LP-GYIK

Gyakori kérdések – válaszokkal

*

a logikai programozás témaköréből

Szeredi Péter
Mészáros Tamás

2006. április 10.

Tartalomjegyzék

Bevezető

Az LP-GYIK[1] a *Deklaratív programozás* tárgy hallgatóinak kérdéseiből és nemritkán válaszaiból készült összeállítás, amelyet a tárgy `dp-l@iit.bme.hu` levelezési listájára 2000. elejétől 2005. nyaráig érkező levelekből válogattunk ki. Ha tehát valaki a saját stílusára ismerne, az nem a véletlen műve. Sok-sok hallgatónak tartozunk köszönettel a jó kérdésekért és a még jobb válaszokért, közülük többen is mindkét térfelen küzdöttek az idők során (ugye, rablóból lesz . . .).

A teljesség igénye nélkül álljon itt jónéhány név a közreműködők közül, köztük olyanoké is, akik a tárgynak „hivatalosan” talán soha nem voltak hallgatói:

> Benkő Tamás, Lukácsy Gergely, Varga Szilvia, Sója Katalin, Wagner Ambrus, Tarján Péter, Pallinger Péter, Altrichter Márta, Pálfalvi Tamás, Hanák Dávid, Berki Lukács Tamás, Békés András György, Ottucsák György, Drótos Márton, Szeredi Tamás, Patai Gergely, Kiss Zoltán, Kovács Zoltán, Csík László, Dóra László, Szabó Péter, Mészáros Tamás.

Az eredeti „hangszerelést”, ahol nem ment az érthetőség rovására, sok helyen megőriztük. Sokszor csaknem ugyanaz a kérdés és válasz többféle változatban is olvasható; ha többször kérdezték ugyanazt, valószínűleg nehezebben érthető dologról van szó, érdemes hát többször foglalkozni vele.

Az LP-GYIK első néhány verziójában biztosan lesznek hibák, elírások: az észrevételeket köszönettel várjuk a „hanak at inf dot bme dot hu” vagy a „szeredi at cs dot bme dot hu” címre.

Budapest, 2006. április — Szeredi Péter, Hanák Péter

[1] GYIK = Gyakori kérdések – válaszokkal

1. fejezet

Cékla

1.1. Kérdés. Win-es céklában hogy kell lefordítani a programot?

> **1.1. Válasz.** ***Hallgató:*** A cekla.exe futtatása után be kell gépelni: `load <fájlnév.c>` ezután lehet tesztelni a függvényeket.

1.2. Kérdés. Találtam egy hibát a Cekla 1.1-ben. Windows XP alatt a program nem hajlandó futni, ha bizonyos könyvtárakból indítom. Például a `d:\` könyvtárból elindul, de ha egy olyanba másolom, amiben pont, ékezetek, és szóköz is van, akkor egyszerűen visszakapom a parancssort.

> **1.2. Válasz.** ***Oktató:*** Sajnálom, ezen nem tudok segíteni. Használjanak szóköz- és ékezetmentes könyvtárneveket.

1.3. Kérdés. Az a kérdésem, hogy a céklában megengedett-e az olyan feltételvizsgálat, ahol a feltétel egy függvény értéke? Tehát ilyenre gondolok, mint:

```
if(fuggveny(x))
```

vagy

```
if(fuggveny(x)==1)
```

> **1.3. Válasz.** ***Oktató:*** Csak a második alak a megengedett.

1.4. Kérdés. Érdeklődnék, hogy a honlapról letölthető céklaverziónál egy filet hogyan lehet betöltetni a fordítónak, mert nekem sehogyan sem sikerült rájönnöm, és a helpjében sem találtam erről információt.

> **1.4. Válasz.** ***Oktató:*** Elindítja a céklat, és begépeli neki, hogy

```
load <állománynev>;
```

2. fejezet

Fejlesztés, rendszer

2.1. Kérdés. Van maximuma az átadható lista méretének?

2.1. Válasz. ***Oktató:*** A memória mérete egy korlát, és vannak még implementációs részletek is (256MB alatt nem számít).

Az átadas csak egy mutató átadását jelenti, tehát ha a lista egyszer már létrejött, az átadás majdnem ingyen van.

A kérdező folytatja: Egy kb 4x20-as listát valamiért nem adott át a programom...

Oktató: Egy 4 elemű listát, aminek minden eleme 20 elemű lista? Ennek semmiképp nem szabad gondot okoznia.

A kérdező folytatja: A trace-t ebben az esetben már nem volt értelme alkalmazni mert több 100 (1000?) művelet míg a kritikus ponthoz jutnak :(

Oktató: Ki lehetne próbálni a „`| ?- spy <eljárásnev>.`” hívást. Ez bekapcsolja a debuggert, és amint <eljárásnev> meghívatik, megállítja a programot és lehet trace-elni. (A „nospy” eljárással lehet levenni a töréspontot.)

2.2. Kérdés. Mit jelent a következő Warning üzenet a SICStus Prologban?

```
| ?-  consult('kishazi.pl').
{consulting kishazi.pl...}
{Warning: [Alap,Jegyek] - singleton variables in user:szam/3
 in lines 17-18}
{consulted kishazi.pl in module user, 60 msec 1288 bytes}
```

2.2. Válasz. ***Hallgató:*** Az a gond, hogy ezek a változók csak egyszer fordulnak elő a hivatkozott (17-18. sorbeli) klózban. Például NEM használtad fel ezeket az eljárás törzsében, pedig a fejben ott vannak (bizonyára lényegtelen változók.) A hiba kivédése: használj helyettük `_Alap`, `_Jegyek`, vagy `_`, változónevet, ebből a Prolog tudni fogja, hogy ezek a változók csak a mintaillesztéshez kellettek.

2.3. Kérdés. Hogyan lehet hatékonyan debuggolni Prologban? A trace-t ismerem, de hogyan lehet megmondani neki, hogy csak egy bizonyos pontnál kapcsoljon be. Tehát vmi search egy eseményre, vagy sorszámra, vagy klóz-hívásra.

2.3. Válasz. ***Hallgató:*** Lásd: `spy/1` (operator). Pl.

```
| ?- spy cikcakk.
| ?- spy member/3.
```

Kikapcsolása `nospy/1`, ill `nospyall/0`.

2.4. Kérdés. Hogyan lehet egy abortálásnal megtudni, hogy hol tartott a program, és mik voltak a változókban?

2.4. Válasz. ***Oktató:*** Ezt nem lehet.

A kérdező folytatja: Hogyan lehet nagyobb mélységig kiíratni egy listát?

Oktató: A debuggerben a < parancs átállítja a max mélységet:

```
+   1  1 Call: member(_138,[1,2,3,4,5,6,7,8,9|...]) ? <20
+   1  1 Call: member(_138,[1,2,3,4,5,6,7,8,9,10,11,12]) ?
```

A `debugger_print_options` Prolog jelző (Prolog flag) átállításával is elérhető ez, illetve a `toplevel_print_options` jelzővel a behelyettesítések kiírasi mélysége is átállítható. A jelzők a `current_prolog_flag` és a `set_prolog_flag` beépített eljárásokkal kérdezhetők le ill. állíthatók be, lásd a jegyzet 5.13 pontját. Például:

```
| ?- length(L, 20).

L = [_A,_B,_C,_D,_E,_F,_G,_H,_I,_J|...] ?

yes
| ?- prolog_flag(toplevel_print_options, Opts).

Opts = [quoted(true),numbervars(true),portrayed(true),
max_depth(10)] ?

yes
| ?- set_prolog_flag(toplevel_print_options, [quoted(true),
numbervars(true),portrayed(true),max_depth(100)]).

yes
| ?- length(L, 20).

L = [_A,_B,_C,_D,_E,_F,_G,_H,_I,_J,_K,_L,_M,_N,_O,_P,_Q,
_R,_S,_T] ?

yes
| ?- set_prolog_flag(debugger_print_options, [quoted(true),
numbervars(true), portrayed(true),max_depth(100)]).


yes
```

```
| ?- use_module(library(lists)).
{loading /usr/local/lib/sicstus-3.8.1/library/lists.po...}
{module lists imported into user}
{loaded /usr/local/lib/sicstus-3.8.1/library/lists.po in
module lists, 20 msec 13256 bytes}

yes
| ?- trace, member(X, [1,2,3,4,5,6,7,8,9,10,11,12]).
{The debugger will first creep -- showing everything
(trace)}
   1  1 Call: member(_117,[1,2,3,4,5,6,7,8,9,10,11,12]) ?
?  1  1 Exit: member(1,[1,2,3,4,5,6,7,8,9,10,11,12]) ?

X = 1 ?

yes
{trace}
| ?-
```

2.5. Kérdés. A weblapon lévő feladatokat nézegetve találtam egyet amelyik nem futott le a SICStusban:

```
p(N) :- write(N),
nl,
number_chars(N, NL),
reverse(NL, [_,DC|_]),
number_chars(D, [DC]),
length(L, D),
append(L, [E|_], NL),
put_char(E),
nl.

{TYPE ERROR: put_char(52) - arg 1: expected character,
found 52}
```

Megnéztem a könyvet, és az volt írva a `number_chars`-hoz: Ha Szám tizes számrendszerbeli alakja c1c2c3c4c5... akkor a Karakterlista= [c1,c2,c3,c4,c5] lesz.

2.5. Válasz. ***Oktató:*** Ez már egy nagyon régi feladat lehet . . . A könyvbeli definíció az ISO módú futásra vonatkozik, annak bekapcsolása után lép életbe, tehát azután, hogy lefuttatta a következő célt:

```
| ?- set_prolog_flag(language, iso).
```

Ezután futnia kell a fenti kódnak.

Egyébként pont emiatt a probléma miatt a `_char(s)` végű eljárások használatát már több éve nem is tanítjuk, helyettük a `_code(s)` végű megfelelőket használjuk (a vizsgafeladatokban is a `_code` végű eljárások szerepelnek). Ha `chars->codes` cserét végrehajtja, a fenti program a nyelv-mód beállítástól függetlenül működőképessé válik.

2.6. Kérdés. A minap találtam egy (talán) bugot: Interaktív módban az `atom(a')` utasítást írva, ahol `a` egy atom,`'` pedig aposztrof, a program nem áll le, de pl. a parancsismétlő működik...

2.6. Válasz. ***Oktató:*** Én nem nevezném ezt hibának. A SICStus Prolog alapértelmezésben megenged az idézett atom belsejében is újsort, tehát az `atom(a').` sor után várja az idézett atom folytatását. ISO-Prolog módban ez nem megengedett, tehát mindenképpen hibás inputnak számít ez, de ekkor is a következő sorvégi pontig elolvas.

2.7. Kérdés. A házi feladatot megcsináltam SWI Prologgal, ment is jól. Viszont mikor megpróbáltam consult()-olni a GNU Prologgal, akkor az alábbi hibát kaptam:

```
| ?- consult(x).
compiling /mnt/store/x.pl for byte code...
/mnt/store/x.pl:62 error: syntex error: ,or ) expected
(char:17)
/mnt/store/x.pl:68 error: syntex error: ,or ) expected
(char:17)
        2 error(s)
compilation failed

no
```

A 61. sortól a file tartalma (a 61. sor a megjegyzás):

```
% ...kiszedtem, nem mondom meg, hogy mit csinal...
szépít(A / B, E # F) :-
        LNKO is gcd(A,B),
        E is A // LNKO,
        F is B // LNKO,
        B > 0.
```

A 68. sorban ugyanilyen hiba van, a klóz majdnem ugyanez. (ki lehet találni, hogy mi is lehet a klóz... :-))

A kérdésem az, hogy a gprologot hogy lehet rávenni, hogy megértse ezt a klózt? Szerintem a gcd-vel van baja, talán consultolni kellene valami matematikai könyvtárat, de mit?

2.7. Válasz. ***Oktató:*** A hibajelzésben a `(char:17)` szöveg arra utal, hogy az adott sor 17. karakterénél észlelte a rendszer a hibát. Jól látszik, hogy a # jellel van a baj. Előadáson említettem is, hogy a # **nem** szabványos operátor. A javításhoz elég annyit csinálnia, hogy programja elején elhelyez egy operátordeklarációt:

```
:- op(<prioritás>, <fajta>, #).
```

2.8. Kérdés. Olyan problémám lenne hogy nem tudom felrakni, jobban mondva futtatni a SICStust! Letöltöttem, felraktam,(valószínű elírtam valamit a telepíteskor és nem indult. (invalid licens)

Nosza nem keseredtem el, gondoltam felrakom újra és ügyelek a bitenkénti beírásra. Nem jött be. Most tuti jól töltöttem ki mindent és: „`License error: License could not be verified`”

2.8. Válasz. ***Oktató:*** Ha jól sejtem windowsról van szó. Ha így van, akkor érdemes lehet az újratelépítés előtt a régit leszedni.

Ellenőrizni kéne, hogy maga a licensz jó-e. Ha licenszproblemára gyanakszunk, akkor nem kell újratelépíteni a SICStust, elég az splm programot használni.

2.9. Kérdés. Hogyan lehet egy Prolog program lépésszámát megmérni? Van valami kapcsoló vagy beépített dolog? Valami belső programba írt lépésszámlálóval hogyan lehetne megcsinálni?

2.9. Válasz. ***Oktató:*** A profi profilozásra van egy könyvtár (lásd a „The Gauge Profiling Tool" c. fejezetet), ill a debugger haladó eszközeivel is lehet ilyet csinálni (lásd az „Advanced Debugging – an Introduction" c alfejezetet).

Alább küldök egy egyszerű módszert, amely azon alapul, hogy 'debug' módban a debugger minden hívásnál 1-gyel növeli a hívásszámlálót, de nem áll meg. Ha ezután bekapcsoljuk a trace módot, akkor látszik a kurrens hívásszámláló.

```
| ?- [user].
% consulting user...
| p(N) :-
     N > 0, N1 is N-1, p(N1).
| p(0).
|
% consulted user in module user, 10 msec 272 bytes
yes
| ?- debug, p(1000), trace, foo.
% The debugger will first leap -- showing spypoints (debug)
% The debugger will first creep -- showing everything
        (trace)
     3003      1 Call: foo ? a
% Execution aborted
| ?-
```

2.10. Kérdés. Szeretném megkérdezni, hogy a SICStus Prolog, vagy általában a Prolog rendszer ad-e valamilyen lehetőséget arra, hogy egy-egy lekérdezés futási idejét meg lehessen tudni? Persze ez nyilván gépfüggő, de valószínűleg sokat segíthetne abban, hogy mivel mennyire lehet egy program teljesítményét növelni (pl: vágók, alternatív megoldások, stb.).

2.10. Válasz. ***Oktató:*** Igen, a futási idő lekérdezés nincs szabványosítva, ezért Prologonként változik. SICStus-ban a `statistics(runtime,...)` ill. a `statistics(walltime,...)` hívások használhatók erre a célra, lásd a doksit.

2.11. Kérdés. Érdekelne hogy hogy lehet Vbasic-ben lefordított programoknak futni egy SICStus nélküli gépen?

Próbáltam átmásolni a vbsp.dll vbsp.po, *.pl, msvbmv50+60.dll, linkelni, újra lefordítani, de csak akkor hajlandóak futni ha SICStus bin-ja a path-ban van.

2.11. Válasz. ***Oktató:*** Olvassa el a Release Notes vonatkozó fejezeteit (a `relnotes.html` állomány az Ön által letöltött disztribúcióban is benne van):

`http://www.sics.se/sicstus/docs/latest/html/relnotes.html`

illetve ezen belül:

```
#Visual%20Basic%20notes
#Runtime%20Systems%20on%20Target%20Machines
#Redistributable%20files%20Windows
```

Ha jól értem, az `sprt.sav` állományt kell a célgépre (SICStus nélküli gépre) áttenni (és ott a path-ba betenni). Ez a „Redistributable files Windows” fejezet alapján úgy tűnik, hogy engedélyezett.

2.12. Kérdés. Azt szeretném kérdezni, hogy hogyan mérhetném meg, hogy egy program lefutása alatt mennyi időt töltünk az egyes függvényekben.

2.12. Válasz. ***Oktató:*** Lásd a kézikönyv ’Execution Profiling’ c. fejezetet, ill. az ebben leírt eljárásokhoz grafikus felületet nyújtó library(gauge) könyvtárat.

2.13. Kérdés. Az alábbi párbeszéd történt a SICStus-szal:

```
| ?- X is 4.2-4.
X = 0.20000000000000018 ? ;
no
```

ugyanakkor:

```
| ?- X is 4.2-2.0.
X = 2.2 ? ;
no
```

de:

```
| ?- X is 4.2-3.0.
X = 1.2000000000000002 ? ;
no
```

Mint a legtöbb programozási nyelv, így a Prolog is az adott gép lebegőpontos számábrázolását használja. A 4.2 valós szám pontosan csak egy végtelen kettedestörttel ábrázolható, aminek véges közelítését tárolja a gép. A SICStus Prolog rendszer szemére legfeljebb az vethető, hogy „palástolatlanul”, tehát teljes pontossággal írja ki a lebegőpontos számokat.

2.14. Kérdés. A kérdésem az lenne, hogy miért nem akar működni az alábbi Prolog progi részlet:

```
volt(X-Y,[VOLTX-VOLTY|LISTA],ERI):-
  (VOLTX=:=X, VOLTY=:=Y-1), ERI is 1.
```

Ugyanis azt írja ki, hogy `[LISTA] is a singleton variable`, vagy valami hasonló.

2.14. Válasz. ***Oktató:*** Ettől még működik a programja, mert ez csak egy figyelmeztetés. Arra figyelmezteti Önt, hogy a `LISTA` változót mindössze egyszer használta az adott klózban, és ez sokszor egy elírás miatt van így.

Ha egy változót _ (aláhúzás) jellel kezd, akkor ezzel konvenció-szerűen jelzi, hogy tudatában van annak, hogy ez egy egyszeri változó-előfordulás. Tehát ha fent `LISTA` helyébe `_LISTA`-t vagy `_`-t ír, akkor nem kap figyelmeztetést.

2.15. Kérdés. Hogyan kell egy bizonyos dolgot file-ba íratni Prologgal? Teszem azt van egy számolás, ami egymás után adja ki az eredményeket, és nem akarom, hogy mindig megjelenjen, hanem szépen írja bele egy file-ba.

2.15. Válasz. ***Hallgató:*** Ez pl. egy nagyházi keretprogramból megnézhető, bevágom ide:

```
open_file(FileOut, write, Stream),
current_output(OOut),
set_output(Stream),
write(Irnivalo),
```

....írkálás.....,

```
set_output(OOut),
close(Stream)
```

Érdemes `findall/3`-at is használni, ha az összes megoldást keresed.

2.16. Kérdés. Az előadás 88, 89. oldalain van szó a jószám című programról. Lefuttattam a programot a 27-es paraméterrel: `joszam(27).` az emacs alatt is, és az rdbg alatt is. Az emacs a yes-ig ment, és utánna kilépett igennel, az rdbg meg a yes után még keresett megoldásokat. Én úgy gondolom, hogy az emacs trace-nek van igaza, mert ha egyszer igaz az állítás, akkor már nem kell több megoldást keresni. Több megoldást akkor kell ha mondjuk `joszam(Sz).` parancsot adtam volna. Azt szeretném megkérdezni, hogy az emacs nyomkövetőjének, vagy az rdbg nyomkövetőjének van igaza? Előre is köszönöm.

2.16. Válasz. ***Oktató:*** Nehéz igazságot tenni.

A SICStus Prolog egy igen/nem kérdésre nem keres több választ, hanem a „yes” kiírása után az igenis létező Prolog keresési tér maradék részét „levágja” és azt nem járja be. Ha „megtéveszti” a Prologot, és a

```
| ?- joszam(27), X = akarmi.
```

kérdést teszi fel, akkor meg is nézheti a fennmaradó keresési teret.

Az rdbg mindenképpen bejárja a teljes keresési teret. Mivel ennek a programnak a feladata a keresési tér szemléltetése, itt természetesebbnek érzem, hogy megmutatja és bejárja a teljes keresési teret.

Ha ez nem világos, legyen szíves és kérdezzen meg.

3. fejezet

Alapok

3.1. Kérdés. Hogyan lehet editálni egy már bevitt Prolog szabályt?

3.1. Válasz. ***Oktató:*** A Prolog consult/compile beépített eljárások „felülcsapják” azt a predikátumot, amit korábban már létrehoztunk. Például:

```
SICStus 3.8.1 (x86-linux-glibc2.1): Tue Dec 21 16:50:07 CET
1999
Licensed to TUB DP course
| ?- [user].
| p(1, X) :- q(X).
| p(2, X) :- r(X).
| {consulted user in module user, 0 msec 312 bytes}

yes
| ?- listing(p).
p(1, A) :-
        q(A).
p(2, A) :-
        r(A).
yes
| ?- [user].
| p(1, X) :- q(X), s.
| {consulted user in module user, 10 msec -168 bytes}

yes
| ?- listing(p).
p(1, A) :-
        q(A),
        s.

| ?-
```

Tehát a második `[user].` parancs, mihelyst meglátta, hogy a `p/2` predikátum egy klóza következik, kitörölte a teljes predikátumot (mindkét klózt), és csak ezután vette fel az új változatot (vegyék észre a negatív memória-foglalást is).

Szeretném még hangsúlyozni, hogy a klózok terminálról való bevitele nem ajánlott, mert nem lehet kényelmesen elmenteni az így bevitt programot. Ehelyett azt javaslom, hogy a SICStus ablak mellett tartsanak nyitva egy szerkesztő ablakot, abban javítsanak, és minden javítás után a SICStus-ba töltsék be újra az állományt, pl. a `[allomanynev].` paranccsal. Ha ez a szerkesztő az Emacs, akkor ezen belül is lehet futtatni a SICStus-t, és külön Emacs parancsok is vannak az újra-betöltésre.

3.2. Kérdés. Azt szeretném megtudni, hogy nincs-e valamilyen olyan beépített eljárás, ami megmondja egy változóról, hogy annak van-e már értéke, vagy nincs. Pl. valami ilyen:
`vanértéke(A)`

3.2. Válasz. ***Hallgató:*** Többféle megoldást is el tudok képzelni:

```
| ?- nonvar(A).

no

| ?- nonvar(f(A)).

true ?

yes

| ?- ground(f(A)).

no

| ?- ground(f(a)).

yes
```

Ezek mind megtalálhatók a jegyzetben.

3.3. Kérdés. Sajnos nem akar működni a windowsos program, ezt csinálja:

```
?- szuloje('Imre', 'Istvan').
(EXISTENCE ERROR: szuloje('Imre', 'Istvan'): procedure
user:szuloje/2 does not exists)
?-
```

3.3. Válasz. ***Hallgató:***

1. Megírod a programodat egy például szulo.pl állományba. Egyszerű szövegszerkesztővel.
2. Betöltöd a SICStus-t.
3. Beírod, hogy consult(szulo).

Erre visszakapsz egy sort, hogy consulting... és egy yes-t.

Ezután kérdezhetsz tőle. :-)

3.4. Kérdés. Hogyan tudok egy listához olyanformán elemet hozzáadni, hogy maga a lista megváltozzon?

3.4. Válasz. ***Oktató:*** Sehogy. Gondolkozzon el, mit is szeretne, mit ért azon, hogy „maga a lista megváltozzon”! Valószinűleg azt, hogy szeretne egy memóriahelyet, amelyben eddig egy lista volt eltárolva, ezután meg ugyanott egy másik listát szeretne eltárolni. Nos, a deklaratív programozás lényege, hogy nem használhatunk ilyen jellegű változót. A deklaratív nyelvekben a változó egyetlen (esetleg még ismeretlen) mennyiséget jelöl, úgy mint a matematikában. Úgyhogy joggal várja el a Prologtol, hogy az X = X+1 -re meghiúsuljon, hiszen a matematikában az X=X+1 állítás azonosan hamis. (Annak, hogy az X=X+1 valójában nem hiúsul meg, hatékonysági okai vannak, de tulajdonképpen Ön valószínűleg inkább az X is X+1 hívást szerette volna használni, amely tényleg meghiúsul.)

A kérdező folytatja: Ezzel a szabállyal próbálkoztam:

```
hozzaad(X,Lista,[X|Lista]).
```

Oktató: Ez nagyon jó. Egy `hozzaad(X, L, L1)` hívással elő tudja állítani `L1`-ben, tehát egy **új változóban** azt a listát, amelynek feje `X` és farka `L`. Ezt még lehet egyszerűsíteni a beépített `=/2` eljárás használatával, tehát a hozzaad hívás helyett írhatja ezt is:

```
L1 = [X|L]
```

Sőt még ezt az egyenlőséget is többnyire ki lehet küszöbölni, úgy hogy az `L1` minden előfordulása helyére az `[X|L]` kifejezést írjuk.

A kérdező folytatja: Lehet-e - és ha igen, hogyan - működőképessé varázsolni ezt a megoldást? Milyen más lehetőség létezik egy lista bővítésére?

Mint mondtam, „változtatható” változót lehetetlen csinálni. De minden programozási feladat megoldható enélkül is, deklaratív módszerekkel. Az imperatív változó szerepet általában egy argumentumpár veszi át, a ciklusból rekurzió lesz. Nézze meg pl. a 6. előadáson szerepelt `revapp` eljárást:

```
% revapp(L, R0, R): L megforditását R0 elé fűzve kapjuk R-t.
revapp([], R, R).
revapp([X|L], R0, R) :-
    revapp(L, [X|R0], R).
```

Itt a második argumentumban gyűjtjük az egyre bővebb listákat, és a rekurzió végén adjuk vissza az eredményt a harmadik argumentumban. Ugyanez C++-ban:

```
list revapp(list list1, list list2)
{ list list3 = list2;
  for (list pos = list1; pos; pos = pos->next) {
    list new_list = new link(pos->elem);
    new_list->next = list3; list3 = new_list;
  }
  return list3;
}
```

Tehát a `list3` imperatív változónak az `<R, R0>` argumentum-pár felel meg a `revapp(L, R0, R)` hívásban, `R0` mutatja a `list3` kezdőértékét a `revapp` meghívásakor, míg `R` a végérték, `revapp` lefutása után.

3.5. Monológ. ***Oktató:*** Itt szeretném felhívni a figyelmet néhány formai szabályra, amelynek betartásával könnyebben olvasható és karbantartható programokat írhatnak:

Az egy predikátumhoz tartozó klózokat (azonos funktorú klózok) folyamatosan, üres sor közbeiktatása nélkül írjuk, a predikátumok közé egy üres sort teszünk.

A klóz fejét a sor elején kezdjük, a törzset nehány szóközzel beljebb. A fej és a `:-` nyakjel közé rakunk egy szóközt, ugyanígy szóközt rakunk a törzsbeli hívásokat elválasztó vesszők és a hívások argumentumait elválasztó vesszők után. Viszont nem rakunk szóközt a listaelemeket, ill. a struktúrák argumentumait elválasztó vesszők után.

A diszjukniót mindenképpen zárójelbe tesszük, úgy, hogy a nyitó zárójel, az alternatívákat kezdő pontosvesszők és a végzárójel pontosan egymás alá kerüljön. Az alternatívák célsorozatait néhány szóközzel beljebb kezdjük.

Példák:

```
% 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).
        % a struktúra-argumentumlistában nincs,
        % a hívás-argumentumlistában van szóköz!

% párban(Lista, Elem): A Lista számlistának Elem olyan
% eleme, amely egy ugyanilyen értékű elemmel szomszédos.
párban(L, E) :-
        append(_, [E,E|_], L).
        % a listaelemek után sincs szóköz!

% member(E, L):  E az L lista eleme.
member(Elem, [Fej|Farok]) :-
        (   Elem = Fej
        ;   member(Elem, Farok)
        )
```

Mindenképpen **tilos** szóközt rakni az eljárás- ill. struktúranév után!

Az egyszer előforduló változókat egyetlen aláhúzásjellel jelöljük, vagy aláhuzásjellel kezdjük. Ugyanannak a „mennyiségnek” a különböző állapotait rendre a 0, 1, ... változó-végződéssel különböztetjük meg, a végállapot végződés nélküli.

Példa:

```
hozzaado(A+B, Ford0, Ford) :-
        hozzaado(A, Ford0, Ford1),
        hozzaado(B, Ford1, Ford).
```

3.6. Kérdés. Érdekelne, hogyan lehet a programlistából egy klózt vagy predikátumot törölni...

3.6. Válasz. ***Oktató:*** Ha jól értem, az iránt érdeklődik, hogy egy már a SICStus-ba betöltött Prolog program hogyan módositható. A válasz az, hogy ez legfeljebb nagyon körülményesen tehető meg.

A mai Prolog-ok többsége, így a SICStus is, nem ad integrált programfejlesztési környezetet, tehát a program-módosítást a kiinduló állomány javitásával és újra-betöltésével kell elvégezni.

Állományok betöltésére a `consult('...')` vagy `compile('...')` beépített eljárások használhatók. Az előbbi lassabb futást eredményez, de jobban nyomkövethető. A Prologból kilépni a `halt` beépített eljárással, vagy a file-vég karakterrel lehet (unixon: `^D`, Windowson: `^Z`).]

3.7. Kérdés. `X is 11 mod 3, X is X+1.` A fenti sort nem értem, mert ugye

`| ?- X is 11 mod 3.` ⟶ `X=2`

`| ?- X is X+1.` ⟶ futási hiba

Viszont a kettő együtt meg ⟶ `no`

Miért nem 'rontja el' az `X is X+1.` az egész kifejezést??? Nekem ez lenne logikus..

3.7. Válasz. ***Oktató:*** A kérdés azt a **tévhitet** tükrözi, hogy az `| ?- X is X+1.` célsorozat azért hibás, mert „nem szabad egy változónak többször értéket adni". Erről szó sincs, hiszen a (szabványos) Prolog nem is tudja mi az a (többszörös) értékadas: a változó eleve csak egyetlen értéket jelenthet. Az értékadas helyett Prologban egyesítésről beszélünk: ez egyrészt behelyettesíthet változókat (pl. az `X = 1` hívás eredménye az `X`←`1` behelyettesítés), de meghiúsulást is okozhat (pl. az `X = 2, X = 1` hívássorozat eredménye meghiúsulás, hiszen a második hívásban az `X` már értékkel bír, és az az érték nem `1`). Tehát a „többszörös értékadás kisérlete" cselekmény egyáltalán nem büntetendő hibajelzéssel, csak közönséges meghiúsulást okoz.

Járjuk egy kicsit bővebben körbe ezt a témat. Kezdjük annak a logikájával, hogy mikor jelez a rendszer hibát és mikor hiúsul meg.

Tulajdonképpen a meghiúsulás tekinthető úgy, mint a kivételkezelés (hibakezelés) legprimitívebb formája. Az első Prolog rendszerekben, ha valaki egy nem definiált eljárást hívott meg, akkor ez meghiúsulást okozott és nem hibajelzést. Ez teljesen logikus, megfelel a (rezolúciós tetélbizonyitási) logikai alapoknak. A programfejlesztés hatékonysága szempontjából viszont sokkal jobb ha hibajelzést ad a rendszer, hiszen különben a meghiúsulás egy másik ágra terelheti a végrehajtást, és így nehéz lehet ennek az okát kinyomozni. (Megjegyzem, hogy a napokban közreadott nyomkövető interpreter írása során nekem is külön kellett küzdenem azért, hogy a nem definiált eljáráshívások hibát jelezzenek.)

A beépített eljárások akkor jeleznek hibát, ha olyan adatot kapnak, amivel nem képesek elvégezni a kivánt műveletet. Például az `is/2` a második argumentumában egy tömör (behelyettesítetlen változót nem tartalmazó) aritmetikai kifejezést vár. Tehát az `X is X+1` azért jelez hibát, mert a jobboldalon behelyettesítetlen változó van (a baloldalt ilyenkor még meg sem nézi!).

Sokszor büszkélkedtem azzal, hogy a relációs programozásban egyetlen reláció megírásával több függvényt is kapunk. A klasszikus példa az `append/3` eljárás, amely listák összefűzésére készült, de szétszedésre is használható. Voltak olyan Prolog rendszerek, ahol az aritmetikai műveleteket is relációs formában kellett irni, pl. a `plus(X, Y, Z)` eljárás azt fejezte ki, hogy `Z = X+Y`. Ezek között a Prolog rendszerek között olyanok is voltak, ahol pl. a `plus/3` eljárást kivonásra is lehetett használni: a `plus(X, Y, Z)` hívás, ha mondjuk X és Z adott volt, akkor Y-t kiszámolta.

Az `is/2` eljárástól nem célszerű ilyen intelligenciát elvárni. De még itt is lehet többféle használatról beszélni. Az `is/2` alapvető használata az, hogy a második argumentumában lévő aritmetikai kifejezést kiértékeli, és az értéket az első argumentumbeli **változóba** helyettesíti. De valójában az `is/2` nem követeli meg, hogy az első argumentum változó legyen, hanem a kifejezés értéket **egyesíti** az első argumentumával. Tehát az `is/2` eljárás arra is használható, hogy ellenőrizzük, hogy a kifejezés értéke megegyezik egy előre megadott számmal.

Általánosan elmondhatjuk, hogy szinte nincs is olyan beépített eljárás, amely egy kimenő paraméteréről kiköti, hogy az változó legyen. A Prolog természetéből fakad, hogy a kimenő paramétereket egyesítéssel tölti fel, ennek következtében az ilyen paraméterpozíció bemenőként is használható, ellenőrzésre.

Tehát az

```
| ?- X = 2, 2 is X+1.
```

célsorozat futásában nincs hiba: X-nek a 2 értéket adjuk, majd megvizsgáljuk, hogy igaz-e az, hogy X+1 kiértékelése a 2 eredményt adja. Itt az első hívás sikerül, a második meghiúsul. De hát akkor az

```
| ?- X = 2, X is X+1.
```

célsorozat sem hibás, hiszen mire a második híváshoz érünk, mindkét esetben ugyanazt a `2 is 2+1` hívást látjuk.

3.8. Kérdés. Mi a különbség az „`is`” és az „`=`” beépített predikátumok között?

3.8. Válasz. ***Oktató:***

- Az `= /2` predikátum egyesíti a két oldalán található **tetszőleges** Prolog kifejezéseket.
- Az `is/2` a jobb oldalán található **aritmetikai** kifejezés értékét egyesíti a bal oldali **tetszőleges** Prolog kifejezéssel. (Haladóknak: nem minden Prolog implementáció szereti, ha az `is` bal oldala nem változó.)

3.9. Kérdés. A mai előadáson keletkezett egy kis gubanc a fejemben. Az elején szó volt róla, hogy azért jó a deklarativ programozás, mert nincsenek benne explicit értékadások. Ilyesmi, hogy `n = n + 1;` Szóval így nincs adat egymásrahatás, könnyen párhuzamosítható. De akkor mi ez, hogy `N1 is N - 1`? Ez nem ellenkezik ezzel??

3.9. Válasz. ***Oktató:*** Itt nincs semmiféle ellentmondás. Nem az „explicit értékadás” a gond, hanem a nem egyszeres értékadás.

`N1 is N-1` esetén `N1` egy, `N`-től teljesen független változó, tehát nem az történik, hogy `N` értékét módosítjuk. Az `N is N-1` hívás meghiúsulna, hiszen az `is/2` eljárás kiértékeli a jobb oldalt (tegyük fel, hogy `N=3`, ekkor a jobboldal kiértékeltje 2) és megpróbálja ezt egyesíteni a bal oldallal. Persze 3 nem egyesíthető 2-vel.

Azért jó, hogy az új érték új nevet kap, mert így az egyes eljáráshívások függése explicitté válik, és a párhuzamosítás lehetősége kiolvasható ezekből a függőségekből. Pl. az alábbi célsorozatban

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

Ha itt az `X` paraméter bemenő (tömör kifejezés), akkor a „vak is látja”, hogy a `q` és `r` hívások egymástól függetlenek, párhuzamosan végrehajthatóak. Ha ugyanezt a kódot imperatív nyelven írjuk le, akkor erre a következtetésre csak úgy juthatunk, ha meg tudunk győződni arról, hogy a `p` és `q` eljáráshívások nem változtatnak meg közös globális változót, ami elég nehéz lehet.

3.10. Kérdés. Miért hiúsul meg a következő: `x (X,Y) = x (1,2).` ? Szerintem átírható lenne így: `x(',')(X,Y))=x(',')(1,2)).`

3.10. Válasz. ***Hallgató:*** Nem hiúsul meg, hanem szintaktikai hibát jelez. Azért, mert az `x` struktúranév után nem lehet szóköz.

Ha a szóközt elhagynánk, akkor működne: `| ?- x(X,Y)=x(1,2).` ⟶ `X = 1, Y = 2`

3.11. Kérdés. Az lenne a kérdésem, hogy Prologban van-e olyan beépített lehetőség, amellyel elérhető, hogy egy tetszőleges hívás, ha annak egyébként lenne több megoldása, az első megoldás megtalálása után nem hoz létre választási pontot, és így már nem is keresi tovább az esetleges további megoldásokat.

3.11. Válasz. ***Oktató:*** Lásd a `once/1` ISO standard eljárást. Az argumentumaként kapott célt meghívja, de az első siker utáni választási pontokat levágja.

4. fejezet

Egyesítés

4.1. Kérdés. Mennyi lesz X értéke és miért?

```
X = 2-1
```

Egyáltalán mit jelent az „=” operátor?

4.1. Válasz. ***Hallgató:*** Egyesítés. Prologban a „`2-1`”-et nem értelmezi a nyelv maga aritmetikai kifejezésként, csak egy struktúraként. (alapstruktúra-alakja „`-(2,1)`”). Tehát itt X értéke „`2-1`” lesz.

4.2. Kérdés. Miért sikerül az, hogy `f(X,a)=f(Y,Y).` ? `X` nem azonos `Y`-nal. Bár kiértékelhető. De az nem az ’is’-lenne?

4.2. Válasz. ***Oktató:*** Az `=` egyesítést jelent. `X` és `Y` kezdetben behelyettesítetlen változók. Tehát az egyesítéshez az alábbiak szükségesek:

`X` egyesíthető `Y`-nal $\longrightarrow$ teljesül

Tehát visszavezettük erre:

`f(Y,a)=f(Y,Y).`

Itt `Y`-t kell egyesíteni `a`-val, ami lehetséges, mert `Y` behelyettesítetlen változó.

`f(a,a)=f(a,a)` fennáll, és `Y=X, Y=a`

Ezzel szemben az „is” aritmetikai műveleteknél használatos:

pl.

```
?- X is 1 + 2.
X = 3 ?
```

4.3. Kérdés. Emlékszem, hogy Szeredi tanár úr egyik előadásan bemutatta, hogy lehetőség van egy Prolog fejben két nevet is megadni, hogy később ne kelljen a programnak egyesíteni. Kicsit érthetőbben egy példán:

```
valami([A|B]) :- A=1, P=[A|B], valami2(P).
```

4.3. Válasz. ***Oktató:*** A fenti kódban a fejillesztés ellenőrzi, hogy legalább egyelemű listáról van szó, és egyben szét is szedi az argumentumot egy `A` fejre és egy `B` farokra. Ezután ellenőrzi, hogy a fej értéke 1, majd újra összerakja a paramétert, hogy `valami2`-t meghívhassa vele. Ezt az újbóli összerakást lehet megtakarítani, ha a fenti kódot a következőképpen átírjuk:

```
valami(P) :- P=[A|B], A=1, valami2(P).
```

Fontos lehet, hogy a törzs első hívása legyen a argumentumot szétszedő egyesítés, mert ettől függ, hogy az indexeléskor tudja-e majd a Prolog, hogy az adott klózt csak `'.'/2` funktorú argumentummal lehet meghívni.

A fenti kódot tovább egyszerűsíthetjük így:

```
valami(P) :- P=[1|_], valami2(P).
```

4.4. Kérdés. Ezt magyarázza meg nekem valaki! Miért nem azt mondja, hogy false???

```
| ?- X=X+1.

X = ... + ... +1+1+1+1+1+1+1+1+1 ? ;

no
```

4.4. Válasz. ***Oktató:*** Ha megengedünk végtelen struktúrákat, akkor `X` és `X+1` azonos alakra hozható a fenti behelyettesítéssel. A „klasszikus” matematikai logika persze nem enged meg ilyen végtelen formulákat. A Prolog rendszerek viszont hatékonysági okokból engedik meg ezeket. Ugyanis Prolog eljárások paramétereinek átadasa legtöbbször egy változó és egy (esetleg jó bonyolult) Prolog kifejezés egyesítését jelenti. Ha el akarnánk kerülni a fenti végtelen struktúra kialakulását, akkor ennek a tipikus egyesítési müveletnek a jelenlegi konstans futási ideje a bonyolult kifejezés méretével arányossá válna. Hiszen a végtelen (ciklikus) kifejezés kialakulását elkerülendő, ellenőrizni kellene, hogy az értéket kapó változó nem fordul elő az értékül adott kifejezésben. Ezt hívják előfordulás-ellenőrzésnek (occurs-check).

Erről olvashatnak a jegyzetben és a fóliákon is. Lásd még az `unify_with_occurs_check/2` beépített eljárást.

Természetesen nem szabad figyelmen kívül hagyni azt sem, hogy az = beépített eljárás **nem** aritmetikai kifejezéseknek tekinti az argumentumait.

4.5. Kérdés. Van olyan operátor, ami az egyesíthetőséget vizsgálja, de nem egyesít? tehát egy egyszerűbb megoldás kellene erre:

```
\+(Valami \= Masik)
```

4.5. Válasz. ***Oktató:*** Ez a megoldás nem is olyan csúnya. Általánosítva: a `\+ \+ <cél>` akkor és csak akkor sikerül amikor `<cél>`, de nem helyettesít be változót (az `X \= Y` beépített eljárás nem más mint `\+ X = Y`). Réges-régen, amikor nem volt még szemétgyűjtes a Prolog megvalósításokban, a \+\+ <cél> szerkezetet azért is használtuk, mert a `<cél>` futásához szükséges vermet is felszabadítja.

De van más megoldás is, megfelelő beépített eljárások használatával.

1. Speciális esetben, ha csak azt akarjuk megvizsgálni, hogy `Valami` adott funktorú-e akkor a

   ```
   functor(Valami, Nev, Argszám)
   ```

   beépített eljárás használható. Például a `Valami = f(_,_,_,_)` hívás helyett használható: `functor(Valami, f, 4)`.
2. Általánosabban: a `copy_term(Kif, Masolat)` elkeszíti `Kif` egy masolatát úgy, hogy minden változót szisztematikusan új változóra cserél, és ezt a másolatot egyesíti `Masolat`-tal. Tehát ha egy Valami kifejezésről el akarjuk dönteni, hogy illeszkedik-e egy `Minta` mintára, akkor ez

   ```
   copy_term(Minta, Valami)
   ```

   hívással eldönthető, anélkül, hogy változó-behelyettesítesek jönnének létre.

4.6. Kérdés. A következő Prolog hívás miért nem hiúsul meg:

```
| ?- \+ \+ X=a, X=b.
```

4.6. Válasz. ***Hallgató:*** Nézzük a `\+` definícióját:

```
\+(A) :- A, !, fail.
\+(_).
```

Ebből azt kell látni, hogy ha az `A` célon sikeresen túljutunk, akkor VÁG és meghiúsul, ha viszont `A`-ban valami bibi volt, és meghiúsul (még a vágó elött), akkor a Prolog rátér a második klózra, ami mindig teljesül. Viszont így egyik esetben sem lesz behelyettesítés... Ha duplán negálva adsz meg egy célt, akkor ezzel azt írod elő, hogy a célnak sikeresen le kell futnia, de az ehhez szükséges változó-behelyettesítések nem maradnak meg.

Ha így futtatod: `X=b, \+ \+ X=a`, akkor meghiúsul, hiszen `X` akkor már egyesítve lett `b`-vel, így `b\=a` miatt lehal a dolog. De a te esetedben `X`-nek még nem volt értéke, ezért történhetett az meg hogy sikeresen futott le...

Oktató: Fontos megjegyezni, hogy a `','` operátor gyengébben köt mint a `\+`. Ha a `| ?- \+ \+ (X=a, X=b).` hívást futtatjuk, akkor meghiúsulást kapunk, hiszen a kétszeres negáció belsejében egy meghiúsuló hívás lesz.

4.7. Kérdés. A PL házim készen van, egy kis apróságtól eltekintve: Listák listájaként keri a bemenetet, mert nem tudok mit kezdeni a `1-2` alaku adatpárral. Mivel nem atom, nem tudom szépen szétbontani.

4.7. Válasz. ***Oktató:*** Pont fordított a helyzet. Az atom – mint neve is mutatja – egy nehezen szétbontható dolog, hiszen ehhez beépített eljárást kell használni. Az összetett adatokat viszont könnyű szétbontani és összerakni, ehhez elég az egyesítés: Például: az „`1-2 = X-Y`” egyesítés eredménye „`X = 1, Y = 2`”.

A kérdező folytatja: Hogyan lehet szétbontani, vagy legalábbis hogyan álljak neki, hol keressem a könyvben?

Oktató: A könyvben a „Paraméter-átadás — egyesítés” és az „Összetett adatstruktúrák” című szakaszok segítenek. (Általában „A Prolog nyelv alapjai” c. fejezet nélkülözhetetlen.)

Az alábbi eljárás párok Kulcs-Érték párok listájából Kulcsok és Értékek listáját állítja elő (visszafelé is használható, ha a 2. és 3. paraméterben azonos hosszúságú listákat adunk meg, akkor ezeket egy listává „zippzárolja”):

```
unzip([], [], []).
unzip([K-V|KVs], [K|Ks], [V|Vs]) :-
        unzip(KVs, Ks, Vs).
```

4.8. Kérdés. A feladat a következő:

```
p(1, 1).
p(3, 1).
p(_, 2).

q([H|T], A) :- p(H, A).
q([H|T], A) :- q(T, A).

| ?- q([1,2,3,4], X).
```

Kérdésem: miért hiúsul meg az az eset, amikor már csak az ures lista a T? A „_” (aláhúzás) nem vonatkozik rá?

4.8. Válasz. ***Oktató:*** Az aláhúzás a `p/2` eljárás első paraméterpozícióján van. A kitűzott feladat a `q/2` eljárás meghívását kérdezi. A `q/2` eljárás mindkét klóza nem üres listát tartalmaz az első paraméterpozícion. Tehát ha üres listával hívja a `q/2`-t, vagy a rekurzió folyamán eljut az üres listáig, akkor a `q([], X)` meghiúsul.

4.9. Kérdés. A következő egyesítéssel nem boldogulok:

```
g(H-G, [H, G]) = g(U*T-a, [T*2|S])

G=a
H=?
S=a
T=?
U=2
```

Van valakinek valamilyen ötlete a hiányzókra?

4.9. Válasz. ***Hallgató:*** A `g` első argumentumában a `-` operátor miatt egyértelmű lesz, hogy `G=a`, és az, hogy `H=U*T`. De ekkor a listás részt nézve `H=T*2` is igaz, ez pedig csak úgy lehet, hogy `U=T=2`, és ekkor `H=2*2`.

4.10. Kérdés. Lenne néhány kérdésem amire nem tudtam rájönni:

```
.(A,X) = [3,2,1].
```

4.10. Válasz. ***Hallgató:***

```
[3,2,1]='.'(3,'.'(2,'.'(1,[])))='.'(A,X)
```

Mintaillesztés után:

```
A=3, X='.'(2,'.'(1,[]))=[2,1]
```

A kérdező folytatja: `a(1,2) = X(1,2).`

Hallgató: Hiba. Mivel a nagybetűs kifejezés változót jelöl, de struktúranév helyén szerepel, ezért ez szintaktikus hiba.

A kérdező folytatja: `X*X = _*1+2.`

Hallgató: Az _ egy semmis változót jelöl (ez speciális a Prologban, egyfajta `/dev/null`, mivel pl. a „`_=a,_=b.`” igaz). Itt csak azt jelzi, hogy az _ változó behelyettesítése nem érdekes.

Írjuk fel a dolgokat alapstruktúra-alakban:

```
*(X,X) = +(*(_,1),2)
```

Látható, hogy a struktúrafejek nem egyeznek meg, tehát meghiúsul a hívás.

A kérdező folytatja: `[X|[]] = [[1]].`

Hallgató: `'.'(X,[]) = '.'('.'(1,[]),[])`
Mintaillesztés után kijön, hogy: `X='.'(1,[])=[1]`

Oktató: Ugyanezt az eredményt a következő érveléssel is megkaphatjuk. Az egyesítés baloldalán egy egyetlen `X` elemet tartalmazó lista áll. A jobboldalon egy olyan lista van amelynek egyetlen eleme `[1]`.

5. fejezet

Struktúrák, operátorok

5.1. Kérdés. Úgy tűnik, hogy automatikus kiértékelés nincs: az `f(1+2)` hívás és az `f(3)` hívás nem ugyanaz.

5.1. Válasz. ***Oktató:*** Mint már sokszor elmondtam, az `1+2` jelsorozat Prologban nem kiértékelendő aritmetikai kifejezést, hanem egy összetett adatstruktúrát jelent. Ezt írhatjuk `+(1,2)` alakban is, tehát egy olyan struktúra-kifejezésről van szó, amelynek neve a `+` atom, argumentumai az 1 és 2 számok. Ezt a struktúra-kifejezést **csak** az aritmetikai beépített eljárások értelmezik aritmetikai kifejezésként. Például az `'is'` eljárás a második argumentumát aritmetikai kifejezésként kiértékeli, és az eredményt egyesíti az elsővel.

A kérdező folytatja: Ez utóbbit az `X is 1+2, f(X)` segítségével tudom csak elérni.

Oktató: Így van. Ez valószínüleg szokatlan, de vegyék figyelembe, hogy a Prolog egy relációs nyelv, tehát a függvenyeket is relációkként kell kezelni. Tulajdonképpen az `'is'` eljárás csak egy kényelmi szolgáltatás, az első Prolog rendszerek, relácios szemlélettel összhangban, az egyes aritmetikai műveletekre külön-külön predikátumokat használtak, pl.

```
times(X, Y, Z), plus(Z, 1, U)
```

kellett írni, a mostani `U is X*Y+1` helyett.

Vannak a Prolognak olyan kiterjesztései, amelyek függvényjelölést is megengednek. Például ilyen a Mercury rendszer. A DP tárgy keretében azonban maradjunk a (többé-kevesbé) szabványos Prolognál...

5.2. Kérdés. A Prolog jegyzet GY2. feladat 1. részének megoldásában a 3 konstans kifejezés alapstruktúra-alakjaként a `.(3,[])` kifejezés szerepel. Ezek szerint egy konstans megegyezne az ugyanebből a konstansból álló egy elemű listával, vagy ez csak egy sajtóhiba?

Oktató: A 3 konstans alapstruktúra-alakja a 3. Nincs sajtóhiba, Ön értette félre a választ.

A feladatban a `[w+B,3]` lista alapstruktúra-alakját kérdeztük, ez `.(+(w,B),.(3,[]))`.

A válaszban szereplő `.(3,[])` nem a 3-nak, hanem a `[3]`-nak az alapstruktúra-alakja. Ez a `[3]` lista pedig nem más, mint a `[2|[3]]` lista farka.

Lehet, hogy ön összezavarja a `[H|T]` szerkezetet az `[E1,E2]` szerkezettel. Az első egy olyan lista, amelynek feje H és farka T, míg a második egy kételemű lista.

Egy egyszerűbb példán:

```
[2,3] = [2|[3]] = [2|[3|[]]] = .(2, .(3, []))
```

5.3. Kérdés. Mi a vesző (logikai ÉS) alapstuktúra alakja? Például hogy kell átírni a `p:-q,r` kifejezést? `:-(p„(q,r))` -re gondoltam de két vesző mást jelent.

5.3. Válasz. ***Hallgató:*** Ez majdnem jó. A tökéletes megoldás: `:-(p,',',(q,r))`.

5.4. Kérdés. Az előadáson szerepelt egy függvény, ami egy szintaktikus édesítőszereket tartalmazó kifejezést átalakított alapstruktúra alakba. Mi volt ez a függvény?

5.4. Válasz. ***Hallgató:*** `write_canonical(_)`

5.5. Kérdés. Tudna nekem valaki segíteni az alábbi kifejezés átírásában alapstruktúra alakba?

```
[9,0'd]
```

A vessző mit jelent?

5.5. Válasz. ***Hallgató:*** A `0'`x együtt jelenti egy x karakter (ascii) kódját. Tehát a `0'd=100` vagyis az alak, amit keresel:

```
.(9,.(100,[]))
```

5.6. Kérdés. A `{p}` kifejezésnek hogyan kell felírni az alapstruktúra alakját?

5.6. Válasz. ***Hallgató:***

```
| ?- write_canonical({p}).
{}(p)
yes
```

Ha valakinek inkább tetszik, írhat `'{}'(p)`-t. Ez tehát egy 1-argumentumú struktúra, amelynek neve `'{}'`.

5.7. Kérdés. Adott ez a példa:

```
p:-q,r,s
```

A feladat alapstruktúráva alakítás. A megoldás szerintem:

```
:-(p,',',(',',(q,r),s))
```

De az ets ezt nem akarja elfogadni. Azt mondja alapstruktúra alak, de nem ugyanaz. Mit csinálok rosszul?

5.7. Válasz. ***Hallgató:*** Csak a zárójelezés irányát tévesztetted el, különben jó. A ’,’ operátor `xfy` fajtájú, tehát jobbról balra zárójelez (így könnyű egy célsorozat első célját levenni ...) Tehát a helyes eredmény így néz ki:

```
:-(p,’,’(q,’,’(r,s)))
```

Egyébként `write_canonical()` sokat segít ilyenkor. Bár a `write_canonical(p:-(q,r,s)).`-t se a SICStus, se a gnu Prolog nem ette meg, ellenben a `write_canonical((p:-q,r,s)).`-t már igen.

Oktató: Az argumentumlistában zárójelezés nélkül a Prolog ugyanis nem enged meg 999-nél nagyobb prioritású operátort. Ha ugyanis megengedné az 1000-es prioritású ’,’ operátort, akkor a `p(1,2,3)` jelsorozatot kétféleképpen is lehetne elemezni: mint egy `p/3` funktorú struktúrát, vagy mint egy `p/1` funktorú struktúrát, amelynek egyetlen argumentuma az `(1,2,3)` kifejezés. De mivel az argumentumban az 1000-es prioritású operátor nem megengedett, `p(1,2,3)` egyértelműen `p/3` funktorú struktúrát jelent, míg a másik eset leírásához a duplán zárójelezett `p((1,2,3))` jelsorozatot kell használni.

5.8. Kérdés. Arra lennék kíváncsi, hogy ha `[c|E]` kanonikus alakjaként `.(c,.(E,[]))`-t írom be az ets-be, miért nem fogadja el?

5.8. Válasz. ***Hallgató:*** Ha a `.(c,.(E,[]))` kifejezést visszaalakítod, akkor a `[c,E]` alakot kapod, ami nem azonos a `[c|E]` alakkal.

A helyes megoldás: `[c|E] == .(c,E)`

Lásd még az 5.2 kérdést.

5.9. Kérdés. `p:-q` , ennek a kifejezésnek, miért nem `:(p,-(q))` vagy `:(p,*(-1,q))` az alapstruktúra alakja, illetve egyáltalán mi ha nem ez?

5.9. Válasz. ***Hallgató:*** Az implikáció egy operátor „:-”, ezért `:-(p,q)`

5.10. Kérdés. Ennek `(p:-q,r,s)` mi az alapstruktúra-alakja? Mert ettől kiakad a `write_canonical()` is.

Mivel a `:-` operátor precedenciája 1000 feletti, ezt a kifejezést még egyszer be kell zárójelezni, ha argumentumként akarod szerepeltetni:

```
write_canonical((p:-q,r,s))
```

És a válasz:

```
:-(p, ’,’(q, ’,’(r, s)))
```

5.11. Kérdés. Hogyan lehet negálni egy visszatérési értéket Prologban? A `member(X,Y)`-ra kellene, hogy akkor menjen tovább, ha `X` nem eleme `Y`-nak...

5.11. Válasz. ***Hallgató:*** A negálast a `’\+’/1` eljárással végezzük. Tehát:

```
\+ member(X, Y)
```

(Fontos, hogy ez a fajta hívás nem végez behelyettesítést!)

5.12. Kérdés. Hol található a precedencia táblázat, és hogyan editálhatóak a precedenciák?

5.12. Válasz. ***Oktató:*** Alapvetően nem javaslom a beépített operátorok precedenciájának megváltoztatását, mert ez sok helyen gondot okozhat.

5.13. Kérdés. `% :- tort ==  számlalo # nevezo`

Ez ugye egy (számláló, nevező) struktúrat akar jelölni, vagyis a # itt egy vesszőt helyettesít?

5.13. Válasz. ***Hallgató:*** Nem, a # a struktúra neve. Itt # egy infix operátor.

5.14. Kérdés. Ha a kifejezésben egész számok, illetve a `+`, `-`, `*`, `/` jelek szerepelhetnek, akkor mit keres a példák között például ez:

```
| ?- tortérték(4/(-6), E).
E = -2#3 ? ;
no
```

(a hangsúly a zárójel jelenlétén van, mint megengedett/nem megengedett karakter)

5.14. Válasz. ***Oktató:*** A kerek zárójelpár minden legális kifejezés körül megengedett. Zárójelekkel felül tudjuk bírálni az operátor-precedenciát, de akár feleslegesen is berakhatunk zárójeleket.

5.15. Kérdés. A `->` operátor merre köt? Jobbra vagy balra?

Az alábbi módokon szokás a választ kideríteni:

```
% sicstus -f
SICStus 3.10.0 (x86-linux-glibc2.2): Tue Dec 17 15:24:21
CET 2002
Licensed to iqsoft.hu
| ?- current_op(A,B,->).
A = 1050,
B = xfy ? ;
no
| ?- write_canonical((a->b->c)).
->(a,->(b,c))
yes
```

Az `xfy` ugyebár azt jelenti, hogy a jobb oldalán enged meg 1050-es (vagy erősebb) operátorokat. Ezt én jobbra kötésnek nevezem.

6. fejezet

Listák

6.1. Kérdés. Olvasgattam a pdf-eket de nem nagyon volt világos hogy a `[]` és `|` mit is jelent.

6.1. Válasz. ***Oktató:*** Nos, a Prologban nyelvi szinten vannak definiálva a listák, pl. a C-vel ellentétben (itt viszont tömbök nincsenek). A lista definíciója (alapstruktúra alak):
`.(Listaelem,Lista)` vagy `[]` Tehát egy lista egy olyan valami, amit egy listaelem, és egy lista alkot, vagy az üres lista (ez a `[]`); éppen ezért minden véges lista utolsó eleme az üres lista (hasonlíthatom ezt a C-beli null-pointerrel jelzett végű listához). Mivel a listák ilyesfajta felírása, hogy `.(a,.(b,.(c,[])))` kissé fáradtságos, ezért létezik egy „szintaktikusan édesített" alak, ami emberibben néz ki: `[Listaelem|Lista]`.
Ebben a fenti listát így írjuk: `[a|[b|[c|[]]]]` Mivel ez - bár kicsit jobban olvasható - még mindig nem mindig kényelmes (értsd: amikor nem csak a feje kell a listának), ezért van egy még további édesített alak:

```
[Listaelem1,Listaelem2,...,ListaelemN] =
[Listaelem1|[Listaelem2|[...| [ListaelemN|[]]...]]]
```

vagy

```
[Listaelem1,Listaelem2,...,ListaelemN|Lista] =
[Listaelem1|[Listaelem2| [...|[ListaelemN|Lista]...]]]
```

Ebben az alakban a fenti listánk már így néz ki: `[a,b,c]` A listák belső ábrázolása természetesen a fenti alapstruktúra alakban történik, csak nekünk kell kevesebbet gépelni.

6.2. Kérdés. Az lenne a kérdésem, hogy hogyan lehetne az `append`-et rávenni arra, hogy 1 megoldás utan leálljon és ne kezdjen újabb behelyettesítésekbe? (vagy csak triviális válasz van (=sehogy)?) Az a problémám, hogy össze szeretnék fűzni 3 listát 1 listával az előadáson latott `append/4` segítségével, de végtelen ciklusba esik, miután az első megoldást kidobja.

6.2. Válasz. ***Oktató:*** Az alabbi megoldás az előadás 6-14 foliajan szerepel.

Nem mindegy, hogy az `append/4` belsejében lévő `append/3` hívások milyen sorrendben vannak. Ez megoldás nem hoz létre végtelen választási pontot (és így nem esik végtelen ciklusba az első néhány megoldás után), ha az első két lista vagy az eredmény lista zárt végű. A futási ideje az első két lista hosszának összegével arányos.

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

6.3. Kérdés. Gyakorló rendszerben találkoztam a következő alapstruktúra-alak példával:

`[9,6]`

erre beírom:

`.(9,6)`

erre kiadja:Alapstruktúra alak, de nem ugyanaz...
Mire nem figyeltem?

6.3. Válasz. ***Hallgató:*** Arra, hogy a lista struktúra második argumentuma is egy lista. A te megoldásodra ez nem igaz, a 6 nem lista.

A helyes megoldás: `.(9,.(6,[])).`

6.4. Kérdés. Ha egy listából,aminek az elemei listák, szeretnék olyan listát, amelynek az elemei az eredeti elem-listák elemei akkor mi is volt a megoldás?

6.4. Válasz. ***Oktató:***

```
:- use_module(library(lists)).
flat([], []).
flat([L0|X], F) :-
        append(L0, F0, F),
        flat(X, F0).
```

6.5. Kérdés. Hol találok arra nézve információt, hogy hogyan lehet egy Prolog listát rendezni?

6.5. Válasz. ***Oktató:*** Ez benne van a jegyzetben is „A legfontosabb beépített eljárások” fejezet 5.7 „Listakezelés” szakaszában. Természetesen a tárgymutatóban is benne van, ott persze angolul kell keresni (sort). Ezen kívül a SICStus kézikönyvben is benne van.

A kérdéses beépített eljárások: `sort/2, keysort/2`

6.6. Kérdés. Az lenne a kérdésem, hogy Prologban hogyan lehet két lista közös elemeit LISTA formájában megkapni (tehát nem a `member(X, L1), member(X, L2)` módszer kell), vagy ha ilyen nincs, akkor hogyan lehet a fenti két `member`-es módszer eredményeit megszámolni?

6.6. Válasz. ***Hallgató:*** `findall(X, (member(X, L1), member(X,L2)), LISTA).`

6.7. Kérdés. Mitől lehet az, hogy a kivalasztóim végére memóriaszemét kerül? A honlapon fent lévő, egyszerű tesztesetre ezt irja ki:

```
E = 2
K = [right|_G743] ;
```

A többi esetnél is ugyanez a helyzet, a megoldások ettől eltekintve helyesek.

6.7. Válasz. ***Oktató:*** Az nem memóriaszemét, hanem egy nyílt végű lista. Az oka pedig az lehet, hogy a rekurzióban „elfelejti" lezárni a listát, tehát valahol, ahelyett, hogy üres listát adna vissza, kitöltetlenül hagyja az argumentumot.

6.8. Kérdés. Hogyan lehet Prologban egy nyílt végű listát lezárni? pl: hogyan kell a `[1,2,3,4|_] -> [1,2,3,4]` konverziot vegrehajtani?

6.8. Válasz. ***Oktató:***

```
close_list_0(L) :-
        once(length(L, _)).
                % Ez a cél így önmagában is használható...

close_list_1(L) :-
        is_list(L), !.

% is_list(L): L egy (tetszőleges) zárt lista.
is_list([]).
is_list([_|L]) :-
        is_list(L).

% A vágó bevitele az  is_list/1 eljárásba. Talán egy picit
% hatékonyabb az előzőnél.
close_list_2([]) :- !.
close_list_2([_|L]) :-
        close_list_2(L).
```

A kérdező folytatja: Illetve hogyan kell egy pontosan N elemű listát létrehozni?

6.8. Válasz. ***Oktató:*** Egy megoldás:

```
% n_elemu_lista(+N, ?L): L egy N elemu lista.
n_elemu_lista(0, L) :-
        !, L = [].
n_elemu_lista(N, [_|L]) :-
        N > 0, N1 is N-1, n_elemu_lista(N1, L).
```

De tökéletes a `length(L, N)` megoldás is.

6.9. Kérdés. Egy kérdésem lenne a kishffel kapcsolatban. Legyártottam egy megoldást ami az elemeket egyenként rakosgatja a listába... És ilyen a végeredmény:

```
| ?- számol2(5,2,E).

E = [[[[],1],0],1] ?
```

```
| ?- számol2(8,2,E).

E = [[[[[],1],0],0],0] ?

| ?- fszámol2(8,2,E).

E = [0,[0,[0,[1,_A]]]] ?
```

A köv utasítással:

```
Jegyek = [Tmp,Jegyek2].
```

6.9. Válasz. ***Oktató:*** Mint a mai előadáson elmondtam, `[Tmp,Jegyek2]` mindig egy **kételemű** lista, amelynek elemei `Tmp` és `Jegyek2`. Önnek valószínűleg `[Tmp|Jegyek2]`-re van szüksége, amely egy olyan lista, amelyek feje `Tmp` és a farka (a fej elhagyása után fennmaradó elemeinek listája) a `Jegyek2`.

Persze ettől valószínűleg még mindig nem lesz jó az `fszámol2` eljárása, mert a lista végén az `_A` változó és nem az üres lista áll.

Az `számol2` eljárás pedig azért sem jó, mert nem listát épít, hanem egy más szerkezetű adatstruktúrát. A lista ugyanis vagy üres (`[]`), vagy egy `.(Elem,Lista)` alakú struktúra, ahol `Lista` megint egy lista. Az Ön által épített struktúra viszont olyan (feltételezve, hogy a `[_,_]`-t `[_|_]`-ra cseréli), amely vagy `[]`, vagy egy `.(Istal, Elem)` alakú struktúra, ahol `Istal` megint ilyen szerkezetű. Bár az `Istal` struktúrában az elemeket balról jobbra olvasva fordított sorrendben kapjuk, mintha listává építettük volna ezeket. Az `Istal` nem az eredeti lista fordítottja!

6.10. Kérdés. Van nem balrekurzív megoldás arra, hogy egy lista végére illesszünk egy elemet?

6.10. Válasz. ***Oktató:*** Természetesen van. Mivel a lista megfordítására van jobbrekurzív megoldás, a szóbanforgó feladatnak is van.

Prologban a „természetes” megoldás jobbrekurzív, hiszen ez maga az `append/3`.

7. fejezet

Módszerek

7.1. Kérdés. Ha már megírtam a programot úgy hogy a megoldásokat egy listában determinisztikusan adja vissza hogyan lehetne hatékonyan nemdeterminisztikussá alakítani a member eljárás kihagyásával?

7.1. Válasz. ***Oktató:*** A kétféle megoldás kapcsolatárol elég sok szó esett az egyik előadáson. Általában a felsoroló eljárás megírása könyebb, mert kevesebb paraméterrel kell foglalkozni.

Lásd még a jegyzet 4.5 fejezetét, amely ezzel a kérdéssel foglalkozik.

7.2. Kérdés. A fejkommentezésnél kisebb problémába ütköztem. Mégpedig: mit kell odaírni, ha egy változó egyszerre be és kimenő is?

7.2. Válasz. ***Oktató:*** Ha a Mercury-szerű jelölést használja, akkor több moddeklarációt adhat meg, pl:

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

Ha a SICStus kézikönyv mod-jelölését használja, akkor a ? jelzi az egyaránt be- és kimenő paramétereket, pl.

```
% append(?L1, ?L2, ?L3)
```

7.3. Kérdés. Az lenne a kérdésem, hogy az indexelés, és mondjuk egyéb mindenféle trükkök (egyesítés a vágó után, nem a klózban... stb) tudják-e lényegesen javítani a programidő nagyságrendjét, vagy inkább kísérletezzek tovább vágóval és a keresési tér szűkítésén.

7.3. Válasz. ***Oktató:*** Az indexelés, determinizmus stb. javítása lineáris gyorsulást eredményez, míg egy „intelligensebb” keresési módszer ennél sokkal jobb eredményt is adhat. A keresési tér szűkítésére én nem elsősorban a vágót ajánlanám, hanem eleve olyan kerésesi algoritmust, amely esetén kisebb keresési teret kell bejárni.

7.4. Kérdés. Hosszú listákban keresés esetén érdemes dinamikus adatbáziskezelést használni? Valahogy így:

```
[data1,data2,data3|L]
```

helyett

```
inlist(data1).
inlist(data2).
inlist(data3)...
```

illetve `assert/1` és társai
Ez javíthat lényegesen a sebességen?

7.4. Válasz. ***Hallgató:*** Csak konzultálva lehet gyorsítast elérni, mert ugyebár csak lekérdezed, és kész. Persze betölteni sem kis idő, de azért én is tudtam egy 2-es szorzót elérni vele. (Sebességben, nem futási időben... :])

DE! A dinamikus predikátumok NEM fordithatóak, a listákkal dolgozók viszont igen. (A fordítás ugyebár kb. egy 10-es szorzó is tud lenni, mert nem lesz benne mindenféle trace info, meg ilyesmi. Szóval az állítólag olyan, mint a byte-kód.) A beadó rendszer viszont fordítás után próbálja futtatni a programot.

Továbbá az `asserta` Prologban nem visszaléptethető, vagyis erről is neked kell gondoskodni.

Oktató: Ha a kereséssel egy X -> Y függvényt kiván megvalósítani, akkor a `library(assoc)` könyvtár használatát ajánlom, amely logaritmikus időben AVL fákkal valósítja meg a keresést. Tehát:

```
| ?- use_module(library(lists)).
yes
| ?- _L = [x1-y1,x2-y2,x3-y3], member(x2-Y, _L).
Y = y2 ? ;
no
| ?-
```

helyett ezt:

```
| ?- use_module(library(assoc)).
yes
| ?- empty_assoc(_A0), put_assoc(x1, _A0, y1, _A1),
        put_assoc(x2, _A1, y2, _A2), put_assoc(x3, _A2, y3, _A),
        gen_assoc(x2, _A, Y).
Y = y2 ? ;
no
| ?-
```

7.5. Kérdés. A Prolog nagyházi közben éppen debuggolni illetve futási időt mérni szerettem volna, viszont a program egyelőre gyerek cipőben lévén a kimentre adja ki a megoldásokat, csak ebből van néhány (több ezer) :))

Kérdésem: Ki lehet-e valahogy kapcsolni azt, hogy amikor talált egy megoldást, akkor azt kiírja, és felhasználói beavatkozásra, „;”-ra várjon? Nekem az kéne, hogy a ; nélkül folyamatosan adja ki az eredményeket, ne kelljen minden megoldás utan ;-ot ütni!

Hallgató: Lásd a `findall/3` megoldásgyűjtő eljárást.

7.6. Kérdés. Hogyan lehet megcsinálni a következő szerkezetet Prologban?

```
utasitas1;
if kif
then utasitas2;
utasitas3;
```

azaz ha a `kif` igaz, akkor `ut1, ut2, ut3`. Ha hamis, akkor `ut1, ut3`. Nekem csak így megy:

```
utasitas1,
( kif ->
  utasitas2
; nothing
),
utasitas3, ...
```

7.6. Válasz. ***Oktató:*** Ha `nothing` alatt a egy `true/0` hívás értendő, akkor ez a helyes megoldás.

A kérdező folytatja: És hogyan lehet innen a `nothing`-ot kihagyni?

Oktató: A `true`-t nem lehet innen kihagyni. Ez nagyon nagy baj?

A kérdező folytatja: vagy:

```
  utasitas1;
  (kif-> utasitas2,utasitas3
       ; utasitas3),
```

Oktató: Ez is jó, csak egy kicsit nehezebben érthető és nehezebben karbantartható. (`utasitas1` után persze továbbra is `,` jelet kell írni.)

Hallgató: van ilyen is: `(->)/2` , ez a `Ha-akkor` szerkezet.

Oktató: Ha jól értem, arra utal, hogy a `;` és az utána következő rész elhagyható. Ez igaz, de:

```
(   Felt
->  Hívás
)
```

a

```
(   Felt
->  Hívás
;   fail
)
```

szerkezettel egyenértékű. Az eredeti kérdezőnek nem erre van szüksége.

Megjegyeznem, hogy a második szerkezet egy olyan `(;)/2` struktúra, aminek első argumentuma egy `(->)/2` struktúra. (Tehát `(->)/3` nem létezik, mint vezérlési szerkezet!)

Hallgató: De ha illesztéssel megoldható, akkor azzal jobb csinálni a feltételes dolgokat. Akkor illeszt, ha a klózra illeszthető, ha nem, akkor másik klózra illeszt...

Oktató: Amennyiben nem kényszerül vágó használatára, hogy a feltételt kifejezze, akkor tényleg ez az elegánsabb megoldás. Egyébként a vágó több odafigyelést igényel, mint a `(->)/2`-os szerkezet.

Hallgató: A fent említett két dolog hatékonyságát meg akartam mutatni valakinek, erre meg kellett lepődnom, mert a

```
f(0,1):-!.
        N1 is N-1,
        f(N1, F1),
        F is N*F1.
```

predikátum valahogy csak gyorsabban számol, mint a

```
f(0,F,F):-!.
f(N,F,A):-
        N1 is N-1,
        A1 is N*A,
        f(N1, F, A1).
```

OK, listáknál már kijön a várt eredmény, de a fenti jelenséget akkor sem éretem... Van valakinek bármilyen tippje??? (Elmeletileg a vágó meg az indexelés is ok, nem?)

Oktató: Azt hiszem van tippem:

Az első megoldás az `1*2*3*...*N` szorzatot számolja ki (pontosabban az `(N*...(3*(2*1)))` szorzatot, de a kommutativitás miatt a két műveletsor azonos időigényű) A második megoldás az `N*(N-1)*(N-2)*...*1` szorzatot számolja ki.

Ahhoz, hogy mérhető időt kapjon, egy nagyobb (1000-és nagysagrendű) bemenő számértékkel hívta az `f` predikátumokat, tehát a faktoriális értéke a nagyszám („bignum” > 2**25) tartományba esik. Namármost a nagyszámok szorzása nem konstans idejű, hanem függ a számok méretétől.

A második megoldásban sokkal nagyobb a részeredmények mérete, emiatt lassabb. A nagyszámok szorzásának időigénye mellett a rekurzió-szervezés igénye eltörpül.

Az indexelés pedig nem tökéletes. A leghatékonyabb futáshoz az első klózoknak így kell kinézniük

```
f(0, F) :- !, F = 1.
f(N, F) :-
  ....
```

ill.

```
f(0, F, A) :- !, F = A.
f(N, F, A) :-
   ...
```

Nézze meg a jegyzet 4.2.5. alfejezeteben lévő példákat és magyarázatokat.

7.7. Kérdés. Az alábbi kérdésem lenne: Amennyiben jól értettem, egy `Ha -> Akkor` kifejezésben szereplő vágó (ami csak `Akkor`-ban lehet) számára `Ha` nem átlátszó, amit, ha jól értettem, úgy kell érteni, hogy a vágó hatása nem terjed túl `Ha`-n, tehát az afölötti elágazásokat már nem vágja le. Jól értem ezt?

7.7. Válasz. ***Oktató:*** A SICStus Prolog rendszernek két üzemmódja van: sicstus és iso (alaphelyzetben sicstus, váltás: `| ?- set_prolog_flag(language, iso).`).

A sicstus üzemmód tiltja a vágót a '`ha -> akkor`' feltételes szerkezet '`ha`' részében, az iso nem. Mindkét üzemmód megengedi a vágót az 'akkor' részben.

Az 'akkor' részben lévő vágó a szülő célig vág. A 'ha' részben lévő vágó csak a 'ha' rész belsejében lévő választási pontokat szüntetheti meg.

[Megjegyzés: Véleményem szerint a Prolog ISO szabványosításában az egyik csúnya hiba volt, hogy megengedték a vágót a 'ha' részben. Ha valaki ott egy ilyen szerkezetet akar használni, akkor nevezze el azt, és ettől sokkal áttekinthetőbb lesz a kód.]

[Az implementációra vonatkozó megjegyzés: Mint azt az előadáson és a jegyzetben is elmondtam, a diszjukciókat és a diszjunktív feltételes szerkezeteket a SICStus segédeljárásokká alakítja. Ebben az átalakításban a '`->`' operátorból egy vágó keletkezik, hiszen a '`->`' elérésének pillanatában a diszjunkció többi ágát le kell vágni. Az '`akkor`' részben lévő vágó viszont nem maradhat közönséges vágó, mert neki nem csak a segédeljárásban, hanem az eredeti nagy eljárásban kell vágnia. Ezért lesz ebből un. távolbaható vágó.]

Az '`akkor`' részbeli vágó tulajdonképpen egy feltételes vágó. Példaként nézzük a klaszikus `ember/2` eljárást. Tegyük fel, hogy a `member/2`-t csak zárt listák elemeinek felsorolására (és esetleg ellenőrzésére) használjuk.

```
member(X, [X|_L]).
member(X, [_|L)]) :-
        member(X, L).
```

Ennek hibája, hogy amikor az utolsó elemet felsorolja, akkor még hagy egy választási pontot. Ha további megoldást kerünk, akkor még a második klózra rátér, és abban végrehajt egy `member(X, [])` hívást, amely meghiúsul. Tehát, ha az első klózban `_L` már `[]`, akkor egy zöld vágót be lehet illeszteni:

```
member2(X, [X|L]) :-
        (   L = [] -> !
        ;   true
        ).
member2(X, [_|L]) :-
        member2(X, L).
```

Megjegyzés: a példa azert egy kicsit erőltetett, mert ennél a megoldásnál jobb az, amit a `library(lists)` könyvtárban látunk, ahol az eredeti `member` hiányosságát nem vágóval, hanem indexeléssel küszöbölik ki:

```
member3(Element, [Head|Tail]) :-
        member_(Tail, Head, Element).

% auxiliary to avoid choicepoint for last element
member_(_, Element, Element).
member_([Head|Tail], _, Element) :-
        member_(Tail, Head, Element).
```

Remélem ez a kis tanulmány választ ad a kérdéskörre.

7.8. Kérdés. Lehet Prologban olyat csinálni, hogy egy paraméterről megmondom, hogy mi nem lehet? Pl. kapok egy `[X|Stb]` paramétert es biztosítani szeretném, hogy `X` nem lehet 5. Ha `X` $\overline{5}$-öt használom, az `fail`-lel tér vissza, ha `X`-nek még nincs konkrét értéke... van esetleg olyan operátor, ami úgy működik, ahogy az nekem jó lenne?

7.8. Válasz. ***Oktató:*** Az a predikátum, amit keresel, a `dif/2`, ami a következőképpen működik:

dif(?X,?Y)

Constrains X and Y to represent different terms i.e. to be non-unifiable. Calls to dif/2 either succeed, fail, or are blocked depending on whether X and Y are sufficiently instantiated.

Ez azt jelenti, hogy amennyiben a kérdés eldönthető (X és Y már értéket kaptak), akkor az eredmény meghiúsulás vagy siker. Ha még nem dönthető el, akkor egy daemon keletkezik, ami addig vár, amíg X és Y egyaránt értéket nem kapnak, és abban a pillanatban, amikor a második is értéket kapott, lesújt: ha X és Y azonosak, akkor meghiúsulást okoz azon a ponton, ahol a második változó is értéket kapott, egyébként pedig semmit nem csinál (és a daemon megszűnik).

A `dif/2` sajnos nem szabványos, de a SICStus Prologban megtalálható. Ezt a technikát, hogy bizonyos predikátumok lefutását későbbre halasztjuk korutin-szervezésnek (coroutining) hívjuk, és az ISO Prolog sajnos nem definiál ezzel kapcsolatban semmit. Viszont a legtöbb Prolog implementáció biztosít valamilyen lehetőséget, a legelterjedtebb a `freeze/2` predikátum.

A leírása elég egyszerű:

freeze(?X,:Goal) Blocks Goal until nonvar(X).

Tehát addig vár, amíg X értéket nem kap, és akkor lefuttatja Goal-t. Ezzel könnyen megírható a `dif/2`, a következő kódrészlet elméletileg ugyanazt csinálja:

```
dif(X,Y):-
        freeze(X,freeze(Y,X=Y)).
```

vagy egy kicsit terjengősebben:

```
dif(X,Y):-
        Test=(X≡Y),
        Goal=freeze(Y,Test),
        freeze(X,Goal).
```

A `freeze/2` megtalálható a SICStusban, a „GNU Prolog RH”-ban ami a GNU Prolog egy továbbfejlesztése, és az SWI-Prolog újabb verzióiban (5.2-ben még nincs, 5.4-ben már van).

8. fejezet

Magasabbrendű eljárások

8.1. Monológ. ***Hallgató:*** Talán még emlékeztek arra a Prolog előadásra, amikor a magasabbrendű eljárásokról volt szó. Ekkor vetődött fel a `map/5` és a `filter/4` eljárások különböző megvalósításainak hatékonysága. (Ld. jegyzet 81. oldal)

Készítettem egy programot, ami négy lehetséges megvalósítás futási idejét vizsgálja. Szeredi tanár úr javaslatára közreadom a tapasztalatokat, és mellékelem a programot is.

A `filter` eljárás négy megvalósítása:

```
1.
%filter(L,X,Pred,Fl):Fl az L lista X elemeinek Pred szerinti
%szűrése
filter_with_call(L,X,Pred,Fl):-
        findall(X,(member(X,L),call(Pred)),Fl).


2.
filter_without_call(L,X,Pred,Fl):-
        findall(X,(member(X,L),Pred),Fl).


3.
%a member_call_1  eljárást használja
filter_with_member_call_1(L,X, Pred, Fl):-
        findall(X, (member_call_1(X, L, Pred)), Fl).

%member_call_1(X,L,Pred): X eleme az L list nak és Pred igaz
member_call_1(X , L, Pred):-
        member(X, L), Pred.


4.
%a member_call_2 eljárást használja
filter_with_member_call_2(L,X, Pred, Fl):-
        findall(X, (member_call_2(X, L, Pred)), Fl).

%member_call_2(X,L,Pred): X eleme az L list nak és Pred igaz
```

```
member_call_2(X , L, Pred):-
        member(X, L), call(Pred).
```

A futási idők átlaga (50000 elemű listából a páros elemek kiszűrése):

	Consult	Compile
filter_with_call :	1264 ms	1196 ms
filter_without_call:	561 ms	523 ms
filter_with_member_call_1:	1296 ms	1252 ms
filter_with_member_call_2:	1286 ms	1242 ms

Az eredmények magyarázata:

A `filter_without_call` a leggyorsabb, hiszen az nem tartalmaz egy fölösleges `call` hívást, ami időigényes. A `call` szerepe az lenne, hogy hívássá alakítsa a predikátumot. Felvetődhet a kérdés, hogy akkor `call` nélkül hogyan működik mégis az eljárás. A megoldás az, hogy a `findall` már eleve tartalmaz egy `call`-t. (Ld. jegyzet 85. oldal, P22 példa).

A másik érdekesség, hogy a `member_call_1` és a `member_call_2` eljárásokat használó megoldások futási idejei gyakorlatilag nem különböznek. Pedig a `member_call_2` egy `call` hívással többet tartalmaz. Ez azonban csak látszólag van így! Ugyanis betöltéskor a Prolog a `member_call_1`-et is kiegészíti egy `call`-lal. Ez betöltés után megnézhető a `listing` eljárás segítségével : `listing(member_call_1/3)`.

A harmadik észrevétel, hogy ha `compile`-lal töltjük be a programot, akkor valamelyest gyorsul a végrehajtás. Ennek az az oka, hogy a rendszer lefordítja a konjunkciót.

A `map` négy megvalósítása egészen hasonló a `filter`-hez, a mért futási idők (50000 elemű lista elemeinek négyzetre emelése):

	Consult	Compile
map_with_call:	1699 ms	1648 ms
map_without_call:	869 ms	846 ms
map_with_member_call_1:	1712 ms	1659 ms
map_with_member_call_2:	1701 ms	1615 ms

Tehát ebben az esetben is ugyanazok a tendenciák figyelhetők meg.

9. fejezet

Házi feladat

9.1. Kérdés. A keretprogram mint modult használja a mi „programunkat”?

> **9.1. Válasz.** ***Oktató:*** Nem, és pont ezért ilyen „trükkös” az alábbi sor. Azért nem akartuk a hallgatoságra kényszeríteni a modulok használatat, hogy ez ne akadályozza őket a megoldás korai elkezdésében.

9.2. Kérdés. Próbáltam használni a Prologos keretprogramot. Legutóbbi kísérletem így esett:

```
| ?- consult('C:/WIN/Asztal/dp01sker/ksatrak.pl').

{consulting c:/win/asztal/dp01sker/ksatrak.pl...}
{module satrak_keret imported into user}
{loading c:/program files/sicstus prolog/library/lists.po...}
{module lists imported into satrak_keret}
{loaded c:/program files/sicstus prolog/library/lists.po in
module lists, 60 msec 11136 bytes}
{consulted c:/win/asztal/dp01sker/ksatrak.pl in module
satrak_keret, 110 msec 28792 bytes}

yes
| ?- consult('C:/Dokumentumok/satrak.pl').

{consulting c:/dokumentumok/satrak.pl...}
{consulted c:/dokumentumok/satrak.pl in module user, 0 msec
-120 bytes}

yes
| ?- megold('/WIN/Asztal/dp01sker/tests/test1d.txt',
'/WIN/Asztal/ered.txt').
{EXISTENCE ERROR: ensure_loaded(user:satrak) - arg 1: file
satrak does not exist}
```

A kódomban a `satrak/2` -t megvalósítottam. Mi lehet a probléma?

9.2. Válasz. ***Oktató:*** A `megold/2` eljárás a keretben **maga** kísérli meg betölteni a `satrak` állományt. Ha a kurrens könyvtárban nem találja, a fenti hibát adja ki. Ha nem akarja, hogy a `megold` töltse be a satrak állományt, akkor kommentezze ki az `ensure_loaded(...)` hívást a `ksatrak.pl` file-ban, és akkor nem kell „helyben” meglenni a `satrak.pl`-nek.

9.3. Monológ. ***Oktató:*** A Prolog és SML könyvtárak használatáról

A házi feladatban bármelyik standard könyvtárat használhatják, mindkét nyelv esetében.

Prolog: A helyes deklaráció:

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

pl.

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

(A `:- use_module('lists.pl').` csak akkor működik, ha mellékeli a forráshoz, de ennek nem nagyon örülnénk.)

Ha a könyvtárnak csak bizonyos predikátumaira van szükségük, használhatják a

```
:- use_module(library(Könyvtar), [pred/1, pred2/3, ...]).
```

deklarációt, ahol a listában az importálandó eljárások funktorait kell feltüntetni.

9.4. Kérdés. Mit lehet tenni, ha megírtam a nagyházit, és korrektül működik, csak nagyon lassú (a példában megadott csigára 45 másodperc kell neki), pedig nem lassú a gépem ?

9.4. Válasz. ***Oktató:*** Az a tippem, hogy egy generate-and-test jellegű keresést valósít meg, tehát egy nagy keresési teret jár be, és minden megoldásra ellenőrzi a feltételek teljesülését. Próbálja meg minél korábban ellenőrizni a feltételeket, hogy a keresési teret csökkentse!

Általánosabban szólva: a legfontosabb a (keresési) algoritmus és a hozzá kapcsolódó adatábrázolás jó megválasztása. Egy (egyértelmű) csiga feladványt (mint amilyen a kiírásban is szerepel) az ember maga is meg tud oldani, következésképpen egy gépnek töredékmásodperc alatt kell(ene) végeznie. (Triviális) javaslatom: „kézzel” kísérelje meg megoldani a feladatokat, és a tanult módszereket próbálja átültetni a megoldó algoritmusba.

9.5. Kérdés. Azt szeretném megkérdezni, hogy miért nem találja meg a Prolog házit tesztelő program a `no_doubles/1` eljárást, mikor az (legalábbis nálam) a `lists` beépített könyvtár része?

9.5. Válasz. ***Oktató:*** A könyvtárakat be kell tölteni használat előtt, pl. a `lists` könyvtárat a `:- use_module(library(lists)).` direktívával. Lehet, hogy a keretprogram ezt már megtette, de ezt nem szabad kihasználni; mindekinek magának kell gondoskodnia az igényelt könyvtárak betöltéséről.

A kérdező folytatja: A `length` eljárást például tudom használni, de ezt nem.

Oktató: A `length/2` eljárás beépített és nem könyvtári, ezért tudja használni.

9.6. Kérdés. Azt lehet tudni, hogy egy ügyes Prolog program nagyjából mennyi idő alatt oldja meg a nagy hazifeladatot? Ugye nem az sml nagysagrendjében van a sebessége?

9.6. Válasz. ***Oktató:*** Ha a megvalósított algoritmus jó, nincs lényeges eltérés egy ügyes Prolog- és egy ügyes SML-program futási ideje között.

9.7. Kérdés. Valaki meg tudná nekem mondani, hogy az alábbi hiba pontosan mit jelent?

```
25. teszteset, időlimit =   2 sec
---------------------------------
% loading
/mnt/store/dp/ets/guts/hwks/05s/khf_pl3/MullerViktor.MNWJK6/eval.ql...
%  module eval imported into user
Cél: kisletra([])
X értékeit keressük.
1 megoldást vártam: X,
2 megoldást kaptam: A,B.

>>>>>A program lefutott, a megoldás HIBÁS.
```

A kódban van `kisletra([]).` klóz. SICStus 3.12 alatt nálam működik és csak annyit mond, hogy „yes”.

9.7. Válasz. ***Oktató:*** Ha nem tévedek, arról van szó, hogy a tesztkörnyezet egyszeres sikert vár, míg az ön programja kétszeresen sikerül. Amikor interaktív módban futtatja a kérdést, azért lát ebből a kettőből csak egyet, mert a SICStus toplevel környezet, ha nincs változó a kérdésben, levágja a többszörös sikereket. De be lehet csapni a következő trükkel:

```
X=X, kisletra([]).
```

Hogy egy egyszerűbb példán szemléltessem:

```
| ?- (true;true).
yes

| ?- X=X, (true;true).
true ? ;
true ? ;
no
```

9.8. Kérdés. Az alábbi tesztesetre nem jól működik a program:

```
| ?- plus(2,7,9).

A várt eredmény: siker,
A kapott eredmény: 2-szeres siker.
```

9.8. Válasz. ***Oktató:*** Ha jól sejtem, nem érti mit jelent ez.

Ezt a hibajelzést olyankor adja a gyakorló rendszer, ha az Ön által beadott megoldás egy eldöntendő kérdésre többszörösen is sikeresen fut le. Tehát ha a sikeres lefutás után visszalépést kényszerítünk ki, akkor újra sikerül. A fenti példában ez összesen két sikeres lefutást jelentett.

A többszörös siker egy alattomos hiba, hiszen csak úgy derül, ki, ha összetettebb programban használjuk a predikátumot. pl:

```
| ?- plus(2, 7, 9).

yes
| ?- plus(2, 7, 9), member(X, [1,2]).

X = 1 ? ;

X = 2 ? ;

X = 1 ? ;

X = 2 ? ;

No
```

10. fejezet

Emacs

10.1. Kérdés. Odáig már sikerült eljutnom, hogy az Emacs felismeri a Prologot, és hajlandó is a .pl kiterjesztésű fájlokat a Tanár úr gépén az előadásokon látott módon megjeleníteni (különböző szűnek a klózok egyes részeire, stb.) Szintén megjelenik a menüben a Prolog pont, de akár a consult, akár csak simán a run prolog opciót választom mindig a `Searching for program: No such file or directory: sicstus` hibaüzenetet kapom. A Prolog leírása szerint ez normális is, egy bizonyos `EPROLOG` nevű változót kellene beállítani úgy, hogy abban a `sicstus.exe` elérési útja legyen:

„If the shell command sicstus is not available in the default path, then it is necessary to set the value of the environment variable EPROLOG to a shell command to invoke SICStus Prolog. This is an example for C Shell:

`setenv EPROLOG /usr/local/bin/sicstus`”

Ezt eddig sajnos nem sierült megoldani, sem a fenti sor, sem pedig a _emacs fileba beírt `(setq EPROLOG "c:/prolog/bin/")` sor nem használt semmit, pedig a SICStus ott van...

10.1. Válasz. ***Oktató:*** Ez nem környezeti változót állít, hanem egy LISP változót, ráadásul nem a program nevét állítja be, hanem a könyvtárának a nevét. Ezért inkább az alábbival kéne próbálkozni:

```
(setenv "EPROLOG" "c:/prolog/bin/sicstus")
```

A leírás arról beszél, hogy egy shellben hogyan kell beállítani a környezeti változót. Windowson ezt mindenféle ablakokban csinélja az ember. De ha valaki mindenáron emacsban akarja beállítani, ez így történik.

Oktató: Ennek talán szebb (és mindenképpen „emacsosabb”) változata a következő:

```
(setq sicstus-program-name "c:/prolog/bin/sicstus")
```

vagy egy fokkal komplikáltabb, de a névtérrel spóroló formája:

```
(add-hook 'prolog-mode-hook'(lambda () (setq sicstus-program-name
"c:/prolog/bin/sicstus")))
```

10.2. Kérdés. Tud valaki olyan progit, ami ismeri a prolog szintaktikáját, szép színes mint az órán, valamint win alatt is fut.

10.2. Válasz. ***Hallgató:*** Az emacs létezik windows alá is. Sajnos pesze nem túl windowsos a kezelése, de ha megszokod, akkor nagyon jól használható.

10.3. Kérdés. Azt szeretném kérdezni, hogy valakinek sikerült-e már syntax highlightingot varázsolni emacsban?

10.3. Válasz. ***Oktató:*** Igen, többeknek.

A kérdező folytatja: Mit csinálok rosszul?

Oktató: Ezt nehéz megmondani. A SICStus kézikönyv alapján működni szokott, még nem láttam ellenpéldát.

A `font-lock-maximum-decoration` változó értéke micsoda? (Érdemes t-re állítani.)

A modeline tartalmazza a „Prolog[SICStus]” szöveget?

11. fejezet

Egyéb

11.1. Kérdés. Azon gondolkodom, hogy mondjuk egy

```
ezpiros( piros( _ )).
```

direktíva ugye minden `piros/1` funktorú paraméterre teljesül, egyébként meg bukik. Lehet olyat csinálni, hogy pont fordítva legyen?

```
eznempiros( A ):-
     \+ ezpiros( A ).
```

11.1. Válasz. ***Oktató:*** Ez így nagyon jól használható annak **vizsgálatára**, hogy valami nem `piros/1` funktorú kifejezés. Generálni viszont nem lehet vele az összes ilyet (nem is lenne reális...).

A kérdező folytatja: Mármint ezt a funkcionalitást szeretném, de csak mintaillesztéssel. Tehát `eznempiros( <valami trukkos> ).`

Oktató: Ez így nem megy. De miért is kellene feltétlenül fejillesztéssel megoldani a feladatot?

11.2. Kérdés. A feladat: egy kifejezésből visszaadni azt a legbővebb részkifejezést, amiben csak `+` és szám van. Miért nem működik? A negáció hogy működik? Miért lesz végtelen ciklus `egysz(a,A).` meghívására?

```
egysz(X,X):- \+ (var(X);atom(X)).

egysz(X,X):- number(X).

egysz(A+B,C):-
  egysz(A,A1),
  egysz(B,B1),
  C = A+B.
```

11.2. Válasz. ***Oktató:*** Gondolom a legutolsó sor `C=A1+B1` akar lenni. De még így sem jó (logikailag nem igaz): attól hogy `A`-ban `A1` egy legbővebb részkifejezés, `B`-ben `B1`, egyáltalán nem következik, hogy `A+B`-ben `A1+B1` legbővebb ilyen részkifejezés (általában nem is részkifejezés).

A kérdező folytatja:

```
egysz(K,Ek):-
  K=..F,
  egylist(F,Ek).

egylist([],[]).
  egylist([A|L],[B|LL]):-
  egysz(A,B),egylist(L,LL).
```

A `=.. F` eredménye `F = [a]`, ezért az `egylist` rögtön visszahívja `egysz(a, _)`-t.

Alább olvasható egy megoldás:

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

jo(Kif) :-
        nonvar(Kif), Kif=A+B,
        jo(A), jo(B).
jo(A):-
        number(A).


egysz(Kif, A) :-
        jo(Kif), !, A=Kif.
egysz(Kif, A) :-
        compound(Kif), Kif =.. [_|ArgL],
        member(Arg, ArgL),
        egysz(Arg, A).


| ?- egysz(1+a+(2+3+c), E).

E = 1 ? ;

E = 2+3 ? ;

no
```

11.3. Kérdés. Valaki megtudná nekem mondani, hogy a

```
0'x
(x=0..9)
```

mit jelent? Mert az értékeket már sikerült kitaláltatnom a SICStussal, de mégis kíváncsi lennék rá, hogy maga az `'` milyen operátor (ha az egyáltalán...)?

11.3. Válasz. ***Oktató:*** A SICStus (az ISO is, kis megszorításokkal) megengedi, hogy egészeket tetszőleges számrendszerben felírjunk. Így például a 23-at írhatjuk a következőképpen:

```
| ?- write(23).
23
yes

| ?- write(2'10111).
23
yes

| ?- write(16'17).
23
yes
```

ahol a `'` előtti tetszőleges (2 és 36 közötti) szám adja meg a számrendszert.

A 0-ás „számrendszer”-nek különleges jelentése van, a megadott karakterkonstans kódjat adja meg. Ezért van az, amit irtal, hogy `0'1` a 49-es kódot reprezentálja, `0'2` az 50-eset, `0'b` a 98-asat.

11.4. Kérdés. Tudja valaki, hogy mi lehet a baj? Nem sikerül Prologban típust létrehoznom. De ha azt írom be amit órán írtunk, arra se megy. pl.

```
:- type valami ---> valami(atom,atom,int).
```

erre is ugyanazt írja ki.

```
| ?- :- type ember == atom.
! Syntax error
! operator expected after expression
! in line 554
! :- type
! <<here>>
! ember == atom .
```

11.4. Válasz. ***Oktató:*** Biztos elkerülte a figyelmét, pedig többször is mondtam az órán, hogy a Prolog nem típusos nyelv, ezeket a típusdefiníciókat csak KOMMENTÁRként, az emberi olvasónak szánjuk.

Persze más LP nyelvekben, mint pl. a Mercury-ban gépi fogyasztásra szánt típusok is vannak.