Deklaratív Programozás

Hanák Péter
hanak@inf.bme.hu

Irányítástechnika és Informatika Tanszék

OM Kutatás-Fejlesztési Helyettes Államtitkárság

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

Számítástudományi és Információelméleti Tanszék

IQSOFT Intelligens Software Rt.

KÖVETELMÉNYEK — TUDNIVALÓK

Deklaratív programozás: tudnivalók

Honlap, levelezési lista

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

Jegyzet

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

Deklaratív programozás: tudnivalók (folyt.)

Fordító- és értelmezőprogramok

- SICStus Prolog (3.8.5, licenszköteles, aláírás ellenében jelszót adunk)
- Moscow SML (2.0, szabad szoftver)
- Mindkettő telepítve van a <kempelen.inf.bme.hu>-n
- Mindkettő letölthető a honlapról (linux, Win95/98/NT)
- Webes gyakorló felület készül (ld. honlap)
- Kézikönyvek HTML-változatban (MOSML pdf is)
- Más programok: swiProlog, gnuProlog smlnj
- emacs-szövegszerkesztő SML-, ill. Prolog-módban (linux, Win95/98/NT)

Deklaratív programozás: félévközi követelmények

Nagy házi feladat (NHF)

- Programozás mindkét nyelven (Prolog, SML)
- Mindenkinek önállóan kell kódolnia (programoznia)!
- Hatékony (időlimit!), jól dokumentált („kommentezett”) programok
- A két programhoz közös, 8-10 oldalas fejlesztői dokumentáció (TXT, TeX/LaTeX, HTML, PDF, PS; de nem DOC vagy RTF)
- Kiadás a 4.-5. héten, a honlapon, letölthető keretprogrammal
- Beadás a 13. héten (létraversenyhez), legkésőbb a vizsgaidőszak első hetében; elektronikus levélben (ld. honlap)
- A beadáskor és a pontozáskor külön-külön tesztsorozatot használunk (nehézségben hasonlókat, de nem azonosakat)
- A minden tesztesetet hibátlanul megoldó programok *létraversenyen* vesznek részt (hatékonyság, gyorsaság plusz pontokért)

Deklaratív programozás: félévközi követelmények (folyt.)

Nagy házi feladat (folyt.)

- Nem kötelező, de ***nagyon*** ajánlott!
- 40%-os szabály (nyelvenként a max. részpontszám 40%-a kell az eredményességhez)
- Súlya az osztályzatban: 15%

Deklaratív programozás: félévközi követelmények (folyt.)

Kis házi feladatok (KHF)

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

Gyakorló feladatok

- Kötelezők!
- Gyakorlás a honlapon keresztül
- A gyakorlatok megoldását nyilvántartjuk
- Pontot nem adunk
- Pótlási lehetőség a vizsgaidőszak első hetében

Deklaratív programozás: félévközi követelmények (folyt.)

Nagyzárthelyi, pótzárthelyi (NZH, PZH)

- NZH a 13. oktatási héten
- Kötelező!
- Semmilyen jegyzet, segédlet nem használható
- A megtanulandó könyvtári függvények, ill. eljárások listáját előre megadjuk
- 40%-os szabály (nyelvenként a max. részpontszám 40%-a kell az eredményességhez)
- PZH a vizsgaidőszak első hetében
- Súlya az osztályzatban: 15%

Deklaratív programozás: vizsga

Vizsga

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

DEKLARATÍV ÉS IMPERATÍV PROGRAMOZÁS

Programozási nyelvek osztályozása

Programozási nyelvek — stílusok

- **Imperatív**
  - *Fortran*
  - *Algol*
  - *C*
  - *C++*
  - …
- **Deklaratív**
  - **Funkcionális**
    - *LISP*
    - *ML*
    - …
  - **Logikai**
    - *SQL*
    - *Prolog*
    - *CLP nyelvek*
    - …

Imperatív és deklaratív programozási nyelvek

Imperatív program

- felszólító módú, utasításokból áll
- változó: változtatható értékű memóriahely
- Példa: `int fakt(int n) {int f=1; while (n>1) f*=n-; return f;}`

Deklaratív program

- kijelentő módú, egyenletekből, állításokból áll
- változó: egy ismeretlen, de (előbb–utóbb) rögzített értékű mennyiség
- SML példa: `fun fakt 0 = 1 | fakt n = n * fakt (n-1);`
- C példa: `int fakt(int n) {if (n<=1) return 1; else return n*fakt(n-1);}`

Deklaratív nyelvek jelszavai

- MIT és nem HOGYAN (WHAT rather than HOW): a *megoldás módja* helyett inkább a megoldandó *feladat leírását* kell megadni
- Egyszeres értékadás (single assignment) — párhuzamos végrehajthatóság

Példa — családi kapcsolatok

Adatok

Egy gyerek–szülő kapcsolat, pl.

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

A feladat:

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

A nagyszülő feladat — C nyelvű megoldás

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

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

A nagyszülő feladat — SML megoldás

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

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

A függvény futtatása

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

A nagyszülő feladat — SQL megoldás

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

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

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

SQL>
```

A nagyszülő feladat — Prolog megoldás

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

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

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

A deklaratív és imperatív megoldások összehasonlítása

A keresési feladat megoldása

- C nyelven: ciklussal
- SQL-ben: beépített adatbázis-kereséssel
- SML-ben: magasabbrendű függvénybe rejtett rekurzióval
- Prologban: beépített mintaillesztéses eljáráshívással

Az összetett feltételek kezelése

- C nyelven: skatulyázott ciklussal
- SML-ben: leképezések komponálásával
- SQL-ben, Prologban: relációk konjunkciójának képzésével

A funkcionális és logikai megoldásokról

- az SML megoldás rendkívül tömör (magasabbrendű függvények)
- a Prolog megoldás többirányú (több függvénykapcsolatnak felel meg)

Egy „számolós” program: faktoriális

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

(* SML megoldás: fakt n = n! *)
fun fakt 0 = 1
  | fakt n = n * fakt (n-1)
```

Tanulságok, érdekességek

- SML skatulyázott függvények ⇔ Prolog egymás mellé rendelt relációk
- Negatív argumentumra a Prolog kód meghiúsul, az SML kód végtelen ciklusba esik

Egy összetettebb példa: N-edik generációs ősök

SML megoldás

```
(* szulei_l lista = a lista-ban szereplő emberek szüleinek listája*)
fun szulei_l ls = List.concat (map szulei ls)
 > val szulei_l = fn : string list -> string list

fun nagyszulei1 gy = szulei_l (szulei gy)
fun nagyszulei2 gy = szulei_l (szulei_l [gy])

fun osei (0, gy) = [gy]
  | osei (n, gy) = szulei_l (osei (n-1, gy))
 > val osei = fn : int * string -> string list

 - osei (2, "Imre");
 > val it = ["Géza", "Sarolt", "Civakodó Henrik", "Burgundi Gizella"]
          : string list
```

N-edik generációs ősök — Prolog megoldás

```
% ose(N, E0, E): E0-nak N-edik generációs őse az E.
% **** N adott szám. ****
ose(0,  E, E).
ose(N, E0, E) :-
        N > 0, N1 is N-1,
        szuloje(E0, Sz),
        ose(N1, Sz, E).
```

Futása

```
| ?- ose(2, 'Imre', Os).
Os = 'Géza' ? ;
Os = 'Sarolt' ? ;
Os = 'Civakodó Henrik' ? ;
Os = 'Burgundi Gizella' ? ;
no
```

```
| ?- ose(2, Utod, 'Burgundi Gizella').
Utod = 'Imre' ? ;
no
| ?- ose(N, 'Imre', Os).
N = 0, Os = 'Imre' ? ;
{INSTANTIATION ERROR: _157>0 - arg 1}
```

N-edik generációs ősök — általánosabb Prolog megoldás

```
% ose1(N, E0, E): E0-nak N-edik generációs őse az E.
ose1(0,  E, E).
ose1(N, E0, E) :-
        szuloje(E0, Sz), ose1(N1, Sz, E), N is N1+1.
```

Futása

```
| ?- ose1(N, 'Imre', Os), N < 2.
N = 0, Os = 'Imre' ? ;
N = 1, Os = 'István' ? ;
N = 1, Os = 'Gizella' ? ;
no
| ?- ose1(I, Utod, 'Burgundi Gizella').
I = 0, Utod = 'Burgundi Gizella' ? ;
I = 2, Utod = 'Imre' ? ;
I = 1, Utod = 'Gizella' ? ;
no
```

A funkcionális programozásról dióhéjban

Alapeszme

- a program elemei értékek, speciálisan függvények
  - egy függvény egy kiszámítási szabályt ad meg
  - a program futása: kiértékelés (egyszerűsítés, redukció)

A funkcionális programozás első megvalósítása: LISP

- alapötlet: listák könnyű/hatékony feldolgozása

A funkcionális programozás egy modern megvalósítása: SML

- a függvények „teljes jogú” értékek
- erős típusfogalom, típusok automatikus levezetése

SML — előnyök és hátrányok

Miért jó?

- nagyon tömör kód
- függvények is értékek: futási időben létrehozhatók
- mintaillesztés: adatstruktúrák könnyen, áttekinthetően kezelhetők
- erős típusrendszer

Mik a hátrányai?

- megszokottól eltérő programozói stílus

Hogyan tovább?

- lusta kiértékelés (Haskell, Clean)
- párhuzamos végrehajtás (Parallel Haskell, CAML — Concurrent ML)
- típusrendszer bővítése öröklődéssel (Haskell, Clean, Objective CAML)

A logikai programozásról dióhéjban

Alapeszme

- A program elemei logikai állításoknak felelnek meg, pl.:
  `nagyszuloje(U, N) :- szuloje(U, Sz), szuloje(Sz, N).`
  matematikai formája:
  $\forall U \forall N \forall Sz(\texttt{nagyszuloje}(U,N) \leftarrow \texttt{szuloje}(U,Sz) \wedge \texttt{szuloje}(Sz,N))$
- A program futása: dedukció (tételbizonyítási folyamat)

A logikai programozás első megvalósítása: a Prolog nyelv

- A logikai állítások egyszerűek, tekinthetők eljárásdefiníciónak is
- A tételbizonyítási folyamat értelmezhető mint:
  mintaillesztéses eljáráshívás + visszalépéses keresés
- Prolog = RDBMS + rekurzió + adatstruktúrák

Prolog — előnyök és hátrányok

Miért jó?

- tömör kód, többirányú eljárások
- „automatikus” visszalépéses keresés, ciklusok kiváltása
- „logikai” változó — meghatározatlan adatok kezelése

Mik a hátrányai?

- nehéz megtanulni (különösen „tapasztalt” programozóknak)
- rögzített, rugalmatlan vezérlési mechanizmus
- gyenge következtetési képesség

Hogyan tovább?

- CLP — korlát logikai programozás (constraint logic programming)
- annotációk, típusok — Mercury
- rugalmasabb vezérlés, párhuzamos végrehajtás — Aurora, Andorra, Oz

Deklaratív programozás — miért tanítjuk?

Új, magasszintű programozási elemek

- rekurzió
- mintaillesztés
- visszalépéses keresés

Új gondolkodási stílus

- a programrészek (relációk, függvények) önálló jelentéssel bírnak
- a kód és a jelentés összevethető: program-verifikáció

Új alkalmazási területek

- szimbolikus alkalmazások
- következtetési módszerekre épülő megoldások
- nagyfokú megbízhatóságot igénylő rendszerek

Egy példa: párbeszéd egy 50 soros Prolog programmal

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

BEVEZETÉS A LOGIKAI PROGRAMOZÁSBA

Bevezetés a Logikai Programozásba

Az előadássorozat áttekintése

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

A Prolog/LP rövid történeti áttekintése

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

Információk a logikai programozásról

Prolog megvalósítások:

- **SWI Prolog:** `http://www.swi.psy.uva.nl/projects/SWI-Prolog/`
- **SICStus Prolog:** `http://www.sics.se/ps/sicstus/sicstus_toc.html`
- **GNU Prolog:** `http://pauillac.inria.fr/~diaz/gnu-prolog/`

Hálózati információforrások:

The WWW Virtual Library: Logic Programming:

`http://www.comlab.ox.ac.uk/archive/logic-prog.html`

CMU Prolog Repository:

`http://www.cs.cmu.edu/afs/cs.cmu.edu/project/ai-repository/ai/lang/prolog/0.html`

Prolog FAQ:

`http://www.cs.cmu.edu/afs/cs/project/ai-repository/ai/lang/prolog/faq/prolog.faq`

Prolog Resource Guide:

`http://www.cs.cmu.edu/afs/cs/project/ai-repositoryai/lang/prolog/faq/prg_[12].faq`

Magyar nyelvű Prolog irodalom

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

Márkusz Zsuzsa: Prologban programozni könnyű.
Novotrade, 1988

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

A PROLOG NYELV KÖZELÍTŐ SZINTAXISA

Predikátumok, klózok

```
% két klózból álló predikátum definíciója, funktora: ose/3
ose(0,  E, E).                   %                      1. klóz, tényállítás
ose(N, E0, E) :-                 %             fej   |
        szuloje(E0, Sz),         % cél        | törzs | 2. klóz, szabály
        ose(N1, Sz, E), N is N1+1.    % cél, cél   |       |
```

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

- Alternatív szóhasználat:
  - predikátum — eljárás
  - cél — hívás

Prolog kifejezések

```
%          ose(0, E, E)                  % összetett kifejezés, funktora ose/3
%           |  |  |  |
% struktúranév |  argumentum, változó
%              \- argumentum, számkonstans
```

⟨ kifejezés ⟩	::=	⟨ változó ⟩ |	{`var`}
		⟨ konstans ⟩ |	{`atomic`}
		⟨ összetett kifejezés ⟩	{`compound`}
⟨ konstans ⟩	::=	⟨ névkonstans ⟩ |	{`atom`}
		⟨ számkonstans ⟩	{`number`}
⟨ összetett kifejezés ⟩	::=	⟨ struktúranév ⟩ (⟨ argumentum ⟩, . . .)	
⟨ struktúranév ⟩	::=	⟨ névkonstans ⟩	
⟨ argumentum ⟩	::=	⟨ kifejezés ⟩	

- összetett kifejezés funktora = struktúranév/argumentumszám, pl. `ose/3`
- konstans funktora = konstans/0, pl. `'István'/0`
- változónak nincs funktora

Lexikai elemek

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

⟨ **változó** ⟩ ⟨ **nagybetű** ⟩⟨ **alfanum** ⟩. . . |
_ ⟨ **alfanum** ⟩. . . |

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

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

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

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

⟨ **számkonstans** ⟩ ::= **{előjeles vagy előjeltelen számjegysorozat esetleges tizedes résszel és exponenssel}**

Szintaktikus édesítőszer: operátorok

```
% N is N1+1 ekvivalens az is(N, +(N1,1)) kifejezéssel
```

Operátor-deklaráció

- `:- op(`⟨ prioritás ⟩, ⟨ fajta ⟩, ⟨ operátornév ⟩`).`
- ⟨ operátornév ⟩ tetszőleges névkonstans
- ⟨ prioritás ⟩ 0–1200 közötti egész
- ⟨ fajta ⟩
  - infix: `yfx`, `xfy`, `xfx`; `A op B` $\equiv$ `op(A, B)`
  - prefix: `fx`, `fy`; `op A` $\equiv$ `op(A)`
  - postfix: `xf`, `yf`; `A op` $\equiv$ `op(A)`
- a ⟨ fajta ⟩-ban `x` és `y` az asszociativitást határozzák meg:
  - `x`: az adott oldalon nem állhat azonos prioritású operátor zárójelezetlenül
  - `y`: az adott oldalon állhat azonos prioritású operátor zárójelezetlenül

Beépített operátorok

Szabványos operátorok

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

Egyéb beépített operátorok

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

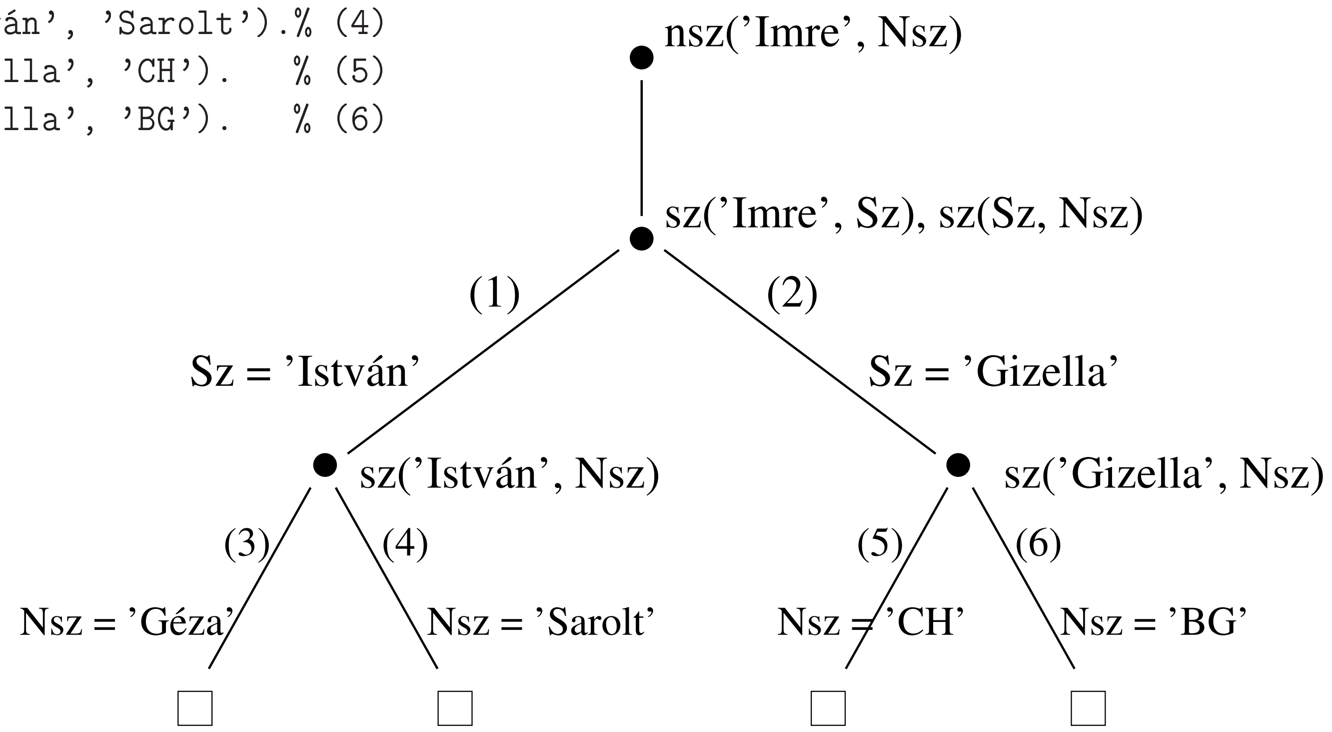
[1] `sicstus` módban 300 `xfx` operátor
[2] `sicstus` módban 500 `fx` operátor
[3] `iso` módban 200 `fy` operátor

A PROLOG VÉGREHAJTÁSI MECHANIZMUSA

Prolog végrehajtási példa

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

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



A végrehajtás alapelemei: egyesítés

Az eljáráshívás és egy klózfej azonos alakra hozása, változók behelyettesítésével

Példák

- **Bemenő paraméterátadás:**
  hívás: `nsz('Imre', Nsz),`
  fej: `nsz(Gy, N),`
  behelyettesítés: `Gy = 'Imre', N = Nsz`
- **Kimenő paraméterátadás:**
  hívás: `sz('Imre', Sz),`
  fej: `sz('Imre', 'István'),`
  behelyettesítés: `Sz = 'István'`
- **Bemenő/kimenő paraméterátadás:**
  hívás: `ose(N, 'Imre', Os)`
  fej: `ose(0, E, E)`
  behelyettesítés: `N = 0, E = 'Imre', Os = 'Imre'`

A végrehajtás alapelemei: redukciós lépés

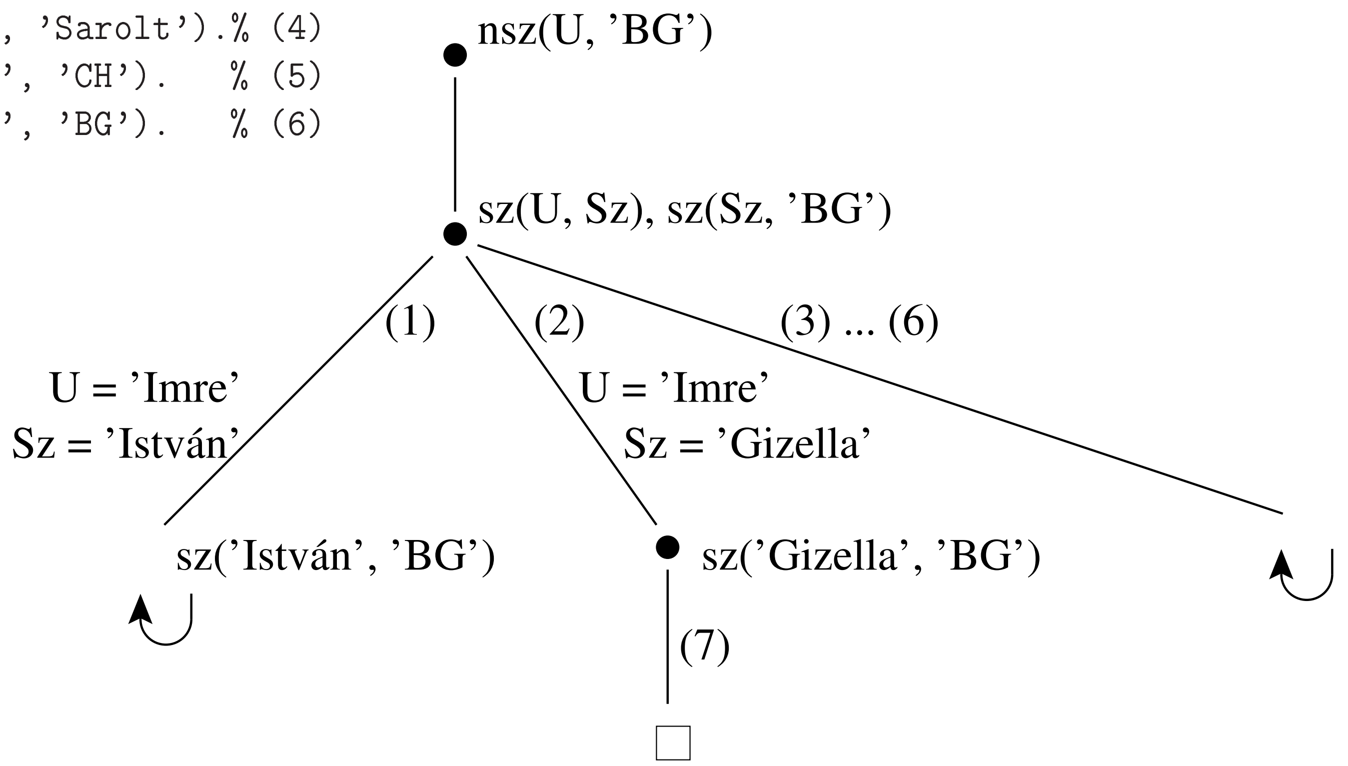
Redukciós lépés

- Egy célsorozat (hívássorozat) redukálása egy újabb célsorozattá egy klóz segítségével
- A redukciós lépés végrehajtása:
  - A klózt lemásoljuk, minden változót szisztematikusan új változóra cserélve.
  - A célsorozatot szétbontjuk az első hívásra és a maradékra.
  - Az első hívást egyesítjük a klózfejjel
  - Az egyesítéshez szükséges behelyettesítéseket elvégezzük a klóz törzsén és a célsorozatot maradékán
  - Az új célsorozat: a klóztörzs és utána a maradék célsorozat

Újabb végrehajtási példa

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

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



A Prolog végrehajtási algoritmusa

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

Előzetes — aritmetikai beépített eljárások

- `X is Kif`: A `Kif` aritmetikai kifejezés értékét egyesíti `X`-szel
- `Kif1 < Kif2`, `Kif1 =< Kif2`, `Kif1 > Kif2`, `Kif1 >= Kif2`, `Kif1 =:= Kif2`, `Kif1 =\= Kif2`: A `Kif1` és `Kif2` aritmetikai kifejezések értéke a megadott relációban van egymással (`=:=` ⇒egyenlő, `=\=` ⇒nem-egyenlő).
- Ha `Kif`, `Kif1`, `Kif2` valamelyike nem aritmetikai kifejezés ⇒hiba.
- Legfontosabb aritmetikai operátorok: `+`, `-`, `*`, `/`, `mod`, `//` (egész-osztás)

```
| ?- X is 1*2+3.
X = 5 ?
| ?- X is alma.
{DOMAIN ERROR: _78 is alma - arg 2: expected expression, found alma}
| ?- X =:= 1*2+3.
{INSTANTIATION ERROR: _84=:=1*2+3 - arg 1}
| ?- 1+2*3 > 2*3+1.
no
| ?-
```

Előzetes — programfejlesztési beépített eljárások

- `consult(File)` vagy `[File]`: A `File` állományban levő programot beolvassa és értelmezendő alakban eltárolja. (`File = user` ⇒ terminálról olvas.)
- `listing` vagy `listing(Predikátum)`: Az értelmezendő alakban eltárolt összes ill. adott nevű predikátumokat kilistázza.
- `compile(File)`: A `File` állományban levő programot beolvassa, lefordítja.
- A beolvasott program elemei Prolog kifejezések (`':-'`, `','` operátorok!)
- `halt`: A Prolog rendszer befejezi működését.

```
> sicstus
SICStus 3.8.5 (x86-linux-glibc2.1): Fri Oct 27 10:16:41 CEST 2000
| ?- consult(fakt).
{consulted /home/user/fakt.pl in module user, 10 msec 384 bytes}
| ?- listing(fakt).
fakt(0, 1).
fakt(A, B) :-
        A>0, C is A-1, fakt(C, D), B is D*A.
| ?- halt.
>
```

Előzetes — kiíró és egyéb eljárások

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

```
| ?- write(+(1,*(2,3))), write('          '), display(1+2*3), nl.
1+2*3          +(1,*(2,3))
yes
| ?- szuloje('István', X), write(X), nl, fail.
Géza
Sarolt
no
```

Prolog végrehajtási példa — rekurzió verem nélkül

```
/* (1) */ fakt(0, 1).
/* (2) */ fakt(N, F) :-
                  N>0, NN is N-1, fakt(NN, FF), F is FF*N.
```

- **Kezdeti célsorozat:** `fakt(2,X), write(X)`
- **Redukció** `(2)`**-vel:** `2>0, NN is 2-1, fakt(NN, FF), X is FF*2, write(X)`
- **Beépítettek végrehajtása:** `fakt(1, FF), X is FF*2, write(X)`
- **Redukció** `(2)`**, beép.:** `fakt(0, FF1), FF is FF1*1, X is FF*2, write(X) (*)`
- **Redukció** `(1)`**-gyel (választási pont!):** `FF is 1*1, X is FF*2, write(X)`
- **Beép.:** `write(2)`**, mellékhatás:** ⇒2 **kiírása, sikeres vég**
- **További megoldás kérése esetén visszalépés (*)-hoz, redukció** `(2)`**-vel:** `0>0, NN2 is 0-1, fakt(NN2, FF2), ...`**, beép. végrehajt., meghiúsulás.**

Prolog végrehajtási példa — nyomkövetés

```
| ?- trace, fakt(2, X), write(X), nl, fail.
        1      1 Call: fakt(2,X) ?
        2      2 Call: 2>0 ?
        2      2 Exit: 2>0 ?
        3      2 Call: NN is 2-1 ?
        3      2 Exit: 1 is 2-1 ?
        4      2 Call: fakt(1,FF) ?
        5      3 Call: 1>0 ?
        5      3 Exit: 1>0 ?
        6      3 Call: NN1 is 1-1 ?
        6      3 Exit: 0 is 1-1 ?
        7      3 Call: fakt(0,FF1) ?
?       7      3 Exit: fakt(0,1) ?
        8      3 Call: FF is 1*1 ?
        8      3 Exit: 1 is 1*1 ?
?       4      2 Exit: fakt(1,1) ?
        9      2 Call: X is 1*2 ?
        9      2 Exit: 2 is 1*2 ?
?       1      1 Exit: fakt(2,2) ?
2
```

```
        1      1 Redo: fakt(2,2) ?
        4      2 Redo: fakt(1,1) ?
        7      3 Redo: fakt(0,1) ?
       10      4 Call: 0>0 ?
       10      4 Fail: 0>0 ?
        7      3 Fail: fakt(0,FF1) ?
        4      2 Fail: fakt(1,FF) ?
        1      1 Fail: fakt(2,X) ?

no
{trace}
| ?-
```

Prolog végrehajtás — egy aritmetikai példa

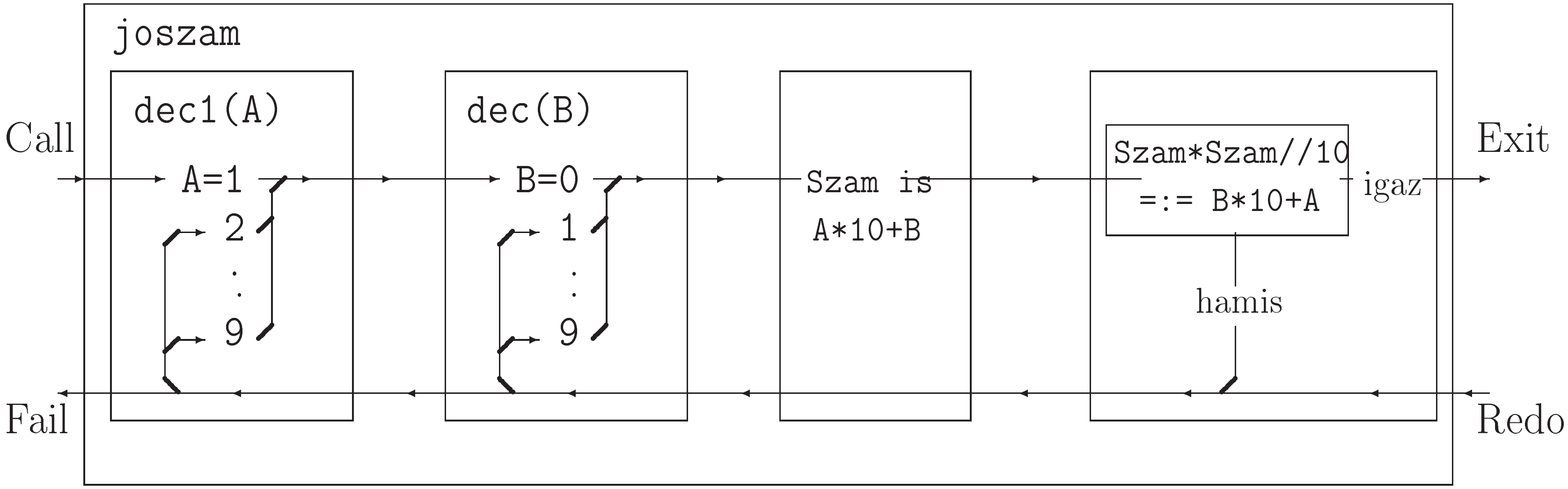
Példa: „jó” számok

Keressük azokat a kétjegyű számokat amelyek négyzete háromjegyű és a szám fordítottjával kezdődik:

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

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

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

Prolog végrehajtás — a 4-kapus doboz modell

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



Prolog végrehajtás — számintervallum felsorolása

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

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

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

Prolog végrehajtás — diszjunkció mint szemantikus édesítőszer

Példa:

```
% Sz-nek gyermeke Gy.
gyermeke(Sz, Gy) :- fia(Sz, Gy).
gyermeke(Sz, Gy) :- lanya(Sz, Gy).
```

Azonos fejű szabályok összevonhatók egy diszjunkció bevezetésével:

```
gyermeke(Sz, Gy) :-
          (   fia(Sz, Gy)
 /*vagy:*/;   lanya(Sz, Gy)
          ).
```

A fejek azonossá is tehetők segédvált. és az `X=Y` beép. eljárás használatával:

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

Az `X = Y` beépített eljárás definiciója:

```
% X = Y: X és Y egyesíthető
X = X.
```

Prolog végrehajtás — diszjunkció kiváltása

```
% Utodnak valódi őse Os.
valodi_ose(Utod, Os) :-
        szuloje(Utod, Szulo),
        (   Os = Szulo
        ;   valodi_ose(Szulo, Os)
        ).
```

- Meghatározzuk a diszjunkcióban szereplő változókat: `Os`, `Szulo`.
- Meghatározzuk a diszjunkción kívül szereplő változókat: `Utod`, `Os`, `Szulo`.
- A segédeljárás argumentumai a közös változók: `seged(Szulo, Os)`.
- Minden alternatívájából egy külön klóz lesz a segédeljárásban.
- A diszjunkciót a segédeljárás hívásával helyettesítjük.

```
valodi_ose(Utod, Os) :-
        szuloje(Utod, Szulo), seged(Szulo, Os).

seged(Szulo, Os) :- Os = Szulo.            % egyszerűsítető: seged(Szulo, Szulo).
seged(Szulo, Os) :- valodi_ose(Szulo, Os).
```

A PROLOG ADATFOGALMA

Összetett adatstruktúrák — példa

- egy ember egy két mezőből álló rekord: vezetéknév, keresztnév
- egy kétargumentumú összetett kifejezéssel ábrázoljuk
- a struktúranév legyen pl. a minusz jel (-): `-(VNév, KNév)` ≡ `VNév-KNév`

```
% szuloje(Gy, Sz): Gy szülője Sz.
szuloje(szabo-laszlo, szabo-gyorgy).
szuloje(szabo-laszlo, laszlo-amalia).
szuloje(szabo-amalia, szabo-gyorgy).
szuloje(szabo-amalia, laszlo-amalia).
```

```
% keresztneve(E, KN): E keresztneve KN.
keresztneve(_Nev-Kereszt, Kereszt).

% vezetekneve(E, VN): E vezetékneve VN.
vezetekneve(Nev-_, Nev).
```

```
| ?- szuloje(E1, E2), keresztneve(E1, K), vezetekneve(E2, K).
        K = laszlo, E1 = szabo-laszlo, E2 = laszlo-amalia ? ;
        no
| ?- szuloje(V1-K1, K1-K2).
        K1 = laszlo, K2 = amalia, V1 = szabo ?
        yes
| ?- szuloje(E1, E2), E1 = _-Nev, E2 = Nev-_.
        E1 = szabo-laszlo, E2 = laszlo-amalia, Nev = laszlo ?
```

Variációk egy témára — a logikai változó fogalma

```
% E keresztneve megegyezik egyik szülőjének vezetéknevével
erdekes(E) :-
        szuloje(E, Sz),
        keresztneve(E, KN),
        vezetekneve(Sz, KN).

erdekes2(V-K) :-
        szuloje(V-K, K-_).

erdekes3(E) :-
        keresztneve(E, KN),
        vezetekneve(Sz, KN),
        szuloje(E, Sz).

erdekes4(E) :-
        E = _-K,
        szuloje(E, K-_).

| ?- spy szuloje, erdekes3(E).
 +      1      1 Call: szuloje(_965-_951,_951-_978) ?
```

Többszörösen összetett adatok — többtagú nevek

- **többtagú kereszt- és vezetéknevek kezelése, pl.** `Kovacs Eva Maria`
- **a tagokat egy másik struktúranévvel kapcsoljuk össze, pl.** `szabo-eva/maria`
- **ugyanazt a - struktúranevet használjuk:** `kis-szabo-laszlo`, `szabo-(eva-maria)`

```
% reszneve(Osszetett, Resz): Osszetett név része a Resz név.
reszneve(Nev, Nev).
reszneve(Elso-_, Nev) :-
        reszneve(Elso, Nev).
reszneve(_-Masodik, Nev) :-
        reszneve(Masodik, Nev).
```

```
| ?- reszneve(kis-kovacs-bela, Resz).
        Resz = kis-kovacs-bela ? ;
        Resz = kis-kovacs ? ;
        Resz = kis ? ;
        Resz = kovacs ? ;
        Resz = bela ? ;
        no
```

```
| ?- reszneve(szabo-(eva-maria), Resz).
        Resz = szabo-(eva-maria) ? ;
        Resz = szabo ? ;
        Resz = eva-maria ? ;
        Resz = eva ? ;
        Resz = maria ? ;
        no
```

Típusok Prologban

A Prolog nem típusos nyelv, de érdemes meghatározni a kezelt adatok típusát, például az alábbi formális típus-leírással.

```
% :- type név1 == {vnév - knév}.    % egy név1 típusú kifejezés az egy - struktúra
                                    % vnév és knév típusú argumentumokkal.
                                    % név1 = { v-k | v ∈  vnév, k ∈ knév }
% :- type vnév == atom.             % egy vnév típusú kifejezés az egy atom
% :- type knév == atom.             % egy knév is egy atom
% :- pred szuloje(név1, név1).  % szuloje mindkét argumentuma név1 típusú.
szuloje(szabo-laszlo, szabo-gyorgy). %...
```

Rekurzívan definiált típusok

```
% :- type név2 == atom \/           % név2 az atom vagy egy
%                   {név2 - név2}.  % két név2-ből álló - /2 struktúra
% :- pred reszneve(név2, név2).     % reszneve argumentumai név2 típusúak.
% reszneve(Osszetett, Resz): Osszetett név része a Resz név.
reszneve(Nev, Nev).
reszneve(Elso-_, Nev) :-    reszneve(Elso, Nev).
reszneve(_-Masodik, Nev) :- reszneve(Masodik, Nev).
```

A Prolog adatfogalma — az egyesítési algoritmus

A Prolog adatfogalma: a Prolog kifejezés

- konstans (szám- ill. névkonstans)
- struktúra-kifejezés
- változó („teljes jogú", struktúra-kifejezések mélyén is lehet)

Adatstruktúrák szétszedése, összerakása: egyesítési algoritmus

- bemenete: két Prolog kifejezés (belső, fastruktúra formában!);
- célja: azon *legáltalánosabb* változó-behelyettesítések meghatározása, amelyekkel a két kifejezés azonos alakra hozható;
- eredménye:
  - siker, változó-behelyettesítések; vagy
  - meghiúsulás (a kifejezések nem hozhatók azonos alakra).

Egyesítés: a behelyettesítés fogalma

A behelyettesítés

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

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

- σ általánosabb mint θ, ha létezik olyan ρ, hogy $\theta = \sigma \otimes \rho$

Legáltalánosabb egyesítő (*mgu* — most general unifier)

- A és B kifejezések egyesíthetőek ha létezik egy olyan σ behelyettesítés, hogy $A\sigma = B\sigma$. Ezt a σ behelyettesítést A és B egyesítőjének nevezzük.
- A és B legáltalánosabb egyesítője σ ($mgu(A, B) = \sigma$), ha σ A és B minden egyesítőjénél általánosabb (Tétel: átnevezéstől eltekintve egyértelmű.)

Az egyesítési algoritmus

Az egyesíthetőség eldöntése, $\sigma = mgu(A, B)$ előállítása

1. Ha A és B azonos változók vagy konstansok, akkor $\sigma = \emptyset$.
2. Egyébként, ha A változó, akkor $\sigma = \{A \leftarrow B\}$.
3. Egyébként, ha B változó, akkor $\sigma = \{B \leftarrow A\}$.
4. Egyébként, ha A és B azonos nevű és argumentumszámú összetett kifejezések és argumentum-listáik A_1,...,A_N ill. B_1,...,B_N, és

   a. A_1 és B_1 legáltalánosabb egyesítője σ_1,
   b. $A_2\sigma_1$ és $B_2\sigma_1$ legáltalánosabb egyesítője σ_2,
   c. $A_3\sigma_1\sigma_2$ és $B_3\sigma_1\sigma_2$ legáltalánosabb egyesítője σ_3,
   d. ...

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

Egyesítési példák

A = `ose(0, E, E)`, B = `ose(N, 'Géza', Os)`

- (4.) A és B neve és argumentumszáma megegyezik
  - (a.) `mgu(0, N)` (3. szerint) $= \{$`N`$\leftarrow$`0`$\} = \sigma_1$
  - (b.) `mgu(E`σ_1`, 'Géza')` = `mgu(E, 'Géza')` (2. szerint) $= \{$`E`$\leftarrow$`'Géza'`$\} = \sigma_2$
  - (c.) `mgu(E`$\sigma_1\sigma_2$`, Os)` = `mgu('Géza', Os)` (3. szerint) $= \{$`Os`$\leftarrow$`'Géza'`$\} = \sigma_3$
- tehát $mgu(A, B) = \sigma_1 \otimes \sigma_2 \otimes \sigma_3 = \{$`N`$\leftarrow$`0`, `E`$\leftarrow$`'Géza'`, `Os`$\leftarrow$`'Géza'`$\}$

A = `keresztneve(V-K, K)`, B = `keresztneve(E, jozsi)`

- (4.) A és B neve és argumentumszáma megegyezik
  - (a.) `mgu(V-K, E)` (3. szerint) $= \{$`E`$\leftarrow$`V-K`$\} = \sigma_1$
  - (b.) `mgu(K`σ_1`, jozsi)` = `mgu(K, jozsi)` (2. szerint) $= \{$`K`$\leftarrow$`jozsi`$\} = \sigma_2$
- tehát $mgu(A, B) = \sigma_1 \otimes \sigma_2 = \{$`E`$\leftarrow$`V-jozsi`, `K`$\leftarrow$`jozsi`$\}$

Egyesítési példák a gyakorlatban

```
| ?- kis-kovacs-bela = X-Y.
        X = kis-kovacs, Y = bela ? ;
        no
| ?- kis-kovacs-bela = kis-X.
        no
| ?- f(X, 3/Y-X, Y) = f(U, B-a, 3).
        B = 3/3, U = a, X = a, Y = 3 ?

| ?- f(f(X), U+2*2) = f(U, f(3)+Z).
        U = f(3), X = 3, Z = 2*2 ?

| ?- ose(0, V-jozsi, szabo-K). % = ose(0, E, E).
        K = jozsi, V = szabo ?

| ?- keresztneve(szabo-(eva-maria), N-maria). % = keresztneve(_-K, K).
        N = eva ?
```

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

Kérdés: `X` és `s(X)` egyesíthető-e?

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

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

Változót tartalmazó kifejezések — végtelen választás veszélye

```
| ?- keresztneve(E, K).
        E = _A-K ? ; no
| ?- keresztneve(E, K), szuloje(E, K-K2).
        E = szabo-laszlo, K = laszlo, K2 = amalia ? ; no
| ?- szuloje(E, K-K2), keresztneve(E, K).
        E = szabo-laszlo, K = laszlo, K2 = amalia ? ; no

| ?- szuloje(V-K, E), reszneve(E, K).
        V = szabo, K = laszlo, E = laszlo-amalia ? ;
        V = szabo, K = amalia, E = laszlo-amalia ? ; no
| ?- reszneve(E, K), szuloje(V-K, E).
        V = szabo, K = laszlo, E = laszlo-amalia ? ;
^C
Prolog interruption (h for help)? a
{Execution aborted}
| ?- reszneve(E, K).
        K = E ? ;
        E = K-_A ? ;
        E = K-_A-_B ?
```

PÉLDA — ÚTVONALKERESÉS

Az útvonalkeresési feladat

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

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

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

Az útvonalkeresési feladat — folytatás

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

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

Körmentes út keresése

A körök kizárására gyűjtenünk kell a már érintett városokat. A gyűjtő adastruktúra legyen pl. `Honnan-Közben1-Közben2- ....`

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

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

A meghiúsulásos negáció (NF — Negation by Failure)

A `\+ Hívás` beépített meta-eljárás (vö. $\nvdash$ — nem bizonyítható)

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

```
| ?- \+ szuloje(szabo-laszlo, X).         ----> no
| ?- \+ szuloje(szabo-gyorgy, X).         ----> true ?
| ?- /* T1 testvére T2:*/ szuloje(T1, _A), szuloje(T2, _A), \+ T1 = T2.
        T1 = szabo-laszlo, T2 = szabo-amalia ?
| ?- \+ X = 1, X = 2.                     ----> no
| ?- X = 2, \+ X = 1.                     ----> X = 2 ?
```

A példa nyomkövetése

```
| ?- spy reszneve, útvonal_2(2, 'Párizs', Hová, H).
{The debugger will first zip -- showing spypoints (zip)}
{Plain spypoint for user:reszneve/2 added, BID=1}
 +      1      1 Call: reszneve('Párizs','Bécs') ? l
 +      1      1 Fail: reszneve('Párizs','Bécs') ? l
 +      7      2 Call: reszneve('Párizs'-'Bécs','Berlin') ? l
 +      8      3 Call: reszneve('Párizs','Berlin') ? l
 +      8      3 Fail: reszneve('Párizs','Berlin') ? l
 +      9      3 Call: reszneve('Bécs','Berlin') ? l
 +      9      3 Fail: reszneve('Bécs','Berlin') ? l
 +      7      2 Fail: reszneve('Párizs'-'Bécs','Berlin') ? l
    H = 1900, Hová = 'Berlin' ? ;
 +     13      2 Call: reszneve('Párizs'-'Bécs','Párizs') ? l
 +     14      3 Call: reszneve('Párizs','Párizs') ? l
 +     14      3 Exit: reszneve('Párizs','Párizs') ? l
?+     13      2 Exit: reszneve('Párizs'-'Bécs','Párizs') ? l
 +     16      2 Call: reszneve('Párizs'-'Bécs','Budapest') ? n
    H = 1510, Hová = 'Budapest' ?
```

Körmentes út keresése — probléma a gyűjtő adatstruktúrával

```
% reszneve(Osszetett, Resz): Osszetett név része a Resz név.
reszneve(Nev, Nev).
reszneve(Elso-_, Nev) :-      reszneve(Elso, Nev).
reszneve(_-Masodik, Nev) :-   reszneve(Masodik, Nev).

járat(kál, kál-kápolna, 20).
járat(hatvan, kál, 30).

| ?- útvonal_2(2, hatvan, kál-kápolna, H).
        H = 50 ? ; no
| ?- útvonal_2(2, kál-kápolna, hatvan, H).
        no
| ?- spy reszneve, útvonal_2(2, kál-kápolna, hatvan, H).
 +      1      1 Call: reszneve(kál-kápolna,kál) ?
 +      2      2 Call: reszneve(kál,kál) ?
 +      2      2 Exit: reszneve(kál,kál) ?
?+      1      1 Exit: reszneve(kál-kápolna,kál) ?
no
| ?-
```

Probléma a gyűjtővel — általános gyűjtő-fogalom

```
% bfa_resze(Bfa, Resz): Bfa része  Resz.
bfa_resze(Bfa, Bfa).
bfa_resze(Balfa-_, Resz) :-      bfa_resze(Balfa, Resz).
bfa_resze(_-Jobbfa, Resz) :-     bfa_resze(Jobbfa, Resz).
```

Milyen típusú adatokat kezel a fenti `bfa_resze`?

- T típusú elemekből épített bináris fa (`bfa(`T`)`) az
- vagy két ugyanilyen bináris fa - struktúranévvel összekapcsolva (`bfa(`T`)-bfa(`T`)`);
- vagy pedig egy T típusú elem;
- formálisabban: `% :- type bfa(T) == {bfa(T)-bfa(T)} \/ T.`
- ez egy *nem megkülönböztetett* únió, mi van ha T funktora `-` `/2`?

Probléma a gyűjtő adatstruktúrával — megkülönböztetett úniók

Bináris fa megkülönböztetett únióval

```
% :- type bfa(T) == {bfa(T) - bfa(T)} \/ {level(T)}.
% egy szintaktikus egyszerűsítést bevezetve:
% :- type bfa(T) ---> bfa(T) - bfa(T) ; level(T).
% Egy T-kből álló bfa az vagy két ilyenből álló - /2, struktúra
% vagy egy level/1 struktúrába csomagolt T típusú adat.
```

„Unáris" vagyis lineáris fa (nekünk ez is elég)

```
% :- type gyujtemeny(T) ---> gyujtemeny(T)-T ; semmi.
eleme(_Gy-E, E).
eleme(Gy-_, E) :- eleme(Gy, E).

| ?- eleme(semmi-1-2, E).
        E = 2 ? ; E = 1 ? ; no
| ?- eleme(semmi-(kal-kapolna)-hatvan, kal).
        no
| ?- eleme(semmi-(kal-kapolna)-hatvan, X).
        X = hatvan ? ;
        X = kal-kapolna ? ; no
```

Körmentes út keresése — javított megoldás

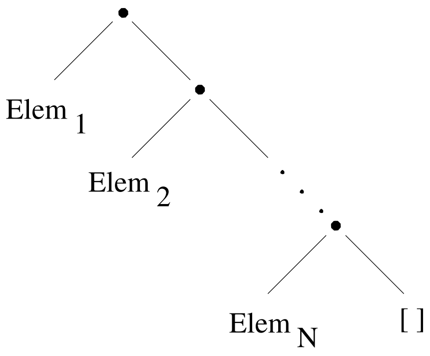
```
% útvonal_3(N, A, B, H): A és B között van (pontosan)
% N szakaszból álló körmentes útvonal, amelynek összhossza H.
útvonal_3(N, Honnan, Hová, H) :-
        útvonal_3(N, Honnan, Hová, semmi-Honnan, H).

% útvonal_3(N, A, B, K, H): A és B között van pontosan
% N szakaszból álló körmentes, K elemein át nem menő H hosszú út.
útvonal_3(0, Hová, Hová, _, 0).
útvonal_3(N, Honnan, Hová, Kizártak, H) :-
        N > 0, N1 is N-1,
        útszakasz(Honnan, Közben, H1),
        \+ eleme(Kizártak, Közben),
        útvonal_3(N1, Közben, Hová, Kizártak-Közben, H2), H is H1+H2.

| ?- útvonal_3(2, kál-kápolna, hatvan, H).
        H = 50 ? ; no
```

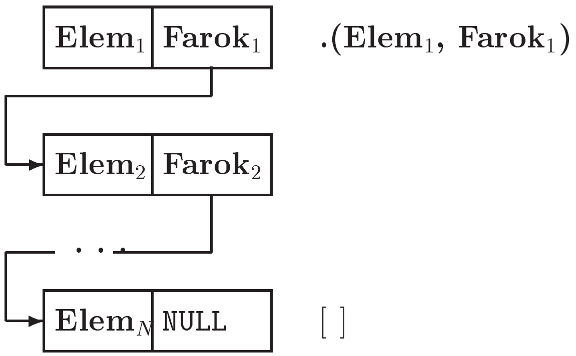
LISTÁK PROLOGBAN

A Prolog lista-fogalma

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

A listák fastruktúra alakja és megvalósítása





Listák jelölése — szintaktikus édesítőszerek

- `[Fej|Farok]` ≡ `.(Fej, Farok)`
- [Elem$_1$,Elem$_2$,...,Elem$_N$|Farok] ≡ [Elem$_1$|[Elem$_2$|[... [Elem$_N$|Farok] ...]]]
- [Elem$_1$,Elem$_2$,...,Elem$_N$] ≡ [Elem$_1$,Elem$_2$,...,Elem$_N$|[]]

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

Listaelemek keresése: `member(E, L): E az L lista eleme`

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

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

Eldöntendő kérdés

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

Megválaszolandó kérdések

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

Vegyes használat, listák metszete

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

Lista elemévé tesz, végtelen választás!

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

member/2 általánosítása: select/3

```
% select(Elem, Lista, Marad): Elemet a Lista-ból elhagyva marad Marad.
select0(Elem, [Elem|Marad], Marad).    % Elhagyjuk a fejet, marad a farok.
select0(Elem, [X|Farok], Marad) :-
        select0(Elem, Farok, Marad0), % A farokból hagyunk el elemet,
        Marad = [X|Marad0].          % a maradék elé tesszük a fejet.

% A második klóz tömörebben (logikai stílusban) --- jobbrekurzív!
select(Elem, [Elem|Marad], Marad).
select(Elem, [X|Farok], [X|Marad0]) :-
        select(Elem, Farok, Marad0).

| ?- select(X, [1,2,3], L).
        L=[2,3], X=1 ? ;  L=[1,3], X=2 ? ;   L=[1,2], X=3 ? ; no
| ?- select(3, L, [1,2]).
        L = [3,1,2] ? ; L = [1,3,2] ? ; L = [1,2,3] ? ; no
| ?- select(3, [2|L], [1,2,7,3,2,1,8,9,4]).
        no   % a logikai stílusban 1 lépés, a funkcionálisban 10!
```

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

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

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

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

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

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

Listák összefűzése: az append/3 eljárás

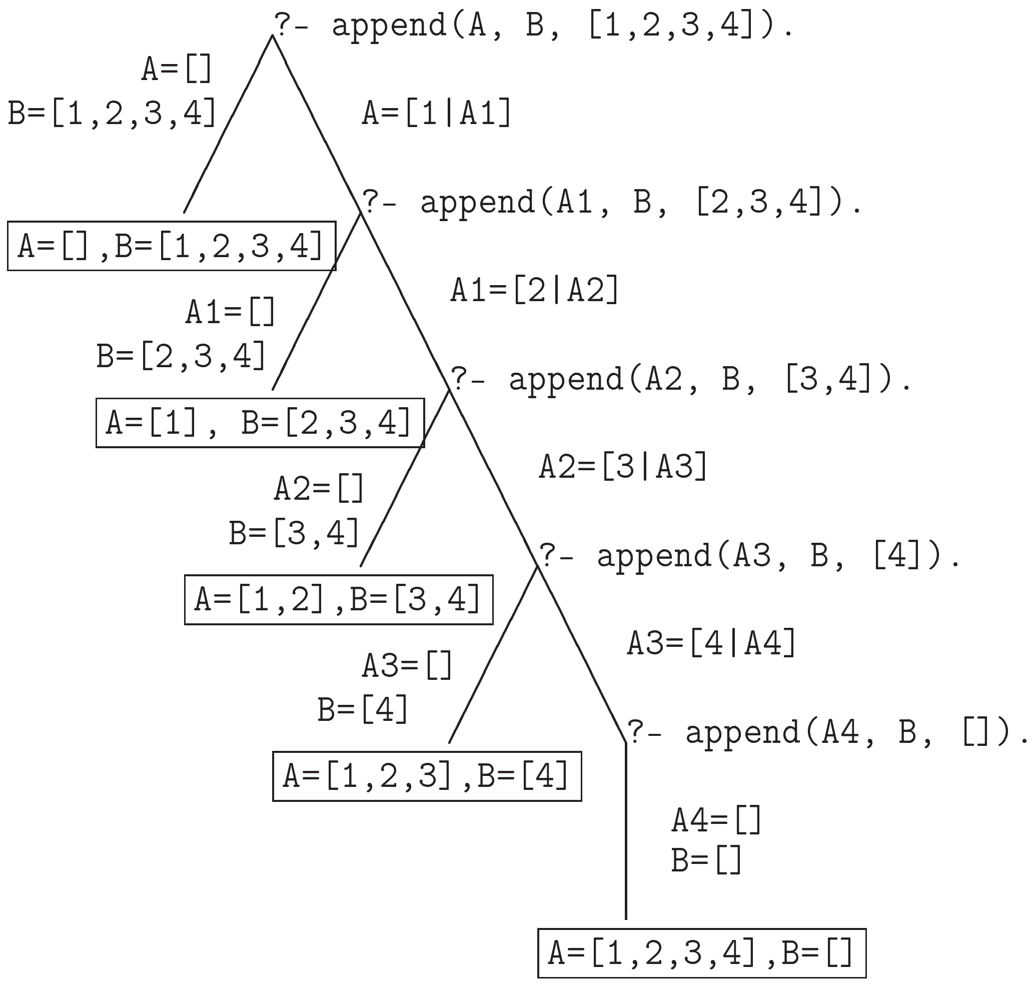
```
% append(L1, L2, L3): Az L3 lista az L1 és L2 listák elemeinek
% egymás után fűzésével áll elő (jelöljük: L3 = L1⊕L2).
append([], L, L).
append([X|L1], L2, [X|L3]) :-
        append(L1, L2, L3).
```

```
| ?- trace, append([1,2], [3,4], L).
        1      1 Call: append([1,2],[3,4],L) ?
        2      2 Call: append([2],[3,4],L3) ? g
Ancestors:
        1      1 Call: append([1,2],[3,4],[1|L3])
        2      2 Call: append([2],[3,4],L3) ?
        3      3 Call: append([],[3,4],L31) ? g
Ancestors:
        1      1 Call: append([1,2],[3,4],[1,2|L31])
        2      2 Call: append([2],[3,4],[2|L31])
        3      3 Call: append([],[3,4],L31) ?
        3      3 Exit: append([],[3,4],[3,4]) ?
```

Az `append(L1, ...)` **komplexitása: futási ideje arányos** `L1` **hosszával (ha** `L1` **zárt).**

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

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

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

Szétszedésre is alkalmas, hatékony változat

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

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

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

Egy érdekes feladvány

Egy szakadékon egy hosszú és keskeny palló ível át, amely egyszerre legfeljebb két embert bír el. A palló egyik oldalán áll négy ember: 10, 20, 50 és 100 évesek. Az N éves ember N perc alatt tud átmenni a hídon. Ha ketten mennek át akkor a lassabb embernek megfelelő idő alatt érnek át.

Sötét van, világítás nélkül lehetetlen átérni, de a társaságnak csak egyetlen zseblámpája van. A feladat: megszervezni az átkelést úgy, hogy a teljes társaság a lehető legrövidebb idő alatt átkeljen a túloldalra.

Kérdés: mennyi idő alatt tudnak leggyorsabban átkelni?

Prolog megoldás: a következő előadáson!

Variációk appendre 2. — lista folytonos része

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

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

A két változat hatékonyságának összehasonlítása

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

Mintakeresés append/3-mal

Párban előforduló elemek

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

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

Dadogó részek

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

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

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 a `member/2`, `select/3`, `append/3` és `reverse/2` eljárások definícióját:

```
:- 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**

```
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;
}
```

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

A `select/3` továbbfejlesztése

```
% cserél(X, XL, Y, YL): egy X elemet XL-ben Y-ra cserélve kapjuk YL-t.
% Deklaratívabban: X ugyanannyiadik eleme XL-nek, mint Y YL-nek, és a két
% lista csak ebben az elemben különbözik.
% A végességhez XL és YL közül legalább az egyik zárt végű kell legyen.
cserél(X, [X|Xlista], Y, [Y|Xlista]).
cserél(X, [Head|Xlista], Y, [Head|Ylista]) :-
        cserél(X, Xlista, Y, Ylista).

| ?- cserél(2, [1,2,3], 4, L).
        L = [1,4,3] ? ;
        no
| ?- cserél(X, [1,2,3], 4, L).
        L = [4,2,3], X = 1 ? ;
        L = [1,4,3], X = 2 ? ;
        L = [1,2,4], X = 3 ? ;
        no
| ?- cserél(X, [1,2,3], Y, L).
        L = [Y,2,3], X = 1 ? ;
        L = [1,Y,3], X = 2 ? ;
        L = [1,2,Y], X = 3 ? ;
        no
```

2000 tavaszi kis házi feladat

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

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

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

2000 tavaszi kis házi feladat — számjegyek fordított sorrendben

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

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

A kétféle irányú gyűjtés összehasonlítása

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

2000 tavaszi kis házi feladat — egyszerűsítés

fszám/3 egyszerűsíthető

- fszám/4 **minden hívása** fszám(_,_,[],_) **alakú.**
- fszám(Sz, A, [], Jk) ⇒fszám12(Sz, A, Jk)

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

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

Körmentes út keresése — megoldás listák használatával

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

% útvonal_4(N, A, B , Út, H): A és B között van (pontosan)
% N szakaszból álló körmentes Út útvonal, amelynek összhossza H.
útvonal_4(N, Honnan, Hová, Út, H) :-
        útvonal_4(N, Honnan, Hová, [Honnan], Út, H).

% útvonal_4(N, A, B, K, Út, H): A és B között van pontosan
% N szakaszból álló körmentes, K elemein át nem menő H hosszú Út út.
útvonal_4(0, Hová, Hová, Kizártak, Út, 0) :-
        reverse(Kizártak, Út).
útvonal_4(N, Honnan, Hová, Kizártak, Út, H) :-
        N > 0, N1 is N-1,
        útszakasz(Honnan, Közben, H1),
        \+ member(Közben, Kizártak),
        útvonal_4(N1, Közben, Hová, [Közben|Kizártak], Út, H2), H is H1+H2.

| ?- útvonal_4(2, 'Párizs', _, Út, H).
        H = 1900, Út = ['Párizs','Bécs','Berlin'] ? ;
        H = 1510, Út = ['Párizs','Bécs','Budapest'] ? ; no
```

Súlyozott gráf ábrázolása éllistával

A gráf ábrázolása

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

Típus-definíció

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

Példa

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

Ismétlődésmentes útvonal keresése listával ábrázolt gráfban

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

% útvonal_5(N, G, A, B, L, H): A G gráfban van egy  A-ból
% B-be menő N szakaszból álló L út, melynek összhossza H.
útvonal_5(0, _Gráf, Hová, Hová, [Hová], 0).
útvonal_5(N, Gráf, Honnan, Hová, [Honnan|Út], H) :-
        N >0, N1 is N-1,
        select(Él, Gráf, Gráf1),
        él_végpontok_hossz(Él, Honnan, Közben, H1),
        útvonal_5(N1, Gráf1, Közben, Hová, Út, H2),
        H is H1+H2.

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

| ?- hálózat(_Gráf), útvonal_5(2, _Gráf, 'Budapest', _, Út, H).
        H = 880, Út = ['Budapest','Bécs','Berlin'] ? ;
        H = 1510, Út = ['Budapest','Bécs','Párizs'] ? ;
        no
```

A PROLOG SZINTAXIS

A Prolog szintaxis összefoglalása

A Prolog szintaxis alapelvei

- Minden programelem kifejezés!
- A szükséges összekötő jelek (`','`, `;`, `:-` `->`): szabványos operátorok.
- A beolvasott kifejezést funktora alapján osztályozzuk:
  - ***kérdés:*** `?- Cél.`
    `Cél`t lefuttatja, és a változó-behelyettesítéseket kiírja.
  - ***parancs:*** `:- Cél.`
    A `Cél`t csendben lefuttatja. Különféle deklarációkat parancsként helyezhetünk el a programban.
  - ***szabály:*** `Fej :- Törzs.`
    A szabályt felveszi a programba.
  - ***nyelvtani szabály:*** `Fej --> Törzs.`
    Prolog szabállyá alakítja és felveszi (lásd a DCG nyelvtanokat).
  - ***tényállítás:*** `Minden egyéb kifejezés.`
    Üres törzsű szabályként felveszi a programba.

A Prolog nyelv-változatok

A SICStus rendszer két üzemmódja

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

Kifejezések szintaxisa — kétszintű nyelvtanok

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

$$
\begin{array}{lcl}
\langle\,\text{kifejezés}\,\rangle ::= & & \langle\,\text{tag}\,\rangle \\
 & | & \langle\,\text{kifejezés}\,\rangle\ \langle\,\text{additív művelet}\,\rangle\ \langle\,\text{tag}\,\rangle \\
\langle\,\text{tag}\,\rangle ::= & & \langle\,\text{tényező}\,\rangle \\
 & | & \langle\,\text{tag}\,\rangle\ \langle\,\text{multiplikatív művelet}\,\rangle\ \langle\,\text{tényező}\,\rangle \\
\langle\,\text{tényező}\,\rangle ::= & & \langle\,\text{szám}\,\rangle \mid \langle\,\text{azonosító}\,\rangle \mid (\ \langle\,\text{kifejezés}\,\rangle\ )
\end{array}
$$

- Ugyanez kétszintű nyelvtannal:

$$
\begin{array}{lcl}
\langle\,\text{kifejezés}\,\rangle ::= & & \langle\,\text{kif}\ \mathit{2}\,\rangle \\
\langle\,\text{kif}\ \boldsymbol{N}\,\rangle ::= & & \langle\,\text{kif}\ \boldsymbol{N\text{-}1}\,\rangle \\
 & | & \langle\,\text{kif}\ \boldsymbol{N}\,\rangle\ \langle\,\boldsymbol{N}\ \text{prioritású művelet}\,\rangle\ \langle\,\text{kif}\ \boldsymbol{N\text{-}1}\,\rangle \\
\langle\,\text{kif}\ \boldsymbol{0}\,\rangle ::= & & \langle\,\text{szám}\,\rangle \mid \langle\,\text{azonosító}\,\rangle \mid (\ \langle\,\text{kif}\ \mathit{2}\,\rangle\ )
\end{array}
$$

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

Kifejezések szintaxisa

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

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

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

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

Kifejezések szintaxisa — folytatás

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

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

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

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

⟨ szám ⟩ ::= ⟨ előjeltelen szám ⟩
| `+` ⟨ előjeltelen szám ⟩
| `-` ⟨ előjeltelen szám ⟩

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

Kifejezések szintaxisa — megjegyzések

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

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

A Prolog lexikai elemei 1. (ismétlés)

⟨ név ⟩

- kisbetűvel kezdődő alfanumerikus jelsorozat (ebben megengedve kis- és nagybetűt, számjegyeket és aláhúzásjelet);
- egy vagy több ún. speciális jelből (`+-*/\$^<>=‘~:.?@#&`) álló jelsorozat;
- az önmagában álló `!` vagy `;` jel;
- a `[]` `{}` jelpárok;
- idézőjelek (’) közé zárt tetszőleges jelsorozat, amelyben \ jellel kezdődő escape-szekvenciákat is elhelyezhetünk.

⟨ változó ⟩

- nagybetűvel vagy aláhúzással kezdődő alfanumerikus jelsorozat.
- az azonos jelsorozattal jelölt változók egy klózon belül azonosaknak, különböző klózokban különbözőeknek tekintődnek;
- kivétel: a semmis változók (_) minden előfordulása különböző.

A Prolog lexikai elemei 2.

⟨ természetes szám ⟩

- (decimális) számjegysorozat;
- 2, 8 ill. 16 alapú számrendszerben felírt szám, ilyenkor a számjegyeket rendre a `0b`, `0o`, `0x` karakterekkel kell prefixálni (csak `iso` módban)
- karakterkód-konstans `0'`*`c`* alakban, ahol *`c`* egyetlen karakter

⟨ lebegőpontos szám ⟩

- mindenképpen tartalmaz tizedespontot
- mindkét oldalán legalább egy (decimális) számjeggyel
- `e` vagy `E` betűvel jelzett esetleges exponens

Megjegyzések és formázó-karakterek

Megjegyzések (comment)

- A % százalékjeltől a sor végéig
- A /* jelpártol a legközelebbi */ jelpárig.

Formázó elemek

- szóköz, újsor, tabulátor, stb. (nem látható karakterek)
- megjegyzés

A programszöveg formázása

- formázó elemek (szóköz, újsor, stb.) szabadon elhelyezhetők;
- kivétel: struktúrakifejezés neve után nem szabad;
- prefix operátor és (közé kötelező;
- ⟨záró-pont⟩: egy . karakter amit egy formázó elem követ.

TÍPUSOK PROLOGBAN

Típusok leírása Prologban

- Típusleírás: (tömör) Prolog kifejezések egy halmazának megadása
- Alaptípusok leírása: `int`, `float`, `number`, `atom`, `any`
- Új típusok felépítése:
  $\{$ `str`$(\mathtt{T}_1,\ \ldots,\ \mathtt{T}_n)$ $\} \equiv \{$ `str`$(e_1,\ \ldots,\ e_n) \mid e_1 \in \mathtt{T}_1,\ \ldots,\ e_n \in \mathtt{T}_n \}, n \geq 0$

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

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

  `% :- type t1 == {atom-atom} \/ atom.`, `% :- type ember == {ember-atom} \/ {semmi}.`
- Megkülönböztetett únió: csupa különböző funktorú összetett típus úniója.
  Egyszerűsített jelölés:
  `:- type` T `== {` S_1 `} \/ ...\/ {` S_n `}.` $\Rightarrow$ `:- type` T `--->` S_1 `; ...;` S_n.

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

Típusok leírása Prologban — folytatás

Paraméteres típusok — példák

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

% :- type szótár == assoc_list(szó, szó).
% :- type szó == atom.
```

Típusdeklarációk szintaxisa

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

Predikátum-deklarációk

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

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

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

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

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

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

A HÍD FELADVÁNY

A feladat

Egy szakadékon egy hosszú és keskeny palló ível át, amely egyszerre legfeljebb két embert bír el. A palló egyik oldalán áll négy ember: 10, 20, 50 és 100 évesek. Az N éves ember N perc alatt tud átmenni a hídon. Ha ketten mennek át akkor a lassabb embernek megfelelő idő alatt érnek át.

Sötét van, világítás nélkül lehetetlen átérni, de a társaságnak csak egyetlen zseblámpája van. A feladat: megszervezni az átkelést úgy, hogy a teljes társaság a lehető legrövidebb idő alatt átkeljen a túloldalra.

Kérdés: mennyi idő alatt tudnak leggyorsabban átkelni?

Az adatstruktúrák

```
% :- type állapot ---> lámpa-list(ember).  % Hol a lámpa és hol vannak az emberek?
% :- type lámpa == oldal.                   % A lámpa az egyik oldalon lehet.
% :- type ember == list(pair(int,oldal)).   % Minden adott korú ember mellett
                                            % ott a tartózkodási helye.
% :- type oldal ---> bal ; jobb.
```

Az állapotátmenet

```
% útszakasz(Áll0, Áll1, Idő): Áll0-ból egy lépésben Áll1-be lehet jutni Idő alatt.
% :- pred útszakasz(áll::in, áll::out, int::out).
útszakasz(Innen-Helyzet0, Ide-Helyzet, Idő) :-
        másik(Innen, Ide),     % A lámpa az Innen oldalon van, Ide a túloldal.
        cserél(E1-Innen, Helyzet0, E1-Ide, Helyzet),       % E1 megy át
        Idő = E1.
útszakasz(Innen-Helyzet0, Ide-Helyzet, Idő) :-
        másik(Innen, Ide),
        cserél(E1-Innen, Helyzet0, E1-Ide, Helyzet1),      % E1 átmegy
        cserél(E2-Innen, Helyzet1, E2-Ide, Helyzet),       % E2 átmegy
        E1 > E2, Idő = E1.

% másik(Egyik, Másik): Egyik és Másik két különböző oldal.
% :- pred másik(oldal, oldal).
másik(bal, jobb).
másik(jobb, bal).

% véghelyzet(Hol, Áll): Az Áll állapotban a zseblámpa és az emberek a Hol
% oldalon vannak.
% :- pred véghelyzet(oldal, áll).
véghelyzet(Hol, Hol-[10-Hol,20-Hol,50-Hol,100-Hol]).
```

Futtatás

```
% Át lehet menni a bal véghelyzetből a jobb véghelyzetbe
% Hossz Idő alatt az Út állapotlistán keresztül.
% :- pred átmegy(list(áll)::out, int::out).
átmegy(Út, Hossz) :-
        véghelyzet(bal, Kezd),
        véghelyzet(jobb, Vég),
        between(1, 10, N),
        útvonal_4(N, Kezd, Vég, Út, Hossz).


| ?- átmegy(Út, H).
        H = 190, Út = [bal-[10-bal,20-bal,50-bal,100-bal],... - ...]] ?
yes
| ?- átmegy(Út, H), H < 190.
        H = 170, Út = [bal-[10-bal,20-bal,50-bal,100-bal],... - ...]] ?
yes
| ?- átmegy(Út, H), H < 170.
        no
        % mintegy 20 másodperc után!
```

2. megoldás — a keresési tér korlátozása, lépések gyűjtése

```
% :- type lépés ---> átmenők>oldal.     % az adott oldalra átmennek az átmenők.
% :- type átmenők == int \/ {int+int}.  % egy vagy két adott korú ember.

% útszakasz(Áll0, Áll1, Idő, Lépés): Egy Lépés lépéssel Idő alatt az Áll0
% állapotból az Áll1 állapotba lehet jutni.
% :- pred útszakasz(áll::in, áll::out, int::out, lépés::out).
útszakasz(Innen-Helyzet0, Ide-Helyzet, Idő, E1>Ide) :-     % E1 megy át.
        másik(Innen, Ide),
        cserél(E1-Innen, Helyzet0, E1-Ide, Helyzet),       % E1 átmegy
        Idő = E1.
útszakasz(Innen-Helyzet0, Ide-Helyzet, Idő, E1+E2>Ide) :-  % E1+E2 megy át.
        másik(Innen, Ide),
        cserél(E1-Innen, Helyzet0, E1-Ide, Helyzet1),      % E1 átmegy
        cserél(E2-Innen, Helyzet1, E2-Ide, Helyzet),       % E2 átmegy
        E1 > E2, Idő = E1.
```

```
% Lépések-kel át lehet menni a bal véghelyzetből a jobb véghelyzetbe
% Hossz idő alatt, ahol Hossz < Max.
% :- pred átmegy(int::in, list(lépés)::out, int::out).
átmegy(Max, Lépések, Hossz) :-
        véghelyzet(bal, Kezd), véghelyzet(jobb, Vég),
        útvonal_6(Kezd, Vég, Max, Lépések, Nyereség),
        Hossz is Max - Nyereség.

% útvonal_6(A, B, Max, Lk, Nyer): A és B között van egy Max-nál
% Nyer-rel rövidebb út (Nyer > 0), amelynek lépéssorozata Lk.
% :- pred útvonal_6(áll::in, áll::in, int::in, list(lépés)::out, int::out).
útvonal_6(Hová, Hová, Max, [], Max)  :- Max > 0.
útvonal_6(Honnan, Hová, Max0, [Lép|LépL], Nyer) :-
        Max0 > 0,
        útszakasz(Honnan, Közben, H1, Lép),
        Max1 is Max0-H1,
        útvonal_6(Közben, Hová, Max1, LépL, Nyer).

| ?- átmegy(190, Lk, H).
        H = 170, Lk = [20+10>jobb,10>bal,100+50>jobb,20>bal,20+10>jobb] ? ;
        H = 170, Lk = [20+10>jobb,20>bal,100+50>jobb,10>bal,20+10>jobb] ? ;
        no
```

1. kis házi feladat

Adott egy lista, amelynek elemei `piros/1`, `feher/1` vagy `zold/1` funktorú struktúrák, tetszőleges sorrendben. A struktúrák argumentuma tetszőleges Prolog kifejezés lehet, ezek az argumentumok a feladat szempontjából érdektelenek.

A feladat a lista rendezése úgy, hogy az elején álljanak a `piros/1` funktorúak, utánuk `feher/1` funktorúak, végül pedig a `zold/1` funktorúak. Az egyes csoportokon belül az elemek sorrendje ne változzék.

Írjon egy olyan `zaszlo/2` Prolog eljárást, amely megvalósítja a leírt rendezést. Ha a fenti háromtól különböző funktorú elem van a listában, akkor az eljárás hiusúljon meg.
Törekedjék arra, hogy a megoldás hatékony legyen! Vigyázzon arra is, hogy az eljárás ne sikerüljön többször!

Pontérték: 1 plusz pont
Beadási határidő: 2001 március 19, 24:00
Beadás módja: a honlapon meghirdetendő módon.

1. kis házi feladat — folytatás

```
% :- lista == list(szin).
% :- szin ---> piros(any) ; feher(any) ; zold(any).
% zaszlo(Bemenet, Kimenet): a Kimenet lista a Bemenet lista elemeinek
% fent leírt módon rendezett listája.
% :- pred zaszlo(lista::in, lista::out).
```

Példák

```
| ?- zaszlo([piros(a),kek(b)], Z).
        no
| ?- zaszlo([zold(c),piros(d),feher(e)], Z).
        Z = [piros(d),feher(e),zold(c)] ? ;
        no
| ?- zaszlo([piros(f),zold(g),piros(h),feher(i)], Z).
        Z = [piros(f),piros(h),feher(i),zold(g)] ? ;
        no
```