfejjel is egyesíthető, akkor a Prolog *minden* lehetséges redukciós lépést megpróbál (meghiúsulás, visszalépés esetén)

A Prolog adatfogalma, a Prolog kifejezés

- konstans (*atomic*)
 - számkonstans (*number*) — egész vagy lebegőpontos, pl. `1`, `-2.3`, `3.0e10`
 - névkonstans (*atom*), pl. `'István'`, `szuloje`, `+`, `-`, `<`, `sum_tree`
- összetett- vagy struktúra-kifejezés (*compound*)
 - ún. kanonikus alak: $\langle \text{struktúranév} \rangle (\langle \text{arg}_1 \rangle, \dots)$
 - a $\langle \text{struktúranév} \rangle$ egy névkonstans, az $\langle \text{arg}_i \rangle$ argumentumok tetszőleges Prolog kifejezések
 - példák: `leaf(1)`, `person(william,smith,2003,1,22)`, `<(X,Y)`, `is(X, +(Y,1))`
 - szintaktikus „édesítőszerek”, pl. operátorok: `X is Y+1` \equiv `is(X, +(Y,1))`
- változó (*var*)
 - pl. `X`, `Szulo`, `x2`, `_valt`, `_`, `_123`
 - a változó alaphelyzetben behelyettesítetlen, értékkel nem bír, az egyesítés (mintaillesztés) művelete során egy tetszőleges Prolog kifejezést vehet fel értékül (akár egy másik változót)

Adatstruktúrák Prologban — példa

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

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

```
% Adattípus-leírás Prologban
% (ún. Mercury jelölés):

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


Bináris fák összegzése

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

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

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

Bináris fák összegzése

- Prolog példafutás

```
% sicstus
SICStus 4.1.2 (x86-linux-glibc2.7): Wed Apr 28 22:42:37 CEST 2010
Licensed to BUTE DP course
| ?- consult(tree).
% consulting /home/szeredi/peldak/tree.pl...
% consulted /home/szeredi/peldak/tree.pl in module user, 0 msec 704 bytes
yes
| ?- tree_sum(node(leaf(5),
                node(leaf(3), leaf(2))), Sum).
Sum = 10 ? ;
no
| ?- tree_sum(Tree, 10).
Tree = leaf(10) ? ;
! Instantiation error in argument 2 of is/2
! goal: 10 is _73+_74
| ?- halt.
%
```

- A hiba oka: a beépített aritmetika egyirányú: a `10 is S1+S2` hívás hibát jelez!

Peano aritmetika — összeadás (kiegészítő anyag)

- A természetes számok halmazán az összeadást definiálhatjuk a Peano axiómákkal ha a számokat az $s(x)$ „rákövetkező” függvény segítségével ábrázoljuk:

$1 = s(0)$, $2 = s(s(0))$, $3 = s(s(s(0)))$, ... (Peano ábrázolás).

`% plus(X, Y, Z): X és Y összege Z (X, Y, Z Peano ábrázolású).`

`plus(0, X, X).`

`% 0+X = X.`

`plus(s(X), Y, s(Z)) :-`

`plus(X, Y, Z).`

`% s(X)+Y = s(X+Y).`

- A `plus` predikátum több irányban is használható:

| `?- plus(s(0), s(s(0)), Z). Z = s(s(s(0))) ? ; no % 1+2 = 3`

| `?- plus(s(0), Y, s(s(s(0)))). Y = s(s(0)) ? ; no % 3-1 = 2`

| `?- plus(X, Y, s(s(0))). X = 0, Y = s(s(0)) ? ; % 2 = 0+2`

`X = s(0), Y = s(0) ? ; % 2 = 1+1`

`X = s(s(0)), Y = 0 ? ; % 2 = 2+0`

`no`

| `?-`

Adott összegű fák építése (kiegészítő anyag)

- Adott összegű fát építő eljárás Peano aritmetikával:

```
tree_sum(leaf(Value), Value).
tree_sum(node(Left, Right), S) :-
    plus(S1, S2, S),
    S1 \= 0, S2 \= 0,           % X \= Y beépített eljárás, jelentése:
                                % X és Y nem egyesíthető
                                % A 0-t kizárjuk, mert különben  $\infty$  sok megoldás van.
    tree_sum(Left, S1),
    tree_sum(Right, S2).
```

- Az eljárás futása:

```
| ?- tree_sum(Tree, s(s(s(0)))).
Tree = leaf(s(s(s(0))) ? ;           % 3
Tree = node(leaf(s(0)),leaf(s(s(0)))) ? ;           % (1+2)
Tree = node(leaf(s(0)),node(leaf(s(0)),leaf(s(0)))) ? ;           % (1+(1+1))
Tree = node(leaf(s(s(0))),leaf(s(0))) ? ;           % (2+1)
Tree = node(node(leaf(s(0)),leaf(s(0))),leaf(s(0))) ? ;           % ((1+1)+1)
no
```

Néhány beépített predikátum

- Kifejezések egyesítése: $x = y$: az x és y **szimbolikus** kifejezések változók behelyettesítésével azonos alakra hozhatók (és el is végzi a behelyettesítéseket).
- Kifejezések nem-egyesíthetősége: $x \neq y$: az x és y kifejezések nem egyesíthetőek.
- Aritmetikai predikátumok
 - $x \text{ is } Kif$: $A \text{ Kif}$ **aritmetikai** kifejezést kiértékeli és **értékét** egyesíti x -szel.
 - $Kif1 < Kif2$, $Kif1 <= Kif2$, $Kif1 > Kif2$, $Kif1 >= Kif2$, $Kif1 == Kif2$, $Kif1 \neq Kif2$:
 $A \text{ Kif1}$ és $Kif2$ aritmetikai kifejezések értéke a megadott relációban van egymással
 ($==$ jelentése: aritmetikai egyenlőség, \neq jelentése aritmetikai nem-egyenlőség).
 - Ha Kif , $Kif1$, $Kif2$ valamelyike nem **tömör** (változómentes) aritmetikai kifejezés \Rightarrow hiba.
 - Legfontosabb aritmetikai operátorok: $+$, $-$, $*$, $/$, rem , $//$ (egész-osztás)
- Kiíró predikátumok
 - $\text{write}(x)$: Az x Prolog kifejezést kiírja.
 - nl : Kiír egy újsort.
- Egyéb predikátumok
 - true , fail : Mindig sikerül ill. mindig megghiúsul.
 - trace , notrace : A (teljes) nyomkövetést be- ill. kikapcsolja.

Programfejlesztési beépített predikátumok

- `consult(File)` vagy `[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 kilistázza.
- `compile(File)`: A `File` állományban levő programot beolvassa, lefordítja.
- A lefordított alak gyorsabb, de nem listázható, **kicsit** kevésbé pontosan nyomkövethető.
- `halt`: A Prolog rendszer befejezi működését.

```
> sicstus
SICStus 4.1.2 (x86-linux-glibc2.7): Wed Apr 28 22:42:37 CEST 2010
| ?- consult(deriv).
% consulted /home/user/szulok.pl in module user, 0 msec 376 bytes
yes
| ?- deriv(x*x+x, D).
D = 1*x+x*1+1 ? ;
no
| ?- listing(deriv).
(...)
yes
| ?- halt.
>
```

Aritmetika Prologban – faktoriális

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

„Adatbáziskezelés” Prologban: a családi kapcsolatok példája

- Adatok

Adottak gyerek–szülő kapcsolatra vonatkozó állítások, pl.

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

- A feladat:

- Definiálandó az unoka–nagyözülő kapcsolat, pl. keressük egy adott személy nagyözüleit.

A nagyszülő feladat — Prolog megoldás

```

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

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

```

```

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

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

```

A Prolog és az adatbáziskezelés

- Miben különbözik a Prolog egy adatbáziskezelőtől
- Mivel több?
 - rekurzió
 - összetett adatszerkezetek
- De: a Prolog egy programozási nyelv
 - pl. nem optimalizálja a részkérdések sorrendjét

Logikai feladvány: lovagok és lóköltők

- A feladat
 - Egy szigeten minden bennszülött lovag vagy lóköltő.
 - A lovagok mindig igazat mondanak.
 - A lóköltők mindig hazudnak.
 - Egy vagy több bennszülöttnak saját magukra vonatkozó kijelentése alapján meg kell határozni a bennszülött típusát.
 - Példa: Találkozunk két bennszülöttel A-val és B-vel. A azt mondja: van köztünk lóköltő. Milyen típusú A és B.
 - Irodalom: Raymond Smullyan: Mi a címe ennek a könyvnek?, A hölgy és a tigris, Typotex kiadó.
 - Továbbfejlesztés: a szigeten lehetnek normális emberek is, akik néha hazudnak, néha igazat mondanak.

Lovag-lóköttő feladványok megoldása Prolog nyelven

● A program:

```
:- op(950, xfy, mondja).
:- op(900, yfx, vagy).
:- op(700, xfx, az).

% A mondja M: Az A bennszülött mondja az M mondatot.
lókötő mondja M :-      értéke(M, 0).
lovag mondja M :-      értéke(M, 1).

% értéke(M, Érték): Az M mondat igazságértéke Érték (1 = igaz, 0 = hamis).
értéke(X az X, 1).
értéke(X az Y, 0) :-
    különböző(X, Y).
értéke(M1 vagy M2, E) :-
    értéke(M1, E1), értéke(M2, E2), E is E1 E2.

% különböző(A, B): A és B különböző típusú bennszülöttek.
különböző(lovag, lókötő).
különböző(lókötő, lovag).
```

● Futás:

```
| ?- A mondja A az lókötő vagy B az lókötő.
A = lovag, B = lókötő ? ; no
```

A PROLOG NYELV KÖZELÍTŐ SZINTAXISA



Predikátumok, klózek

● Példa:

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

● Szintaxis:

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

Prolog programok formázása

- Programok javasolt formázása:
 - Az egy predikátumhoz tartozó klózok legyenek egymás mellett a programban, közéjük ne tegyünk üres sort. A predikátumokat válasszuk el üres sorokkal.
 - A klózfejet írjuk sor elejére, minden célt lehetőleg külön sorba, néhány szóközzel beljebb kezdve

Prolog kifejezések

● Példa — egy klózfej mint kifejezés:

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

● Szintaxis:

⟨ kifejezés ⟩	::=	⟨ változó ⟩	{Nincs funktora}
		⟨ konstans ⟩	{Funktora: ⟨ konstans ⟩ / 0}
		⟨ összetett kifejezés ⟩	{Funktora: ⟨ struktúranév ⟩ / ⟨ arg.szám ⟩}
		⟨ egyéb kifejezés ⟩	{Operátoros, lista, zárójeles, ld. később}
⟨ konstans ⟩	::=	⟨ névkonstans ⟩	
		⟨ számkonstans ⟩	
⟨ számkonstans ⟩	::=	⟨ egész szám ⟩	
		⟨ lebegőpontos szám ⟩	
⟨ összetett kifejezés ⟩	::=	⟨ struktúranév ⟩ (⟨ argumentum ⟩, ...)	
⟨ struktúranév ⟩	::=	⟨ névkonstans ⟩	
⟨ argumentum ⟩	::=	⟨ kifejezés ⟩	

Lexikai elemek

● Példák:

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

● Szintaxis:

```
⟨ változó ⟩      ::= ⟨ nagybetű ⟩⟨ alfanumerikus jel ⟩...|
                  _⟨ alfanumerikus jel ⟩...
⟨ névkonstans ⟩ ::= '⟨ idézett karakter kar ⟩... ' |
                  ⟨ kisbetű ⟩⟨ alfanumerikus jel ⟩...|
                  ⟨ tapadó jel ⟩...| ! | ; | [ ] | { }
⟨ egész szám ⟩   ::= {előjeles vagy előjeltelen számjegysorozat}
⟨ lebegőpontos szám ⟩ ::= {belsejében tizedespontot tartalmazó
                           számjegysorozat esetleges exponenssel}
⟨ idézett karakter ⟩ ::= {tetszőleges nem ' és nem \ karakter} | \⟨ escape szekvencia ⟩
⟨ alfanumerikus jel ⟩ ::= ⟨ kisbetű ⟩ | ⟨ nagybetű ⟩ | ⟨ számjegy ⟩ | _
⟨ tapadó jel ⟩   ::= + | - | * | / | \ | $ | ^ | < | > | = | ` | ~ | : | . | ? | @ | # | &
```

LISTA, MINT SZINTAKTIKUS „ÉDESÍTŐSZER”

A Prolog lista-fogalma

● A Prolog lista

- Az üres lista a `[]` névkonstans. A nem-üres lista `'.'` (`Fej, Farok`) struktúra ahol
 - `Fej` a lista feje (első eleme), míg
 - `Farok` a lista farka, azaz a fennmaradó elemekből álló lista.
- A listák írhatók egyszerűsített alakban („szintaktikus édesítés”).
- Megvalósításuk optimalizált, időben és helyben is hatékonyabb, mint a „közönséges” struktúráké.

● Példa

```
számlista.(E,L) :-
    number(E), számlista(L).
számlista([]).

| ?- listing(számlista).
számlista([A|B]) :-
    number(A),
    számlista(B).
számlista([]).

| ?- számlista([1,2]).      % [1,2] == .(1,.(2,[])) == [1|[2|[]]]
    yes
| ?- számlista([1,a,f(2)]).
    no
```

Listák írásmódjai

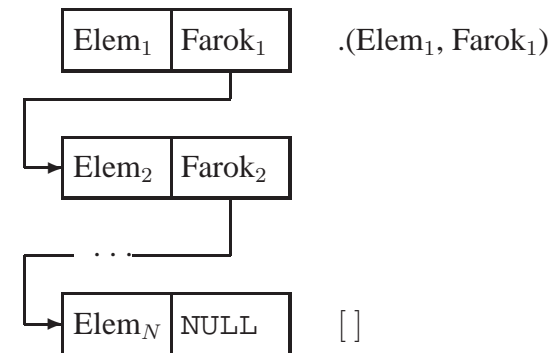
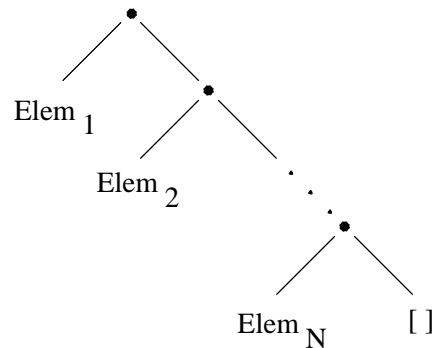
- Egy N elemű lista lehetséges írásmódjai:

- alapstruktúra-alak: $.(Elem_1, .(Elem_2, \dots, .(Elem_N, []) \dots))$

- ekvivalens lista-alak: $[Elem_1, Elem_2, \dots, Elem_N]$

- kevésbe kényelmes ekvivalens alak: $[Elem_1 | [Elem_2 | \dots | [Elem_N | []] \dots]]$

- A listák fastruktúra alakja és megvalósítása



Listák jelölése — szintaktikus édesítőszer

• az alapvető édesítés: $[Fej|Farok] \equiv .(Fej, Farok)$

• N -szeri alkalmazás kevesebb zárójellel:

$$[Elem_1, Elem_2, \dots, Elem_N | Farok] \equiv [Elem_1 | [Elem_2 | \dots | [Elem_N | Farok] \dots]]$$

• Ha a farok $[]$: $[Elem_1, Elem_2, \dots, Elem_N] \equiv [Elem_1, Elem_2, \dots, Elem_N | []]$

| ?- $[1, 2] = [X|Y].$ $\Rightarrow X = 1, Y = [2] ?$

| ?- $[1, 2] = [X, Y].$ $\Rightarrow X = 1, Y = 2 ?$

| ?- $[1, 2, 3] = [X|Y].$ $\Rightarrow X = 1, Y = [2, 3] ?$

| ?- $[1, 2, 3] = [X, Y].$ \Rightarrow no

| ?- $[1, 2, 3, 4] = [X, Y|Z].$ $\Rightarrow X = 1, Y = 2, Z = [3, 4] ?$

| ?- $L = [1|_], L = [_ , 2|_].$ $\Rightarrow L = [1, 2|_A] ?$ % nyílt végű

| ?- $L = .(1, [2, 3|[]]).$ $\Rightarrow L = [1, 2, 3] ?$

| ?- $L = [1, 2|. (3, [])].$ $\Rightarrow L = [1, 2, 3] ?$

| ?- $[X|[3-Y/X|Y]] = .(A, [A-B, 6]).$ $\Rightarrow A=3, B=[6]/3, X=3, Y=[6] ?$

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

- A logikai változó fogalma:
 - kifejezésként, kifejezésben egyaránt előfordulhat, vö. a változókat a (lista) mintákban.
 - két változó azonossá tehető (azaz egyesíthető): pl. két azonos változó egy kifejezésben.
 - a változó „teljes jogú” állampolgár a (rész)kifejezések világában
- Tömör (ground) kifejezés: változót nem tartalmazó kifejezés
- Minta: egy általában nem tömör kifejezés, mindazon kifejezéseket „képviseli”, amelyek belőle változó-behelyettesítéssel előállnak.
- Lista-minta: listát (is) képviselő minta.
- Nyílt végű lista: olyan lista-minta, amely bármilyen hosszú listát is képvisel.
- Zárt végű lista: olyan lista(-minta), amely egyféle hosszú listát képvisel.

Zárt végű	Milyen listákat képvisel		Nyílt végű	Milyen listákat képvisel
[X]	egyelemű		X	tetszőleges
[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 , . . .

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: $L3 = L1 \oplus L2$) — két megoldás:

```
append0([], L2, L) :- L = L2.
```

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

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

```
> append0([1,2,3],[4],A)
(2) > append0([2,3],[4],B), A=[1|B]
(2) > append0([3],[4],C), B=[2|C], A=[1|B]
(2) > append0([], [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] ?
```

```
append([], L, L).
```

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

```
    append(L1, L2, L3).
```

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

- AZ `append0/append(L1, ...)` komplexitása: futási ideje arányos `L1` hosszával.
- Miért jobb az `append/3` mint az `append0/3`?
 - `append/3` **jobbrekurzív**, ciklussal ekvivalens (nem fogyaszt vermet)
 - `append([1, ..., 1000], [0], [2, ...])` azonnal, `append0(...)` 1000 lépésben hiúsul meg
 - `append/3` használható szétszedésre is (lásd később), míg `append0/3` nem.

Lista építése *előlről* — nyílt végű listákkal

- Az append eljárás már az első redukciónál felépíti az eredmény fejét!
(az eredményparaméter egy lista-minta lesz, a farok még ismeretlen, vö. logikai változó)

```
append([], L, L).
append([X|L1], L2, [X|L3]) :-      append(L1, L2, L3).
| ?- append([1,2,3], [4], Ered) => Ered = [1|A], append([2,3], [4], A)
```

- Haladó nyomkövetési lehetőségek ennek demonstrálására
 - `library(debugger_examples)` — példák a nyomkövető programozására, új parancsokra
 - új parancs: ‘N <név>’ — fókuszált argumentum elnevezése
 - szabványos parancs: ‘^ <argszám>’ — adott argumentumra fókuszálás
 - új parancs: ‘P [<név>]’ — adott nevű (ill összes) kifejezés kiírása

```
| ?- use_module(library(debugger_examples)).
| ?- trace, append([1,2,3],[4,5,6],A).
      1      1 Call: append([1,2,3],[4,5,6],_543) ? ^ 3
      1      1 Call: ^3 _543 ? N Ered
      1      1 Call: ^3 _543 ? P                => Ered = _543
      2      2 Call: append([2,3],[4,5,6],_2700) ? P => Ered = [1|_2700]
      3      3 Call: append([3],[4,5,6],_3625) ? P  => Ered = [1,2|_3625]
      4      4 Call: append([], [4,5,6],_4550) ? P  => Ered = [1,2,3|_4550]
      4      4 Exit: append([], [4,5,6], [4,5,6]) ? P => Ered = [1,2,3,4,5,6]
      3      3 Exit: append([3],[4,5,6],[3,4,5,6]) ?
      2      2 Exit: append([2,3],[4,5,6],[2,3,4,5,6]) ?
      1      1 Exit: append([1,2,3],[4,5,6],[1,2,3,4,5,6]) ?
=> A = [1,2,3,4,5,6] ? ; no
```


Listák megfordítása

- Naív (négyzetes lépésszámú) megoldás

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

- Lineáris lépésszámú megoldás

```
% reverse(R, L): Az R lista az L megfordítása.  
reverse(R, L) :- revapp(L, [], R).  
  
% revapp(L1, L2, R): L1 megfordítását L2 elé fűzve kapjuk R-t.  
revapp([], R, R).  
revapp([X|L1], L2, R) :-  
    revapp(L1, [X|L2], R).
```

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

- A könyvtár betöltése:

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

append és revapp — listák gyűjtési iránya

● Prolog megvalósítás

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

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

● C++ megvalósítás

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

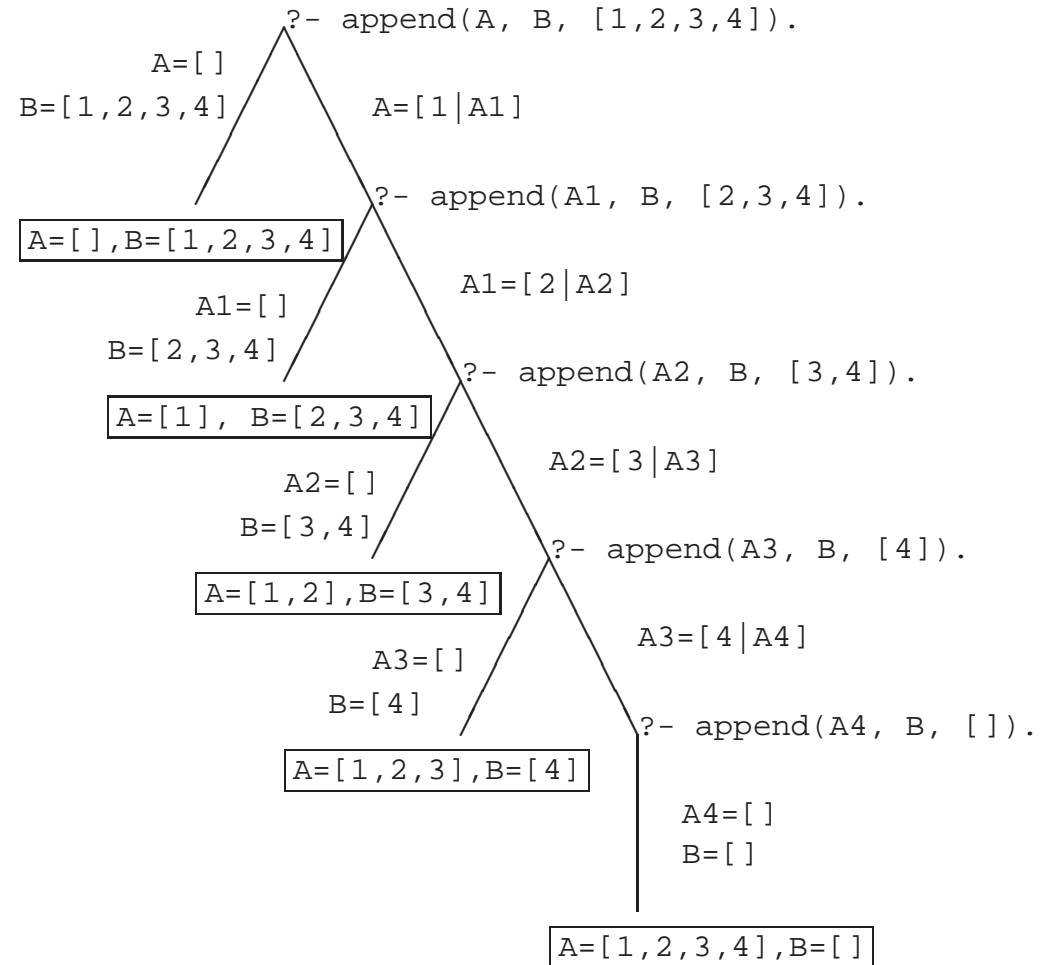
```
list append(list list1, list list2)
{ list list3, *lp = &list3;
  for (list p=list1; p; p=p->next)
  { list newl = new link(p->elem);
    *lp = newl; lp = &newl->next;
  }
  *lp = list2;
  return list3;
}
```

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

Listák szétbontása az append/3 segítségével

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

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



Variációk appendre 1. — Három lista összefűzése

- Az `append/3` keresési tere **véges**, ha első és harmadik argumentuma közül legalább az egyik zárt végű lista.

- `append(L1, L2, L3, L123): L1 ⊕ L2 ⊕ L3 = L123`

```
append(L1, L2, L3, L123) :-
    append(L1, L2, L12), append(L12, L3, L123).
```

- Nem hatékony, pl.: `append([1, ..., 100], [1, 2, 3], [1], L)` 103 helyett 203 lépés!

- Szétszedésre nem alkalmas — végtelen választási pontot hoz létre

- Szétszedésre is alkalmas, hatékony változat

```
% L1 ⊕ L2 ⊕ L3 = L123, ahol vagy L1 és L2, vagy L123 adott (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] ?
```

- Az `L3` argumentum behelyettesítettsége (nyílt vagy zárt végű lista-e) nem számít.

Mintakeresés append / 3-mal

● Párban előforduló elemek

```
% párban(Lista, Elem): A Lista számlistának Elem olyan
% eleme, amelyet egy ugyanilyen elem követ.
párban(L, E) :-
    append(_, [E,E|_], L).

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

● Dadogó részek

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

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

Keresés listában

- `member(E, L)`: E az L lista eleme

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

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

- A `member/2` felhasználási lehetőségei

- Eldöntendő (igen-nem) kérdés:

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

- Lista elemeinek felsorolása:

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

- Listák közös elemeinek felsorolása – mindkét fenti hívásmintát használja:

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

- Egy értéket egy (nyílt végű) lista elemévé tesz, végtelen választás!

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

- A `member/2` keresési tere **véges**, ha második argumentuma zárt végű lista.

member/2 általánosítása: select/3

- `select(Elem, Lista, Marad)`: Elemet a Listából elhagyva marad Marad.

```
select(Elem, [Elem|Marad], Marad).      % Elhagyjuk a fejet, marad a fark.
select(Elem, [X|Farok], [X|Marad0]) :- % Marad a fej,
    select(Elem, Farok, Marad0).      % a farkból hagyunk el elemet.
```

- Felhasználási lehetőségek:

```
| ?- select(1, [2,1,3], L).             % Adott elem elhagyása
    L = [2,3] ? ; no
| ?- select(X, [1,2,3], L).             % Akármelyik elem elhagyása
    L=[2,3], X=1 ? ; L=[1,3], X=2 ? ; L=[1,2], X=3 ? ; no
| ?- select(3, L, [1,2]).               % Adott elem beszúrása!
    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]).
                                           % Beszúrható-e 3 az [1,...]-ba
    no                                     % úgy, hogy [2,...]-t kapjunk?
| ?- select(1, [X,2,X,3], L).
    L = [2,1,3], X = 1 ? ; L = [1,2,3], X = 1 ? ; no
```

- A `lists` könyvtár tartalmazza a `member/2` és `select/3` eljárások definícióját is.
- A `select/3` keresési tere **véges**, ha 2. és 3. argumentuma közül legalább az egyik zárt végű.

Listák permutációja

- `permutation(Lista, Perm)`: Lista permutációja a Perm lista.
(Az alábbi definíció a `library(lists)` könyvtárból származik:)

```
permutation([], []).
permutation(Lista, [Elso|Perm]) :-
    select(Elso, Lista, Maradek),
    permutation(Maradek, Perm).
```

- Felhasználási példák:

```
| ?- permutation([1,2], L).
      L = [1,2] ? ; L = [2,1] ? ; no

| ?- permutation([a,b,c], L).
      L = [a,b,c] ? ; L = [a,c,b] ? ; L = [b,a,c] ? ;
      L = [b,c,a] ? ; L = [c,a,b] ? ; L = [c,b,a] ? ;
      no

| ?- permutation(L, [1,2]).
      L = [1,2] ? ;
      végtelen keresési tér
```

- Ha `permutation/2`-ben az első argumentum ismeretlen, akkor a `select` hívás keresési tere végtelen!