Bevezetés a Logikai Programozásba

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

Számítástudományi
és Információelméleti Tanszék

IQSOFT Rt.

Az előadássorozat áttekintése

Bevezetés

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 megjelenése (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:

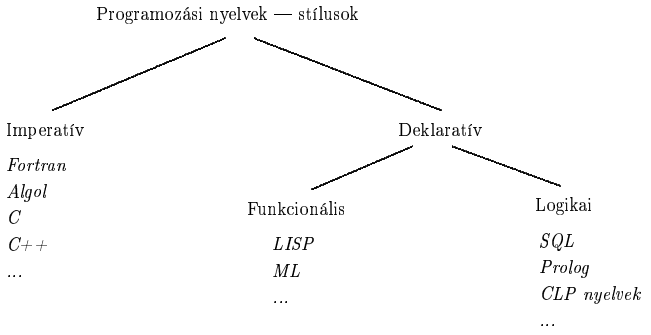
`http://www.cs.cmu.edu/afs/cs/project/ai-repository/ai/lang/prolog/faq/prg_[12].faq`

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

Programozási nyelvek osztályozása



Imperatív és deklaratív programozás

Imperatív program:

- felszólító módú
- utasítások
- változó: változtatható értékű memóriahely

Deklaratív programozás

- kijelentő módú
- egyenletek, állítások
- változó: egy ismeretlen de (előbb–utóbb) rögzített értékű mennyiség

Jelszavak

- MIT és nem HOGYAN
- egyszeres értékadás (single assignment)

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

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

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 — az ML funkcionális nyelven írt megoldás

```
fun szulo "Imre"    = ["István", "Gizella"]
  | szulo "István"  = ["Géza", "Sarolt"]
  | szulo "Gizella" = ["Civakodó Henrik",
                       "Burgundi Gizella"]
  | szulo _         = [];

fun nagyszulok g = List.concat (map szulo (szulo g));
```

Tömörebb megoldás

```
val nagyszulok = List.concat o (map szulo) o szulo;
```

A függvény futtatása

```
- nagyszulok "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

```
szülője('Imre', 'István').
szülője('Imre', 'Gizella').
szülője('István', 'Géza').
szülője('István', 'Sarolt').
szülője('Gizella', 'Civakodó Henrik').
szülője('Gizella', 'Burgundi Gizella').

nagyszülője(Gyerek, Nagyszülő) :-
    szülője(Gyerek, Szülő),
    szülője(Szülő, Nagyszülő).
```

Futtatás

```
| ?- nagyszülője('Imre', NSz).

NSz = 'Géza' ? ;
NSz = 'Sarolt' ? ;
NSz = 'Civakodó Henrik' ? ;
NSz = 'Burgundi Gizella' ? ;

no
| ?- nagyszülője(U, 'Géza').

U = 'Imre' ? ;

no
```


A logikai programozás

Alapeszme

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

Következmények

- egyszerű szemantika
- könnyű verfikálhatóság

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

Egy összetettebb példa

```
% őse(E0, N, E): E0 embernek N-edik generációs őse
% az E ember.
őse(E,  0, E).
őse(E0, N, E) :-
        szülője(E0, Sz),
        őse(Sz, N1, E),
        N is N1+1.
```

Futása:

```
| ?- őse('Imre', I, Ős).
I = 0, Ős = 'Imre' ? ;
I = 1, Ős = 'István' ? ;
I = 2, Ős = 'Géza' ? ;
I = 2, Ős = 'Sarolt' ? ;
I = 1, Ős = 'Gizella' ? ;
I = 2, Ős = 'Civakodó Henrik' ? ;
I = 2, Ős = 'Burgundi Gizella' ? ;
no
| ?- őse(Utód, I, 'Burgundi Gizella').
I = 0, Utód = 'Burgundi Gizella' ? ;
I = 2, Utód = 'Imre' ? ;
I = 1, Utód = 'Gizella' ? ;
no
| ?-
```


Egy „igazi” program

Faktoriális Prologban

```
% 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.
```

Faktoriális ML-ben

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


Prolog — előnyök és hátrányok

Miért jó?

- deklaratív, könnyen ellenőrizhető
- 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
- egyes beépített eljárások algoritmikusak
- gyenge következtetési képesség

Hogyan tovább?

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

A Prolog nyelv alapjai

A Prolog programok elemei

Tényállítások

```
    % járat(A, B, T): Az A és B városok között
    % van járat, melynek úthossza T km.
(1) járat('Budapest', 'Prága', 515).
(2) járat('Budapest', 'Bécs', 245).
(3) járat('Bécs', 'Berlin', 635).
(4) járat('Bécs', 'Párizs', 1265).
```

```
| ?- járat('Budapest', 'Bécs', 245).
yes
```

```
| ?- járat('Budapest', 'Bécs', Táv).
Táv = 245 ?
```

```
| ?- járat('Bécs', Cél, Táv).
Cél = 'Berlin', Táv = 635 ? ;
Cél = 'Párizs', Táv = 1265 ? ;
no
```

Egy összetett kérdés

```
| ?- járat('Budapest', Közben, T1),
     járat(Közben, Cél, T2), T1+T2 > 1000.
T1 = 245, T2 = 1265, Cél = 'Párizs', Közben = 'Bécs' ? ;
no
```

kérdés$_1$: `járat('Budapest', Közben, T1), járat(Közben, Cél, T2), T1+T2>1000.`
illesztés: (1) $\Rightarrow$ `Közben = 'Prága', T1 = 515`

kérdés$_2$: `járat('Prága', Cél, T2), 515+T2>1000.`
illesztés: nincs

kérdés$_1$: `járat('Budapest', Közben, T1), ...`
illesztés: (2) $\Rightarrow$ `Közben = 'Bécs', T1 = 245,`

kérdés$_3$: `járat('Bécs', Cél, T2), 245+T2>1000.`
illesztés: (3) $\Rightarrow$ `Cél = 'Berlin', T2 = 635`

kérdés$_4$: `245+635 > 1000.`
illesztés: sikertelen

kérdés$_3$: `járat('Bécs', Cél, T2), 245+T2>1000.`
illesztés: (4) $\Rightarrow$ `Cél = 'Párizs', T2 = 1265`

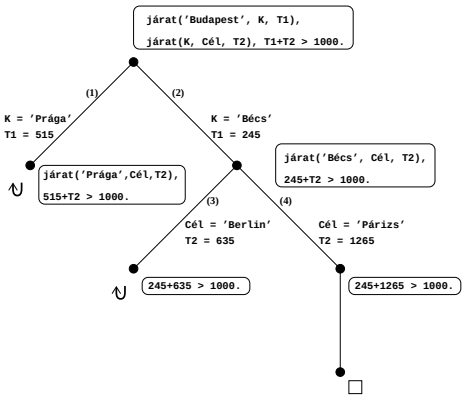
kérdés$_5$: `245+1265 > 1000.`
illesztés: sikeres

kérdés$_6$: üres

Végső behelyettesítések:

`Közben = 'Bécs', T1 = 245, Cél = 'Párizs', T2 = 1265.`

Keresési fa



Egy teljes párbeszéd

```
> sicstus
SICStus 3.8: Thu Oct  7 14:58:41 MET DST 1999
| ?- consult(jarat).
{consulting /home/user/jarat.pl...}
{/home/user/jarat.pl consulted, 0 msec 688 bytes}

yes
| ?- járat('Bécs', Cél, Táv).

Cél = 'Berlin', Táv = 635 ? ;

Cél = 'Párizs', Táv = 1265 ? ;

no
| ?- járat(Honnan, 'Párizs', Táv).

Táv = 1265, Honnan = 'Bécs' ?

yes
| ?- consult(user).
| végállomás('Budapest').
| végállomás('Párizs').
| end_of_file.
{user consulted, 0 msec 424 bytes}

yes
| ?- halt.
>
```


Szabályok

```
% járat2(Kezdet, Cél, Táv): Kezdet városból vezet egy
%     két szakaszból álló,  Táv km hosszú út Cél-ba.
járat2(Kezdet, Cél, Táv) :-
    járat(Kezdet, Közben, T1),
    járat(Közben, Cél, T2),
    Táv is T1+T2.
```

A szabály használata

```
| ?- járat2('Budapest', Cél, T), T > 1000.
T = 1510, Cél = 'Párizs' ? ;
no
```

A szabály, mint makró

```
járat('Budapest', Közben, T1), járat(Közben, Cél, T2),
T is T1+T2, T > 1000.
```

Szabályok — folytatás

```
% járat2(Kezdet, Cél, Táv): Kezdet városból vezet egy
% két szakaszból álló,  Táv km hosszú út Cél-ba.
járat2(Kezdet, Cél, Táv) :-
    járat(Kezdet, Közben, T1),
    járat(Közben, Cél, T2),
    Táv is T1+T2.
```

A szabály jelentése

```
járat2(Kezdet, Cél, Táv) igaz ha
    járat(Kezdet, Közben, T1) igaz és
    járat(Közben, Cél, T2) igaz és
    Táv is T1+T2 igaz.

(tetszőleges Kezdet, Cél, Táv értékek esetén)
Kezdet városból vezet egy két szakaszból álló,
Táv km hosszú út Cél-ba, ha
    (létezik olyan Közben érték, hogy)
        a Kezdet és Közben városok között van járat,
            melynek úthossza T1 km és
        a Közben és Cél városok között van járat,
            melynek úthossza T2 km és
        Táv = T1+T2.
```


Predikátum definiálása több szabállyal

```
% járatszakasz(A, B, H): A és B között, vagy B és A
%    között van járat, amelynek úthossza H.
járatszakasz(Kezdet, Cél, Táv) :-
    járat(Kezdet, Cél, Táv).
járatszakasz(Kezdet, Cél, Táv) :-
    járat(Cél, Kezdet, Táv).

| ?- járatszakasz('Bécs', Hová, H).
H = 635, Hová = 'Berlin' ? ;
H = 1265, Hová = 'Párizs' ? ;
H = 245, Hová = 'Budapest' ? ;
no
```

Diszjunkció

```
% járatszakasz(A, B, H): A és B között, vagy B és A
%    között van járat, amelynek úthossza H.
járatszakasz(Kezdet, Cél, Táv) :-
    (   járat(Kezdet, Cél, Táv)
    ;   járat(Cél, Kezdet, Táv)
    ).
```


Rekurzív szabályok

```
% útvonal(N, A, B, Táv): A és B között van (pontosan)
% N szakaszból álló útvonal, amelynek összhossza Táv.
útvonal(0, Kezdet, Kezdet, 0).
útvonal(N, Kezdet, Cél, Táv) :-
    N > 0,
    N1 is N-1,
    járatszakasz(Kezdet, Közben, TávA),
    útvonal(N1, Közben, Cél, TávB),
    Táv is TávA+TávB.

| ?- útvonal(2, 'Budapest', Hová, Táv), Táv > 1000.
Táv = 1030, Hová = 'Budapest' ? ;
Táv = 1510, Hová = 'Párizs' ? ;
no
```

Az 1. klóz egy változata

```
útvonal(N, Honnan, Hová, Táv) :-
    N = 0, Hová = Honnan, Táv = 0.
```


Egy aritmetikai példa

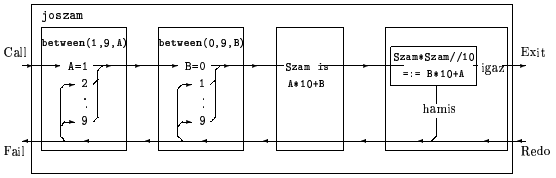
Példa: „jó" számok

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

```
% Szam négyzete háromjegyű és a Szam
% fordítottjával kezdődik.
joszam(Szam):-
        between(1, 9, A),
        between(0, 9, B),
        Szam is A * 10 + B,
        Szam * Szam // 10 =:= B * 10 + A.

% 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).
```

A 4-kapus doboz modell

```
% Szam négyzete háromjegyű és a Szam
% fordítottjával kezdődik.
joszam(Szam):-
        between(1, 9, A),
        between(0, 9, B),
        Szam is A * 10 + B,
        Szam * Szam // 10 =:= B * 10 + A.
```




```
| ?- trace, joszam(X).
        1      1 Call: joszam(_181) ?
        2      2 Call: between(1,9,_649) ? s
?       2      2 Exit: between(1,9,1) ?
        3      2 Call: between(0,9,_642) ? s
?       3      2 Exit: between(0,9,0) ?
        4      2 Call: _181 is 1*10+0 ?
        4      2 Exit: 10 is 1*10+0 ?
        5      2 Call: 10*10//10=:=0*10+1 ?
        5      2 Fail: 10*10//10=:=0*10+1 ?
        3      2 Redo: between(0,9,0) ? s
?       3      2 Exit: between(0,9,1) ?
```


Prolog programok végrehajtása

Szemléletek

- tételbizonyítási folyamat (SL rezolúciónak),
- célvezérelt keresési folyamat, vagy
- általánosított eljáráshívási folyamat.

Tételbizonyítás	**Célvezérelt keresés**	**Eljárás-szervezés**
∗ **predikátum**		∗ **eljárás**
∗ **klóz, állítás**	∗ **redukciós szabály**	**eljárás-ág**
következmény ← **feltétel**	**minta** ⇒ ∗ **célsorozat**	∗ **fej :- törzs**
feltétel	∗ **cél**	∗ **eljáráshívás**
rezolúciós lépés	∗ **redukciós lépés**	**eljáráshívási lépés**

Paraméter-átadás — egyesítés

```
% útvonal(N, A, B, Táv): A és B között van (pontosan)
% N szakaszból álló útvonal, amelynek összhossza Táv.
útvonal(0, Kezdet, Kezdet, 0).
útvonal(N, Kezdet, Cél, Táv) :-
    N > 0,
    N1 is N-1,
    járatszakasz(Kezdet, Közben, TávA),
    útvonal(N1, Közben, Cél, TávB),
    Táv is TávA+TávB.
```

1. példa

Hívás:	`útvonal(0, 'Budapest', Cél, Táv)`
Fej:	`útvonal(0, Kezdet, Kezdet, 0).`
Behely.:	`Kezdet = 'Budapest', Cél = 'Budapest', Táv = 0`
Közös alak:	`útvonal(0, 'Budapest', 'Budapest', 0)`

2. példa

Hívás:	`útvonal(2, 'Budapest', Hová, Táv)`
Fej:	`útvonal(N, Kezdet, Cél, Táv)`
Behely.:	`N = 2, Kezdet = 'Budapest', Cél = Hová`
Közös alak:	`útvonal(2, 'Budapest', Hová, Táv)`

A redukciós lépés

```
útvonal(0, Kezdet, Kezdet, 0).
útvonal(N, Kezdet, Cél, Táv) :-
    N > 0,
    N1 is N-1,
    járatszakasz(Kezdet, Közben, TávA),
    útvonal(N1, Közben, Cél, TávB),
    Táv is TávA+TávB.
```

Célsorozat:

```
útvonal(2, 'Budapest', Hová, Táv), Táv > 1000
```

Redukció a 2. klózzal:

Behelyettesítés:

```
N = 2, Kezdet = 'Budapest', Cél = Hová
```

Redukált célsorozat:

```
2 > 0, N1 is 2-1,
járatszakasz('Budapest', Közben, TávA),
útvonal(N1, Közben, Hová, TávB),
Táv is TávA+TávB, Táv > 1000.
```

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 vermeljük `<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.

Egy végrehajtási példa

	Célsorozat (CS) *behelyettesítések*	Klóz (I)	Verem	Lépés

(cs1)	`útvonal(2, 'Budapest', Hová, Táv),`
	`Táv > 1000,`
	`answer(Hová), answer(Táv).`
		2	üres	4.

(cs2)	`2 > 0, N1 is 2-1,`
	`járatszakasz('Budapest', Közben, TávA),`
	`útvonal(N1, Közben, Hová, TávB),`
	`Táv is TávA+TávB, Táv > 1000, ...`
	`N1 = 1`
		–		2., 2.

(cs3)	`járatszakasz('Budapest', Közben, TávA),` `útvonal(1, Közben, Hová, TávB), ...`	1	<cs3,1>	4.

(cs4)	`járat('Budapest', Közben, TávA),`
	`útvonal(1, Közben, Hová, TávB), ...`
	`Közben = 'Prága', TávA = 515`
		1	<cs4,1>
			<cs3,1>
				4.

(cs5)	`útvonal(1, 'Prága', Hová, TávB),` `Táv is 515+TávB, Táv > 1000, ...`	2		4.

(cs6)	`járatszakasz('Prága', Közben_1, TávA_1),`
	`útvonal(0, Közben_1, Hová, TávB_1),`
	`TávB is TávA_1+TávB_1, Táv is 515+TávB,`
	`Táv > 1000, ...`
	`N1_1 = 0`
		1	<cs6,1>
			<cs4,1>
			<cs3,1>
				2., 2., 4.

(cs7)	`járat('Prága', Közben_1, TávA_1), ...`			4., 6., 7.
(cs6)	`járatszakasz('Prága', Közben_1, TávA_1),` `...`	2	<cs4,1> <cs3,1>	4.

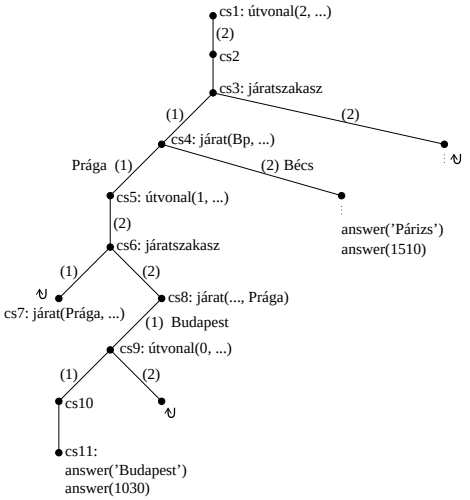
(cs8)	`járat(Közben_1, 'Prága', TávA_1),`
	`útvonal(0, Közben_1, Hová, TávB_1), ...`
	`Közben_1 = 'Budapest', TávA_1 = 515`
		1	<cs8,1>
			<cs4,1>
			<cs3,1>
				4.

(cs9)	`útvonal(0, 'Budapest', Hová, TávB_1),`
	`TávB is 515+TávB_1, Táv is 515+TávB,`
	`Táv > 1000, answer(Hová), answer(Táv).`
	`Hová ='Budapest', TávB_1 = 0`
		1	<cs9,1>
			<cs8,1>
			<cs4,1>
			<cs3,1>
				4.

(cs10)	`TávB is 515+0, Táv is 515+TávB,`
	`Táv > 1000,`
	`answer('Budapest'), answer(Táv).`
	`TávB = 515, Táv = 1030`
		–		2., 2., 2.

(cs11)	`answer('Budapest'), answer(1030).`			

A példaprogram keresési fája



A futás

```
| ?- útvonal(2, 'Budapest', Hová, Táv), Táv > 1000.
Táv = 1030, Hová = 'Budapest' ? ;
Táv = 1510, Hová = 'Párizs' ? ;
no
```


Összetett adatstruktúrák — listák

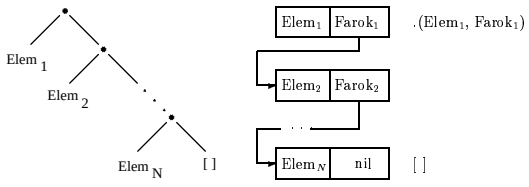
Listák

- üres lista: `'[]'` név, ua., mint `[]`
- nem üres lista: `[Fej|Farok]` ua., mint `.(Fej,Farok)`
- „normális” esetben, a `Farok` is lista.

N elemű lista — szintaktikus egyszerűsítés

- teljes forma: $[Elem_1|[Elem_2|[\ldots\ [Elem_N|[]]\ \ldots]]]$
- egyszerűsített forma: $[Elem_1, Elem_2, \ldots, Elem_N]$

A listák fastruktúra alakja és megvalósítása



Listák — folytatás

Lista-minták

- üres lista: `[]`
- nem üres lista: `[Fej|Farok]`, ahol `Farok` változó
- `[Elem`$_1$`,Elem`$_2$`,...,Elem`$_N$`|Farok]` ahol `Farok` változó. Ez az `[Elem`$_1$`|[Elem`$_2$`|[... [Elem`$_N$`|Farok] ...]]]` alakkal azonos.

Szétszedés

```
% feje(Lista, Fej): Lista feje Fej.
feje([Fej|Farok], Fej).

% farka(Lista, Farok): Lista farka Farok.
farka([Fej|Farok], Farok).

| ?- feje([1,2], X).
X = 1
```

A hívás szabványos alakja: `feje([1|[2|[]]], X)`.
Behelyettesítés: `Fej = 1, Farok = [2|[]], X = 1`

Eldöntendő kérdések

```
| ?- feje([1,2], 1).
yes
| ?- feje([1,2,3], 2).
no
```


Listaelemek keresése

```
% member(Elem, Lista):  Elem a Lista elemeként előfordul.
member(Elem, Lista) :-
    feje(Lista, Elem).
member(Elem, Lista) :-
    farka(Lista, Farok),
    member(Elem, Farok).
```

Eldöntendő kérdés

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

Kiegészítendő kérdés (felsorolás)

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

Vegyes használat — két lista metszete

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


A `member/2` tisztázása

`feje(Lista, Fej)` ⇒ `Lista = [Fej|Farok]`

Az 1. klóz új alakja

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

Vagy még egyszerűbben:

```
member(Elem, [Elem|Farok]).
```

Tiszta változat

A `farok/2` eliminálása után:

```
% member(Elem, Lista):  Elem a Lista elemeként előfordul.
member(Elem, [Elem|_]).
member(Elem, [_|Farok]) :-
    member(Elem, Farok).
```

A `member/2` általánosítása

```
% select(Elem, Lista, Marad):  Elem a Lista elemeként
% előfordul, kihagyása után marad a Marad lista.
select(Elem, [Elem|Marad], Marad).
select(Elem, [Egyéb|Farok], [Egyéb|Farok1]) :-
    select(Elem, Farok, Farok1).
```


Listák összefűzése

Funkcionális stílus

```
% append(L1, L2, L3):  Az L3 lista az L1 és L2 listák
% elemeinek egymás után fűzésével áll elő (L3 = L1⊕L2)
append([], L, L).
append([X|L1], L2, L12) :-
    append(L1, L2, L3), L12 = [X|L3].
```

A 2. klóz logikai stílusban

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

Logikai stílus előnyei

- részleges behelyettesítettség ⇒ jobbrekurzió
- gyors nemleges válasz: `append([1,2], [3], [2,3,4])`

Az `append` komplexitása

- `append(L1, ...)` futási ideje arányos `L1` hosszával.

Listák szétszedése

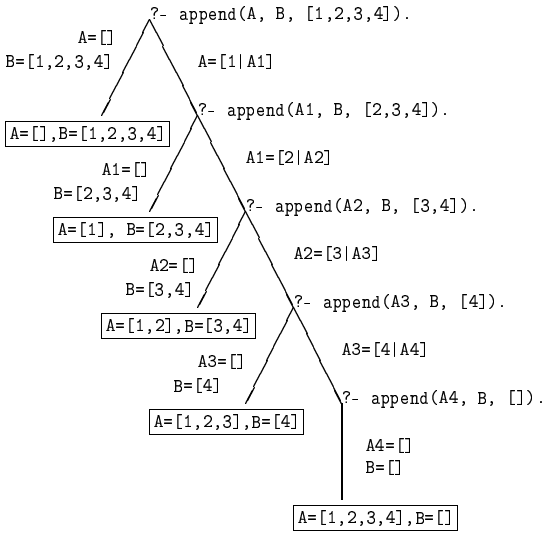
Az iménti append/3 eljárás

```
% 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).
```

Hívása

```
| ?- 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
```


Az append/3 futás keresési (végrehajtási) fája



Variációk appendre 1.

Három lista összefűzése

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

Nem hatékony

- `append([1,...,100],[1,...,1000],[1,...,10000])`

Szétszedésre nem alkalmas

- append/3 „biztonságosan” működik, ha
  - vagy az első arg. konkrét lista
  - vagy a harmadik arg. konkrét lista
- ellenkező esetben végtelen választási pont

Szétszedésre is alkalmas változat

```
% L1 ⊕ L2 ⊕ L3 = L123 ha L123 vagy L1, L2, L3 adott
append(L1, L2, L3, L123) :-
    append(L1, L23, L123), append(L2, L3, L23).
```


Variációk appendre 2.

Lista folytonos része

```
% L123 = _ ⊕ L2 ⊕ _ ha L123 adott
csublist(L2, L123) :-
    append(_L1, L23, L123), append(L2, _L3, L23).
% L123 = _ ⊕ L2 ⊕ _ ha L2, L123 adott
check_csublist(L2, L123) :-
    append(L2, _L3, L23), append(_L1, L23, L123).
```

Hatékonyság

- L123 = [$\underbrace{0,1,2,3,4}_{\text{x10000}}$,...,10], L2 = [0,1,2,3,4,10]
- csublist(L2, L123): 340 msec
- check_csublist(L2, L123): 160 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).
```


Körmentes út keresése

Az út ábrázolása

- Egy [Város_1,Város_2,...,Város_N] lista, ahol
- Város_1 a kiindulási hely, Város_N a célpont, és
- a szomszédos elemek között van `járatszakasz`.

Az eljárás

```
% útvonal_2(N, A, B, L): Az A-ból B-be menő pontosan N
%   szakaszból álló körmentes út állomásainak listája L.
útvonal_2(N, Honnan, Hová, Út) :-
    útvonal_2(N, Honnan, Hová, [Honnan], Út).

% útvonal_2(N, A, B, K, L): L az A-ból B-be menő pontosan
% N szakaszból álló körmentes, K elemein át nem menő út.
útvonal_2(0, Hová, Hová, _, [Hová]) :- !.
útvonal_2(N, Honnan, Hová, Kizártak, [Honnan|Út]) :-
    N > 0, N1 is N-1,
    járatszakasz(Honnan, Közben, _),
    \+ member(Közben, Kizártak),
    útvonal_2(N1, Közben, Hová, [Közben|Kizártak], Út).
```

Egy futás

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


A Prolog negáció

`\+ member(Közben, Kizártak)` jelentése: `Közben` nem eleme a `Kizártak` listának.

Pontosabban:

- `\+ Cél` jelentése:
- `Cél` nem bizonyítható (logikailag)
- `Cél` futása sikertelen (procedurálisan)

A vágó (!) beépített eljárás

```
útvonal_2(...)  :- !.
útvonal_2(...)  :- ...
```

A vágó eljárás mindig sikerül, és leszűkíti a keresési teret. Ebben a példában letiltja az adott eljárás későbbi klózainak választását.

A példabeli vágó ún. „zöld” vágó, mert csak egy amúgy is meghiúsuló klózt vág le.

Összetett adatstruktúrák — rekord-struktúrák

Formája

- *rekordnév*(A_1,A_2,...,A_N), ahol
- *rekordnév* egy névkonstans,
- A_1,...,A_N tetszőleges Prolog objektumok

Példák

- súlyozott gráf éle: `él(Kezdő, Vég, Súly)`
- lista (speciális ábrázolású rekord): `[Fej|Farok]`
- lista belső alakja: `.(Fej, Farok)`

Összetett adatstruktúrák — példa

Irányítatlan gráf éleinek felsorolása

```
% Gráf gráfnak egy éle Kezdőből Végbe megy Súly súllyal.
éle(Gráf, Kezdő, Vég, Súly) :-
    member(él(Kezdő, Vég, Súly), Gráf).
éle(Gráf, Kezdő, Vég, Súly) :-
    member(él(Vég, Kezdő, Súly), Gráf).

| ?- éle([él(a,b,5),él(c,a,2)], a, V, S).
V = b, S = 5 ? ;
V = c, S = 2 ? ;
no
```

Rekordok szétszedése/összerakása mintaillesztéssel

Hívás:	`member(él(a, V, S), [él(a,b,5)|(...)])`
Fej:	`member(Elem, [Elem|_])`
Behely.:	`V = b, S = 5, Elem = él(a, b, 5)`
Közös alak:	`member(él(a, b, 5), [él(a,b,5)|(...)])`

Útkeresés éllistával

Adatábrázolás:

- `él(Honnan, Hová, Hossz)` elemek listája
- `Hossz` most figyelmen kivül marad

```
% útvonal_3(N, G, A, B, L): A G gráfban van egy  A-ból
% B-be menő N szakaszból álló L út.
útvonal_3(0, _Gráf, Hová, Hová, [Hová]) :- !.
útvonal_3(N, Gráf, Honnan, Hová, [Honnan|Út]) :-
        N >0, N1 is N-1,
        select(Él, Gráf, Gráf1),
        él_végpontok(Él, Honnan, Közben),
        útvonal_3(N1, Gráf1, Közben, Hová, Út).

% él_végpontok(Él, A, B): Az Él irányítatlan él
% végpontjai A és B.
él_végpontok(él(A,B,_), A, B).
él_végpontok(él(A,B,_), B, A).

| ?- _Gráf = [él('Budapest','Bécs',245),
              él('Budapest','Prága',515),
              él('Bécs','Berlin',635),
              él('Bécs','Párizs',1265)],
     útvonal_3(2, _Gráf, 'Budapest', _, Út).
Út = ['Budapest','Bécs','Berlin'] ? ;
Út = ['Budapest','Bécs','Párizs'] ? ;
no
```


Az egyesítési algoritmus

A behelyettesítés

- Egy függvény, ami változókhoz kifejezéseket rendel.
- Pl. $\sigma = \{\mathtt{X} := \mathtt{a}, \mathtt{Y} := \mathtt{s(b,B)}, \mathtt{Z} := \mathtt{C}\}$
- Alkalmazása: $\mathtt{f(g(Z,h),A,Y)}\sigma = \mathtt{f(g(C,h),A,s(b,B))}$
- Kompozíciójuk: $\sigma' \otimes \sigma''$

A Prolog adatok szabványos alakja

- Egy fa, a csomópontjaiban rekordnevek,
- a levelekben számok, nevek, változók.

Az egyesítési algoritmus

- két (fa struktúrájú) kifejezés sorozat egyesítése lehet
- sikeres $\Rightarrow$ behelyettesítés
- sikertelen, és ilyenkor nincs behelyettesítés

Az algoritmus

Az egyesítendő kifejezés-sorozatok: $A_1,\ldots,A_N$ ill. $B_1,\ldots,B_N$, az eredő behelyettesítés σ.

1. Ha $N > 1$, akkor ha A_1 és B_1 egyesíthető σ'-vel, és $(A_2,\ldots,A_N)\sigma'$ és $(B_2,\ldots,B_N)\sigma'$ egyesíthető σ''-vel, akkor $\sigma = \sigma' \otimes \sigma''$.

A továbbiakban feltételezhetjük, hogy $N = 1$.

2. Ha A_1 és B_1 azonos változók vagy konstansok, akkor $\sigma = \emptyset$.
3. Egyébként, ha A_1 változó, akkor $\sigma = \{A_1 := B_1\}$.
4. Egyébként, ha B_1 változó, akkor $\sigma = \{B_1 := A_1\}$.
5. Ha A_1 és B_1 azonos nevű és argumentumszámú összetett kifejezések és argumentum-listáik egyesíthetőek σ'-vel, akkor $\sigma = \sigma'$.
6. Minden más esetben az egyesítés meghiúsul.

Előfordulás-ellenőrzés (*occurs check*)

Változó–struktúra egyesítésnél ellenőrizni kell, hogy a változó nem fordul-e elő a struktúrában. Ha igen, akkor az egyesítésnek elvben meg kell hiúsulnia.

A Prolog rendszerek többsége alaphelyzetben nem alkalmazza.

`unify_with_occurs_check/2` — szabványos beépített eljárás.

SICStus kiterjesztés: ciklikus kifejezések: `X = f(1,X)`.

Egyesítési példa

Egyesítendő

- `member(él(a, V, S), [él(a,b,5),él(c,a,2)])`
- `member(Elem, [Elem|_])`

Lépések

A_1: `él(a,V,S)`,	A_2: `.(él(a,b,5),.(él(c,a,2),[]))`
B_1: `Elem`,	B_2: `.(Elem,_)`

4. $\{$`Elem` := `él(a,V,S)`$\}$

A_2: `.(él(a,b,5),.(él(c,a,2),[]))`	
B_2: `.(él(a,V,S),_)`	

5. $\emptyset$

A_1': `él(a,b,5)`,	A_2': `.(él(c,a,2),[])`
B_1': `él(a,V,S)`,	B_2': `_`

5. $\emptyset$
2. $\emptyset$
4. $\{$`V` := `b`$\}$
4. $\{$`S` := `5`$\}$
4. $\{$`_` := `.(él(c,a,2),[])`$\}$ (nem érdekes)

Az eredő behelyettesítés

$\{$`Elem` := `él(a,b,5)`, `V` := `b`, `S` := `5`$\}$

Operátorok

Szintaktikus édesítőszer

- az operátoros alakok is „közönséges” struktúrák
- például `T is T1+T2` szabványos alakja `is(T, +(T1,T2))`
- az egyesítés nem aritmetika: `1-1 = 0` meghiúsul!

Operátor-deklaráció

- `:- op(`*`prioritás, fajta, operátornév`*`)`.
- *`operátornév`* tetszőleges atom
- *`prioritás`* 0–1200 közötti egész
- *`fajta`*
  - infix: `yfx`, `xfy`, `xfx`
  - prefix: `fx`, `fy`
  - postfix: `xf`, `yf`
- 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

Operátorok — folytatás

Szintaktikus édesítőszer 2.

- a Prolog a szabványos alakot „látja”
- `a+X = a+b+c` meghiúsul, hiszen `a+b+c = (a+b)+c`

Szabványos operátorok

- a Prolog program Prolog kifejezések sorozata
- pl. `is/2` és `:-` és a vessző szabványos operátorok
- a vessző prioritása 1000 $\Rightarrow$ argumentum-listában zárójelezés nélkül max. 999-es prioritású operátorok

Beépített operátorok

A beépített 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 szintaxis összefoglalása

A Prolog szintaxis alapelvei

- minden programelem kifejezés
- szükséges összekötő jelek: szabványos operátorok
- a beolvasott kifejezést funktora alapján osztályozzuk

Programelemek osztályozása

kérdés `?-` *`Cél`*`.`
(A *`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`*`.`

nyelvtani szabály *`Fej`* `->` *`Törzs`*`.` (Lásd a DCG nyelvtanokat).

tényállítás *`Minden egyéb kifejezés.`*

SICStus végrehajtási módok

- `iso` Az ISO Prolog szabványnak megfelelő.
- `sicstus` Korábbi változatokkal kompatibilis.
- Állítása: `set_prolog_flag(language,` *`Mód`*`)`

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 kell álljon}
| `(` ⟨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⟩

Megjegyzések

- A ⟨kifejezés N⟩-ben ⟨köz⟩ csak akkor kell ha az őt követő kifejezés `(`-lel 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, pl.

```
| ?- write("baba").
[98,97,98,97]
```

A Prolog lexikai elemei

⟨**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.

A Prolog lexikai elemei — folytatás

⟨**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 (decimális) egy 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 szabadon elhelyezhetők;
- kivétel: struktúrakifejezés neve után nem szabad;
- prefix operátor és `(` közé kötelező;
- ⟨záró-pont⟩: egy `.` karakter amit egy formázó elem követ.

Típusok Prologban

A Prolog alapvetően nem típusos nyelv

Típusfogalmat vezet be pl:

- Visual Prolog (Turbo Prolog)
- Mercury

Típusfogalom előnyei:

- fordítás-idejű hibajelzés
- könnyebb olvashatóság, karbantarthatóság

Típustalan Prologban is célszerű a típusinformációkat kommentárként leírni!

Típusinformációk

A Mercury típusfogalmát és -szintaxisát használjuk, csekély módosítással.

Típusinformációk fajtái:

- adattípus-deklarációk
- predikátumtípus-deklarációk

Adattípus-deklarációk:

- Alaptípusok:
  `int, float, number, atom, univ`
- Összetett típusok
  - típuskonstrukciók
  - típusátnevezések

Típusdeklarációk

Típuskonstrukciók

⟨típuskonstrukció⟩ ::= `:- type` ⟨típusazonosító⟩ `-->` ⟨típustörzs⟩
⟨típustörzs⟩ ::= ⟨konstruktor⟩ `; ...`
⟨konstruktor⟩ ::= ⟨név⟩ | ⟨rekordnév⟩`(`⟨típusazonosító⟩, ...`)`
⟨típusazonosító⟩ ::= ⟨név⟩ | ⟨név⟩`(`⟨változó⟩, ...`)`

Példák

```
:- type gyumolcs ---> alma ; korte ; szilva.

:- type szemely ---> szemely(atom, atom, int).
          % egy szemely típusú adat két
          % argumentuma atom, a harmadik egész.

:- type binarisfa ---> ures
        ;              bfa(int, binarisfa, binarisfa).

:- type list(T) ---> []
        ;            [T|list(T)].

:- type pair(T1, T2) ---> T1 - T2.
          % egy olyan '-' nevű kétargumentumú struktúra,
          % amelynek első argumentuma T1, második
          % argumentuma pedig T2 típusú.
```


Típusdeklarációk — folytatás

Típusátnevezések

⟨típusátnevezés⟩ ::= `:- type` ⟨típusazonosító⟩ `==` ⟨típusazonosító⟩

Példák:

```
:- type hossz == int.

:- type assoc_list(KeyType, ValueType)
              == list(pair(KeyType, ValueType)).

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


Predikátum-deklarációk

Predikátumtípus-deklaráció

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

Példák:

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

Predikátummód-deklaráció

`:- mode` ⟨eljárásnév⟩`(`⟨módazonosító⟩, ...`)`
ahol ⟨módazonosító⟩ ::= `in` | `out`.
Egy eljárásra több móddeklaráció is adható.

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).
```

Prolog programozási módszerek

- **A keresési tér szűkítése**
- **Determinizmus és indexelés**
- **Jobbrekurzió és akkumulátorok**
- **Algoritmusok Prologban**
- **Megoldások gyűjtése és felsorolása**
- **Összes megoldás gyűjtése**
- **Meta-logikai eljárások**
- **Magasabb rendű eljárások**
- **Dinamikus adatbáziskezelés**
- **Modularitás**

A keresési tér szűkítése

Eszközök

- a vágó beépített eljárás: `!`
- feltételes diszjunktív szerkezet:
  `( felt -> akkor ; egyebkent )`

Motiváció

- mi tudjuk, hogy ott nincs megoldás (zöld vágó),
- el akarunk dobni megoldásokat (vörös vágó).

Példák a vágó eljárás használatára

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


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


% unió(+S1, +S2, ?U): Az ismétlődés nélküli listával
% ábrázolt S1 és S2 halmazok uniója U.
unió([], S, S).
unió([E|S1], S2, U) :-
       member(E, S2), !, % vörös vágó
       unió(S1, S2, U).
unió([E|S1], S2, [E|U]) :-
       /* \+ member(E, S2), */
       unió(S1, S2, U).
```


A vágó definíciója

Segédfogalom: cél szülője

- Egy cél *szülője* az a cél, amelyik az őt tartalmazó klóz fejével illesztődött
- A `g` (ancestors) nyomkövetési parancs kiírja a kurrens cél szülőjét, annak szülőjét, stb.

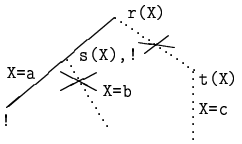
```
| ?- trace, fakt(3, F).
{The debugger will first creep -- showing everything (trace)}
        1      1 Call: fakt(3,_129) ?
(...)
       18      4 Call: fakt(0,_4565) ? g
Ancestors:
        1      1 Call: fakt(3,_129)
        4      2 Call: fakt(2,_891)
       15      3 Call: fakt(1,_2728)
       18      4 Call: fakt(0,_4565) ?
```

A vágó végrehajtása

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

Egy általános példa

```
q(X):- s(X).
q(X):- t(X).

r(X):- s(X), !.
r(X):- t(X).

s(a).
s(b).

t(c).

:- q(X), write(X), fail.

abc

:- r(X), write(X), fail.

a
```



A vágás két alapesete

- Első megoldásra való megszorítás: az `s(X)` további megoldásainak letiltása
- Elkötelezettség az adott klóz mellett: az `r/1` második klózának letiltása

Első megoldásra való megszorítás

Eldöntendő kérdés

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

Végtelen választási pontok megszelidítése

- lásd később, pl. `memberchk/2`

Feladatspecifikus optimalizálás

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


Klóz mellett való elkötelezés

Feltételes szerkezet — példa

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

Általános alak

```
p :- felt, !, akkor.
p :- /* nem_felt, */egyébként.
```

Diszjunktív feltételes szerkezet

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

Általános alak

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


Példa: két szám maximuma

```
% max(X, Y, Z): X és Y maximuma Z.
% 1. változat, tiszta Prolog
max(X, Y, X) :- X >= Y.
max(X, Y, Y) :- Y > X.

% 2. változat zöld vágó felhasználásával
max(X, Y, X) :- X >= Y, !.
max(X, Y, Y) :- Y > X.

% 3., rossz változat, vörös vágó felhasználásával
max(X, Y, X) :- X >= Y, !.
max(X, Y, Y).
```

Ez hibás, pl. `max(10, 1, 1)` sikerül.

```
% 4., helyes változat, vörös vágó felhasználásával
max(X, Y, Z) :- X >= Y, !, Z=X.
max(X, Y, Y) /* :- Y > X */.
```

A kimenő paraméterek értékadását mindig a vágó után végezzük!

Példa: listában való előfordulás ellenőrzése

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

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

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

`memberchk/2` **használata**

- Eldöntő kérdésre, pl.:

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

  visszalépéskor nem keresi végig a lista maradékát.
- Nyílt végű lista elemévé tesz, pl.:

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


Példa: szótárprogram

```
szotaraz(Sz):-
        read(M-A), !,
        memberchk(M-A,Sz),
        write(M-A), nl,
        szotaraz(Sz).
szotaraz(_).
```

Futása:

```
| ?- szotaraz(Sz).
|: alma-apple.
alma-apple
|: korte-pear.
korte-pear
|: alma-X.
alma-apple
|: X-pear.
korte-pear
|: vege.

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

yes
```


Determinizmus

Hívások **osztályozása**

- determinisztikus — (legfeljebb) egyféleképpen sikerül
- nem-determinisztikus — többször sikerül

Egy hívás ***determinisztikusan fut le***

- ha nem hagy választasi pontot a híváshoz tartozó részfában;
- ez természetesen függ a Prolog megvalósítástól is!

SICStus példák

- nyomkövetésbeli ? → *nem*determinisztikus lefutás

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

| ?- p(1, X).
        1      1 Exit: p(1,a)

| ?- p(Y, a).
?       1      1 Exit: p(1,a)

| ?- p(Y, b), Y > 2.
?       1      1 Exit: p(2,b)
        1      1 Exit: p(3,b)
```


A determinisztikus lefutás

Haszna

- a futás gyorsabb lesz,
- a tárigény csökken,
- más optimalizálások (pl. jobbrekurzió) alkalmazható.

Hogyan ismeri fel a fordító a determinizmust?

- indexelés
- vágók és feltételes szerkezetek

Indexelés

Mi az indexelés

- egy hívásra illeszthető klózok gyors kiválasztása,
- egy eljárás klózainak fordítási idejű csoportosításával,
- főként az első fejargumentum külső funktora alapján. (A külső funktor `C` szám vagy névkonstans esetén `C/0`; `R` nevű és `N` argumentumú struktúra esetén `R/N`; változó esetén nem értelmezzük)

Példa

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

A példából generált kód: ha az első aktuális argumentum

- változó →`(1) (2) (3) (4) (5)`
- `0` →`(1) (4)`
- fő funktora `s/1` →`(2) (3) (4)`
- `9` →`(4) (5)`
- minden más esetben →`(4)`

Indexelés és determinizmus

Egy híváshoz a Prolog nem hoz létre választási pontot, ha

- felismeri, hogy determinisztikus
- pl. az indexelési csoport egyelemű

Az alábbi példában

- a `p(`*`Adat`*`, V)` esetén nem jön létre választási pont
- a `p(V, `*`Adat`*`)` hívások létrehoznak választási pontot

```
p(0, a).
p(s(X), Y) :- pp(X, Y).
p(9, z).

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

Választási pont felesleges létrejötte esetén

- vágóval elérhető, hogy a hívás legalább determinisztikusan *fusson le*
- például `p_vissza(Y, X) :- p(X, Y), !.` esetén a `p_vissza(a, X)` hívás determinisztikusan fut le.

Listakezelő eljárások indexelése

A feldolgozandó lista legyen az első argumentum

- az `[]` és `[...|...]` eseteket az indexelés megkülönbözteti
- a két klóz sorrendje nem érdekes (kivéve változó-arg)

Példák

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

% last(L, E): Az L lista utolsó eleme E.
last([E], E).             % nem-determinisztikusan fut le
last([_|L], E) :- last(L, E).

% last1(L, E): Az L lista utolsó eleme E.
last1([E], E) :- !.       % zöld vágó
last1([_|L], E) :-
        last1(L, E).

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

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


Aritmetikai elágaztatások

Az indexelés

- általában nem foglalkozik az aritmetikai vizsgálatokkal
- pl. `N = 0` és `N > 0` feltételek nem „zárják ki” egymást

Példa

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

Az indexelés és a vágó kölcsönhatása

A vágó figyelembevételének feltételei

- A fejben az első argumentum funktora kivételével különböző változók szerepeljenek,
- a törzs első hívása a vágó legyen.
- Például: `p(s(A, B, C), D, E) :- !, ....`

Példák

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

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

```
tarifa(szombat, T) :- !, T = 4.
tarifa(vasárnap, T) :- !, T = 4.
tarifa(_, 10).
```


A vágó és az indexelés hatékonysága

Egy Fibonacci-szerű sorozat

$$f_1 = 1;\quad f_2 = 2;\quad f_n = f_{\lfloor 3n/4 \rfloor} + f_{\lfloor 2n/3 \rfloor},\quad n > 2$$

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

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

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

Futási idők $N = 2000$ esetén

	fib	fibc	fibci
futási idő	4410 ms	4060 ms	3820 ms
meghiúsulási idő	730 ms	0 ms	0 ms
összesen	5140 ms	4060 ms	3820 ms

Jobbrekurzió és akkumulátorok

Jobbrekurzió-optimalizálás

utolsó hívás opt. — last call optimisation)

- az eljárástörzs utolsó hívása esetén,
- feltéve, hogy a törzs eddig determinisztikusan futott le,
- azaz a szülő célig (azt bele nem értve) nincs választási pont.

Egyszerű listaösszegzés

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

% sum1(L, S): Az L számlista elemeinek összege S.
% (sum/2 jobbrekurzív változata.)
sum1(L, S):-
        sum1(L, 0, S).

% sum1(L, S0, S): Az L számlista elemeit hozzáadva
% S0-hoz kapjuk S-et.
sum1([], S, S).
sum1([X|L], S0, S):-
        S1 is S0+X, sum1(L, S1, S).
```


Akkumulátorok

- a hagyományos, „változtatható” változók megfelelői
- párosával járnak: aktuális–végső állapot
- az általános séma:

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

Példa

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

Példa több akkumulátor használatára

```
% sum12(L, S0, S, Q0, Q): S = S0+ΣL, Q = Q0+ΣL^2
sum12([], S, S, Q, Q).
sum12([X|L], S0, S, Q0, Q):-
        S1 is S0+X, Q1 is Q0+X*X,
        sum12(L, S1, S, Q1, Q).
```

Akkumlátorok és listák

A `revapp` is gyűjt

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

Sőt, az `append` is!

```
% append(Xs, L, L0): Xs = L0-L
append([], L, L).
append([X|Xs], L, L0) :-
       L0 = [X|L1],
       append(Xs, L, L1).
```


$a^n b^n$ alakú sorozatok

Első megoldás, $3n$ lépés

```
% anbn(N, L): L = [a, ..., a, b, ..., b]
%                    N db       N db
anbn(N, L) :-
        an(N, a, AN),
        an(N, b, BN),
        append(AN, BN, L).

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

Második megoldás, $2n$ lépés

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

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


$a^n b^n$ alakú sorozatok (folyt.)

Harmadik megoldás, n lépés

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

% anbn(N, L0, L): L = [a, ..., a, b, ..., b |L0]
%                          N db       N db
anbn(0, L0, L) :- !, L = L0.
anbn(N, L0, [a|L]) :-
        N > 0,
        N1 is N-1,
        anbn(N1, [b|L0], L).
```

A második klóz nem jobbrekurzív változata

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

SML megoldás

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


$a^n b^n$ alakú sorozatok (folyt.)

Harmadik megoldás, C++

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

1. kis házi feladat megoldása

Állítsa elő egy N nem negatív egész szám B alapú számrendszerben vett jegyeinek listáját ($B > 1$ 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, Jegyek): A Szám >= 0 szám
% Alap > 1 alapú számrendszerben balról jobbra
% vett jegyeinek listája Jegyek.
szám(Sz, Alap, Jk) :-
        Sz > 0, !, szám(Sz, Alap, [], Jk).
szám(0, _, [0]).

% szám(Szám, Alap, Jk0, Jk): A Szám >= 0 szám
% Alap > 1 alapú számrendszerben balról jobbra
% vett jegyeinek listája Jk-Jk0.
szám(Sz, Alap, Jk0, Jk) :-
        Sz > 0, !, Sz1 is Sz//Alap,
        UtsoJegy is Sz mod Alap,
        szám(Sz1, Alap, [UtsoJegy|Jk0], Jk).
szám(0, _, Jk, Jk).
```


1. kis házi feladat megoldása, folyt.

Számjegyek előállítása fordított sorrendben

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

% fszám(Szám, Alap, Jk0, Jk): A Szám >= 0 szám
% Alap > 1 alapú számrendszerben jobbról balra
% vett jegyeinek listája Jk-Jk0.
fszám(Sz, Alap, Jk0, [UtsoJegy|Jk]) :-
        Sz > 0, !, Sz1 is Sz//Alap,
        UtsoJegy is Sz mod Alap,
        fszám(Sz1, Alap, Jk0, Jk).
fszám(0, _, Jk, Jk).
```

A kétféle irányú gyűjtés összehasonlítása

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

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


1. kis házi feladat — megjegyzések.

Kétféle irányú gyűjtés — egyszerű példa

```
ntol(N, L0, [N|L]) :-
    N > 0, !, N1 is N-1,
    ntol(N1, L0, L).
ntol(0, L0, L0).

| ?- ntol(5, [], L).
L = [5,4,3,2,1] ?
```

```
nig(N, L0, L) :-
    N > 0, !, N1 is N-1,
    nig(N1, [N|L0], L).
nig(0, L0, L0).

| ?- nig(5, [], L).
L = [1,2,3,4,5] ?
```

`fszám/3` egyszerűsíthető

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

% fszám1(Szám, Alap, Jk): A Szám > 0 szám
% Alap > 1 alapú számrendszerben jobbról balra
% vett jegyeinek listája Jk.
fszám1(Sz, Alap, [UtsoJegy|Jk]) :-
        Sz > 0, !, Sz1 is Sz//Alap,
        UtsoJegy is Sz mod Alap,
        fszám1(Sz1, Alap, Jk).
fszám1(0, _, []).
```


Összetettebb adatstruktúrák akkumulálása

Egészek gyűjtése rendezett bináris fában

- Az adatstruktúra:
  `% :- type bfa --> ures ; bfa(int, bfa, bfa).`
- `beszur(BFa0, E, BFa)`: Az `E` egész `BFa0`-ba való beszúrása `BFa`-t eredményezi.
- Itt `BFa0` és `BFa` egy akkumulátor-pár, de az indexelés érdekében `BFa0` az első argumentum-pozícióba kerül

Elem beszúrása bináris fába

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

Akkumulálás bináris fákkal — folyt.

Lista konverziója bináris fába

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

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

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


Akkumulálás bináris fákkal — folyt.

Bináris fa konverziója listába

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

Rendezés bináris fával

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

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


Algoritmusok Prologban

Hatványozás

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

A Prolog eljárás

- háromargumentumú: `hatv(A, H, E)`
- `A`, `H` csak „bemenő” → „sima” paraméterek
- `E`-re a végén is szükség van → akkumulátor-pár
- `E` kezdőértéke 1

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


Hatványozás (folyt.)

A „ciklus”

```
% hatv(H, A, E0, E):
%        E0 * (A**H) = E.
                                % ism:
hatv(0, _, E0, E) :- !, E=E0. %   if (h == 0) return e;
hatv(H, A, E0, E) :-          %
        H > 0,                %
        (   H /\ 1 =:= 1      %   if (h & 1)
        ->  E1 is E0*A        %     e *= a;
        ;   E1 = E0           %
        ),                    %
        H1 is H >> 1,         %   h >>= 1;
        A1 is A*A,            %   a *= a;
        hatv(H1, A1, E1, E).  %   goto ism;
```

A C ciklus és a Prolog eljárás kapcsolata

- C: értékadás → Prolog: új változó
- az új változót minden ágon létre kell hozni (ld. `if`)
- C: ciklus vége → Prolog: rekurzív visszahívás
- a rekurzív hívásban a C változók pillanatnyi értékét tükröző Prolog változó szerepel
- Prolog eljárás fejkommentje → C ciklus *ciklus-invariánsa*

Programhelyesség-bizonyítás

Algoritmusok helyességének bizonyítása

- az előfeltételekből következnek az utófeltételek
- lineáris kódra egyszerű
- a ciklusokat „fel kell vágni” egy *ciklus-invariáns*-sal:
  - az előfeltételekből következik,
  - ha a ciklus elején fennáll, akkor a ciklus végén is (indukció),
  - belőle és a leállási feltételből következik a ciklus utófeltétele.

```
/* hatv(a, h) = a**h */
int hatv(int a0, unsigned h0)
{
  int e = 1, a = a0, h = h0;
  while (h > 0)
  { /* assert( a0**h0 == e * a**h)); */
    assert( abs(pow(a0,h0)-e*pow(a,h)) < 0.00001 );

    if (h & 1)
        e *= a;    /* e1 = e * (a ** (h&1)) */
    h >>= 1;       /* h1 = (h-(h&1))/2 */
    a *= a;        /* a1 = a*a */
  }
  return e;
}
```


Fibonacci sorozat

A C függvény

```
/* fib(0) = 0; fib(1) = 1; fib(n) = fib(n-1)+fib(n-2), n > 1 */
unsigned fib(unsigned n)
{
  unsigned f0 = 0, f1 = 1, t;
  while (n > 0) t = f1, f1 += f0, f0 = t, --n;
  return f0;
}
```

A Prolog eljárás

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

% fib(N, F0, F1, FN):
%    Az F0 és F1 kezdőértékű
%    fib sorozat N. eleme FN.
fib(0, F0, _, F0).
fib(N, F0, F1, F) :-
        N > 0,
        N1 is N-1,
        F2 is F0+F1,
        fib(N1, F1, F2, F).
```

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


Megoldások gyűjtése és felsorolása

Keresési feladat Prolog megoldása

- **gyűjtés** — az összes megoldás összegyűjtése, pl. egy listába;
- **felsorolás** — a megoldások visszalépéses felsorolása.

Kettő hatványainak gyűjtése

```
% L azon H = 2**i alakú egészek listája,
% amelyekre 1 =< H =< Max.
khatvanyok(Max, L) :-
        khatvanyok(1, Max, L).

% L azon H = 2**i alakú egészek listája,
% amelyekre H0 =< H =< Max,
% (ahol H0 maga is 2**j alakú).
khatvanyok(H0, Max, L0) :-
        H0 =< Max, !,
        L0 = [H0|L1],
        H1 is 2*H0,
        khatvanyok(H1, Max, L1).
khatvanyok(_H0, _Max, []) /* :-
        _H0 > _Max */.
```


Kettő hatványainak felsorolása

```
% H = 2**i alakú egész, amelyre 1 =< H =< Max.
khatvany(Max, H) :-
        khatvany(1, Max, H).

% H = 2**i alakú egész, amelyre H0 =< H =< Max.
% (ahol H0 maga is 2**j alakú).
khatvany(H0, Max, H) :-
        H0 =< Max,
        (   H = H0
        ;   H1 is 2*H0,
            khatvany(H1, Max, H)
        ).

/* A gyűjtő változat:
khatvanyok(H0, Max, L0) :-
        H0 =< Max, !,
        L0 = [H0|L1],
        H1 is 2*H0,
        khatvanyok(H1, Max, L1).
khatvanyok(_H0, _Max, []).
*/
```

Nyolcra végződő kettő-hatványok gyűjtése

```
% L azon H = 2**i alakú, 8-as jegyre végződő
% egészek listája, amelyekre 1 =< H =< Max.
khatvanyok8(Max, L) :- khatvanyok8(1, Max, L).

% L azon H = 2**i alakú, 8-asra végződő egészek
% listája, ahol H0 =< H =< Max és H0 = 2**j alakú.
khatvanyok8(H0, Max, L) :-
        következő(H0, Max, E, H1), !,
        L = [E|L1],
        khatvanyok8(H1, Max, L1).
khatvanyok8(_, _, []).

% E a legkisebb olyan 8-ra végződő kettőhatvány,
% amelyre H0 =< E =< Max. H az E-t követő kettőhatvány.
következő(H0, Max, E, H) :-
        H0 =< Max, H1 is H0*2,
        (   H0 mod 10 =:= 8 -> E = H0, H = H1
        ;   következő(H1, Max, E, H)
        ).

% H = 2**i alakú 8-ra végződő egész,  1 =< H =< Max.
khatvany8(Max, H) :- khatvany8(1, Max, H).

% H = 2**i alakú 8-ra végződő egész,  H0 =< H =< Max
% (ahol H0 maga is 2**j alakú).
khatvany8(H0, Max, H) :-
        következő(H0, Max, E, H1),
        (   H = E
        ;   khatvany8(H1, Max, H)
        ).
```


A gyűjtő és felsoroló sémák összehasonlítása

A gyűjtő séma

```
megoldások(V0, Param, L) :-
    következő(V0, Param, E, V1), !,
    L = [E|L1],
    megoldások(V1, Param, L1).
megoldások(_, _, []).
```

A felsoroló séma

```
megoldás(V0, Param, E) :-
    következő(V0, Param, E0, V1),
    (   E = E0
    ;   megoldás(V1, Param, E)
    ).
```


Példa: fennsíkok felsorolása

Egy listában fennsíknak nevezünk:

- egy csupa azonos elemből álló,
- maximális (nem kiterjeszthető),
- legalább kételemű, folytonos részlistát

Fennsíkok felsorolása — 1. megoldás
(gyorsprogramozási szemlélettel)

```
% Az L listában az F pozición egy H hosszú fennsík van.
fennsík0(L, F, H) :-
        Teste = [E,E|_],
        append(Eleje, Teste, L),
        \+ last(Eleje, E),
        length(Eleje, F0), F is F0+1,
        kezdethossz(Teste, H).      % lásd korábban

% Mint fennsík0, de egyetlen eljárással
fennsík1(L, F, H) :-
        Teste = [E,E|_],
        append(Eleje, Teste, L),
        \+ last(Eleje, E),
        length(Eleje, F0), F is F0+1,
        (   append(Ek, Farok, Teste),
            \+ Farok = [E|_] ->
            /* az Ek lista most csupa E-ből áll */
            length(Ek, H)
        ).
```


Fennsíkok felsorolása — 2., hatékony megoldás

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

% Az L0 listában (P0-tól számozva)
% az F pozición egy H hosszú fennsík van.
fennsík(L0, P0, F, H) :-
        első_fennsík(L0, P0, F0, H0, L1),
        (   F = F0, H = H0
        ;   P1 is F0+H0, fennsík(L1, P1, F, H)
        ).

% első_fennsík(L0, P0, F, H, L): L0-ban az első
% fennsík hossza H, az F pozición van (P0-tól számozva),
% a fennsík utáni maradék lista L.
első_fennsík([E,E|L1], P0, F, H, L) :-
        !, F = P0, azonosak(L1, E, 2, H, L).
első_fennsík([_|L1], P0, F, H, L) :-
        P1 is P0+1,
        első_fennsík(L1, P1, F, H, L).

% azonosak(L0, E, H0, H, L): Az L0-L lista H-H0 darab
% E elemből áll, és L nem kezdődik E-vel.
azonosak([X|L0], E, H0, H, L) :-
        E = X, !,
        H1 is H0+1,
        azonosak(L0, E, H1, H, L).
azonosak(L, _, H, H, L).
```

Gyűjtés és felsorolás kapcsolata

Felsorolás gyűjtésből

```
khatvany(Max, H) :-
        khatvanyok(Max, Hk), member(H, Hk).
```

Megoldásgyűjtő beépített eljárások — `findall`

`findall(?Gyűjtő, :+Cél, ?Lista)`: a `Cél` kifejezést eljáráshívásként értelmezi, meghívja és minden egyes megoldásához előállítja `Gyűjtő` egy másolatát. Végül ezeket a `Gyűjtő` értékeket egy listába összegyűjti és egyesíti `Listával`. Példák:

```
| ?- findall(X, (member(X, [1,7,8,3,2,4]), X>3), L).
             L = [7,8,4] ?
| ?- findall(X-Y, (between(1, 3, X),
                   between(1, X, Y)), L).
             L = [1-1,2-1,2-2,3-1,3-2,3-3] ?
```

Gyűjtés felsorolásból

```
khatvanyok(Max, Hk) :-
        findall(H, khatvany(Max, H), Hk).
```


Megoldásgyűjtő eljárások (folyt.)

A `bagof` gyűjtő eljárás

`bagof(?Gyűjtő, :+Cél, ?Lista)`: a `Cél` kifejezést eljáráshívásként értelmezi, és összegyűjti a megoldásait. Ha a `Célban` vannak olyan üres változók, amelyek a `Gyűjtőben` nem szerepelnek, akkor ezek minden behelyettesítését felsorolja és külön-külön összegyűjti a `Gyűjtő` összes megoldását `Listába`. Például:

```
gráf([a-b,a-c,b-c,c-d,b-d]).

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

Ha a `bagof` eljárás második argumentuma `V1^..Vn^Cél` alakú, akkor a `V1,...` változók behelyettesítéseit nem sorolja fel (egzisztenciális kvantifikálás). Például:

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


Megoldásgyűjtő eljárások (folyt.)

Példa kvantor használatára

```
% G gráf fokszámlistája FL. A fokszámlista olyan A-F
% párokból áll, ahol A a gráf egy pontja,
% és F>0 az A pont fokszáma.
fokszámai(G, FL) :-
        bagof(A-F, Vk^(bagof(V, member(A-V, G), Vk),
                       length(Vk, F)), FL).

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

Kvantor elkerülése segédeljárás segítségével

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

fokszámai(G, FL) :-
        bagof(A-F, pont_foka(A, G, F), FL).
```


Megoldásgyűjtő eljárások (folyt.)

A `bagof` és `findall` közötti további különbségek

- Ha `Cél`nak nincs megoldása. `findall` üres listát ad, `bagof` meghiúsul.
- Ha `Gyűjtő` nem tömör (van benne üres változó), akkor
  - `findall` ezeket megoldásonként szisztematikusan új változókra cseréli
  - `bagof` megőrzi a változókat
- A `bagof` végrehajtása időigényesebb.

Példák

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

Megoldásgyűjtő eljárások (folyt.)

A `setof` eljárás

`setof(?Gyűjtő, :+Cél, ?Lista)`: ugyanaz mint `bagof`, de az eredménylistát rendezi (az ismétlődések kiszűrésével). A rendezéshez a minden Prolog kifejezésre alkalmazható `@<` összehasonlító beépített eljárást használja.

Példa

```
gráf([a-b,a-c,b-c,c-d,b-d]).

% Gráf egy pontja P.
pontja(P, Gráf) :-
        member(P-_, Gráf).
pontja(P, Gráf) :-
        member(_-P, Gráf).

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

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


Meta-logikai eljárások

Olyan „nem logikus” beépített eljárások, amelyek

a. a Prolog kifejezések pillanatnyi behelyettesítettségi állapotát tekintik:

- kifejezések osztályozása
- kifejezések rendezése

b. kifejezéseket szétszednek vagy összeraknak

- (struktúra) kifejezés $\Longleftrightarrow$ név és argumentumok
- atomok és számok $\Longleftrightarrow$ karaktereik

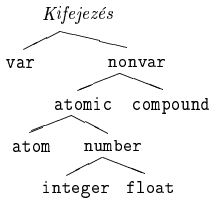
Az a. típusú eljárások általában sorrend-függőek:

```
| ?- var(X) /* X változó? */, X = 1.
X = 1 ?
yes
| ?- X = 1, var(X).
no
| ?- X @< 3 /* X megelőzi 3-t? */, X = 4.
   % a változók megelőzik a nem változó kifejezéseket
X = 4 ?
yes
| ?- X = 4, X @< 3.
no
```


Kifejezések osztályozása

Kifejezés-osztályok fastruktúrája

- egyben egyargumentumú ellenőrző eljárások is



Mire használhatók az osztályozó eljárások?

- `var`, `nonvar` — többirányú eljárások esetén a különböző irányok elágaztatására.
- `number`, `atom`, ... — általános adatstruktúrák, pl. aritmetikai kifejezések feldolgozásánál.
- ...

Többirányú eljárások elágaztatása

Példa: a `length/2` beépített eljárás megvalósítása

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

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

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

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

| ?- length(L, Len).
L = [], Len = 0 ? ;
L = [_A], Len = 1 ? ;
L = [_A,_B], Len = 2 ? ;
L = [_A,_B,_C], Len = 3 ? ;
L = [_A,_B,_C,_D], Len = 4 ?
```

Szimbolikus kifejezés-feldolgozás

Deriválás

```
% deriv(Kif, X, D):  Kif-nek az X atom szerinti
%   deriváltja D.  Kif a +, -, *, /  műveletekkel
%   atomokból és számokból felépített kifejezés.
deriv(X, X, D) :- !, D = 1.
deriv(C, _X, D) :-
      atomic(C), !, D = 0.
deriv(U+V, X, DU+DV) :-
      deriv(U, X, DU), deriv(V, X, DV).
deriv(U-V, X, DU-DV) :-
      deriv(U, X, DU), deriv(V, X, DV).
deriv(U*V, X, DU*V + U*DV) :-
      deriv(U, X, DU), deriv(V, X, DV).
deriv(U/V, X, (DU*V - U*DV)/(V*V)) :-
      deriv(U, X, DU), deriv(V, X, DV).

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

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


Struktúrák szétszedése és összerakása

```
+Kif =..  ?Lista
-Kif =..  +Lista
```

Argumentumok:

`Kif` Az argumentum egy tetszőleges kifejezés.

`Lista` Az argumentum egy lista, az első eleme egy név vagy egy szám, a többi eleme tetszőleges kifejezés. A lista első eleme csak akkor lehet szám, ha több eleme már nincsen.

Jelentés: Igaz, ha `Kif = rekord(`A_1`,...,` A_n`)` és `Lista = [rekord,`A_1`,...` A_n`]`. ∎

Megjegyzések

- Prologban nem megengedett: `Kif = Fun(X1, X2, ...,Xn)`
- Helyette használjuk: `Kif =..  [Fun, X1, X2, ...,Xn]`
- A C *vararg*-jához hasonló szerkezet: `Kif =..  [Fun|Args]`

Példák

```
| ?- el(a,b,10) =.. L.       L = [el,a,b,10] ?
| ?- el(a,b,10) =.. [F|As]. F = el, As = [a,b,10] ?
| ?- Kif =.. [/,1,2+3].      Kif = 1/(2+3) ?
| ?- [a,b,c] =.. L.          L = ['.',a,[b,c]] ?
```


Az univ eljárás építőelemei

```
functor(-Kif, +Nev, +ArgSzam)
functor(+Kif, ?Nev, ?ArgSzam)
```

Argumentumok:

`Kif` Az argumentum egy összetett kifejezés, név vagy szám.

`Nev` Az argumentum egy név vagy egy szám.

`ArgSzam` Az argumentum egy egész.

Jelentés: Igaz, ha `Kif` egy `Nev`/`ArgSzam` funktorú kifejezés.

Megjegyzések: Kétirányú eljárás. A számok és atomok 0-argumentumúnak számítanak. ∎

```
arg(+Sorszam, +Kif, ?Arg)
```

Argumentumok:

`Sorszam` Az argumentum egy egész.

`Kif` Az argumentum egy összetett kifejezés.

`Arg` Az argumentum egy tetszőleges kifejezés.

Jelentés: A `Kif` struktúra `Sorszam`-adik argumentuma `Arg`. ∎

Univ helyett `functor` és `arg`, példa

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


Példák: `functor/3` és `arg/3`

```
| ?- functor(szemely(kiss, pal, 1990), Nev, Aszam).
Nev = szemely, Aszam = 3 ?
yes
| ?- arg(2, szemely(kiss, pal, 1990), Arg).
Arg = pal ?
yes
| ?- functor(Str, szemely, 3).
Str = szemely(_A,_B,_C) ?
yes
| ?- functor(Str, szemely, 3), arg(1, Str, kiss),
     arg(2, Str, pal).
Str = szemely(kiss,pal,_A) ?
yes
| ?- functor([a,b,c], Nev, Aszam).
Nev = '.', Aszam = 2 ?
yes
| ?- arg(1, [a,b,c], A_1), arg(2, [a,b,c], A_2).
A_1 = a, A_2 = [b,c] ?
yes
| ?-
```

Univ: ismétlődő sémák összevonása

Kifejezések egyszerűsítése, univ nélkül

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

| ?- deriv((x+y)*(2+x), x, D), egysz0(D, ED).
        D = (1+0)*(2+x)+(x+y)*(0+1),
        ED = 1*(2+x)+(x+y)*1 ?
```

Kifejezések egyszerűsítése, univ segítségével

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


Univ: általános kifejezés-bejárás

Tetszőleges kifejezés kiiratása

- általában: alap-struktúra alakban, de
- a kétargumentumú operátorok infix formában

```
% Kif-et kiírja a fenti alakban
alapki(Kif) :-
        compound(Kif), !, Kif =.. [Func, A1|ArgL],
        (   current_op(_, Kind, Func),     % Func operátor?
            (Kind = xfy ; Kind = yfx ; Kind = xfx),
            ArgL = [A2] ->      % kétargumentumú használat?
            write('('), alapki(A1), write(' '),
            write(Func), write(' '), alapki(A2), write(')')
        ;   write(Func), write('('), alapki(A1),
            arglistaki(ArgL), write(')')
        ).
alapki(Kif) :- write(Kif).

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

| ?- alapki(1+2+3).           | ?- alapki(f(X,2,g(X))).
((1 + 2) + 3)                 f(_117,2,g(_117))

| ?- alapki([1,2]).           | ?- alapki(f(+a, b*c*d, e)).
.(1,.(2,[]))                  f(+(a),((b * c) * d),e)
```


Univ: általános kifejezés-bejárás

Kifejezés változómentesítése

- a kifejezés minden változóját behelyettesítjük pl. `'$myvar'(N)` alakú kifejezésekre;
- `numbervars/3` nevű változata SICStusban beépített eljárás, de `'$VAR'(N)` alakú kifejezéseket használ.
- `'$VAR'(0)`, `'$VAR'(1)`, ... `write`-tal való kíráskor `A`, `B` ... változónévként jelenik meg.

```
numbervars1(Term, N0, N) :- var(Term), !,
        Term = '$myvar'(N0), N is N0+1.
numbervars1(Term, N0, N) :-
        Term =.. [_|Args],
        number_list(Args, N0, N).

number_list([], N, N).
number_list([A|As], N0, N) :-
        numbervars1(A, N0, N1),
        number_list(As, N1, N).

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

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


`numbervars` egy alkalmazása

Két kifejezés azonossága

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

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

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

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

Konstansok szétszedése és összerakása 1.

`atom_codes(+Atom, ?KódLista)`
`atom_codes(-Atom, +KódLista)`

Argumentumok:

`Atom` Az argumentum egy atom.
`KódLista` Az argumentum karakterkódok listája.

Jelentés:

Igaz, ha `Atom` egyes karakterkódjainak a listája `KódLista`.

Megjegyzések:

Ha híváskor `Atom` ismert, és a $c_1c_2...c_n$ karakterekből áll, akkor a rendszer ezt szétszedi egy $[k_1, k_2, ..., k_n]$ számlistává, ahol k_i a c_i karakterkódja, és ezt egyesíti az eljárás második argumentumával. Ha `Atom` változó, akkor a `KódLista` karakterkód-listából összerak egy nevet, és azt írja be `Atom`-ba. ∎

```
| ?- atom_codes(ab, Cs).
  Cs = [97,98] ?

| ?- Cs = [0'b,0'c], atom_codes(Atom, Cs).
  Cs = [98,99], Atom = bc ?
```


Konstansok szétszedése és összerakása 2.

`number_codes(+Szám, ?KódLista)`
`number_codes(-Szám, +KódLista)`

Argumentumok:

`Szám` Az argumentum egy number.
`KódLista` Az argumentum karakterkódok listája.

Jelentés:

Igaz, ha `Szám` tizes számrendszerbeli alakjában az egyes karakterkódoknak a listája `KódLista`.

Megjegyzések:

Ha `Szám` tizes számrendszerbeli alakja a $c_1c_2...c_n$ karakterekből áll, akkor `KódLista = [`$k_1, k_2, ..., k_n$`]` lesz, ahol k_i a c_i karakterkódja. Ha `Szám` változó, akkor a `KódLista` karakterkód-listából összerak egy számot, és azt írja be `Szám`-ba. ∎

```
| ?- number_codes(1234, Cs).
  Cs = [49,50,51,52] ?

| ?- Cs = [0'1,0'2], number_codes(Num, Cs).
Cs = [49,50], Num = 12 ?
```


Konstansok szétszedése és összerakása — példák

```
| ?- use_module(library(lists)).

| ?- atom_codes(ab, _L), reverse(_L, _R),
     append(_L, _R, LR), atom_codes(X, LR).

  X = abba, LR = [97,98,98,97] ?

% Rész olyan részatomja Atom-nak, amelyet egy
% vele közvetlenül megegyező részatom követ.
dadogó_rész(Atom, Rész) :-
        atom_codes(Atom, Cs), dadogó(Cs, Ds),
        atom_codes(Rész, Ds).

% 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_rész(babaruhaha, R).

  R = ba ? ;
  R = ha ? ;
no
```


Kifejezések rendezése: szabványos sorrend

Legyen X és Y két tetszőleges Prolog kifejezés, ha X megelőzi Y-t, azt írjuk, hogy $X \prec Y$.

1. Ha X és Y azonos, akkor sem $X \prec Y$ sem $Y \prec X$ nem igaz és fordítva.
2. Ha X és Y típusa különböző, akkor a típus dönt: *változó* $\prec$ *lebegőpontos szám* $\prec$ *egész szám* $\prec$ *név* $\prec$ *struktúra*.
3. Ha X és Y változó, akkor az eredmény rendszerfüggő.
4. Ha X és Y lebegőpontos vagy egész szám, akkor $X \prec Y \Leftrightarrow X < Y$.
5. Ha X és Y név, akkor sorrendjük megegyezik az abc sorrenddel.
6. Ha X és Y struktúrák:
   (a) Ha X és Y aritása különböző, $X \prec Y \Leftrightarrow X$ aritása kisebb mint Y aritása.
   (b) Egyébként, ha a rekordok neve különböző, $X \prec Y \Leftrightarrow X$ neve $\prec Y$ neve.
   (c) Egyébként balról az első nem azonos argumentum dönt.

Végtelen (ciklikus) kifejezésekre a fenti rendezés nem érvényes.

Kifejezések összehasonlítása

`(==)/2`, `(\==)/2`, `(@<)/2`, `(@=<)/2`, `(@>)/2`, `(@>=)/2`

Jelentés:

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

Ezen eljárások minden argumentuma tisztán bemenő.

Argumentumok:

`Kif1` Az argumentum egy tetszőleges kifejezés.
`Kif2` Az argumentum egy tetszőleges kifejezés.

▪

Logikailag nem tiszta eljárások.

Rendezés a belső ábrázolás szerint:

```
| ?- [1, 2, 3, 4] @< struktúra(1, 2, 3).
```

sikerül (6a szabály).

@< egy megvalósítása

```
% T1 precedes T2 in standard order.
% Will order vars according to first occurrence.
precedes(T1, T2) :-
        \+ \+ (numbervars1(T1-T2, 0, _), prec(T1, T2)).

% T1 precedes T2 (both numbervar'd).
prec(T1, T2) :-
        class(T1, C1), class(T2, C2),
        (   C1 =:= C2 ->
            (   C1 =:= 1 -> T1 < T2
            ;   C1 =:= 2 -> T1 < T2
            ;   struct_prec(T1, T2)   % var case, too
            )
        ;   C1 < C2
        ).

% class(T, C): T belongs to class C.
class('$myvar'(_), C) :-   !, C = 0.
class(T, C) :- float(T),   !, C = 1.
class(T, C) :- integer(T), !, C = 2.
class(T, C) :- atom(T),    !, C = 3.
class(_T, 4).
```


@< egy megvalósítása, folytatás

```
% S1 precedes S2 (both structures or atoms).
struct_prec(S1, S2) :-
        functor(S1, F1, N1), functor(S2, F2, N2),
        (   N1 =:= N2 ->
            (   F1 = F2 ->
                args_prec(1, N1, S1, S2)
            ;   atom_prec(F1, F2)
            )
        ;   N1 < N2
        ).

% Args N0, ..., N of S1 precede those of S2.
args_prec(N0, N, S1, S2) :-
        N0 =< N, arg(N0, S1, A1), arg(N0, S2, A2),
        (   A1 = A2 -> N1 is N0+1,
            args_prec(N1, N, S1, S2)
        ;   prec(A1, A2)
        ).

% Atom A1 precedes A2 (precondition: A1 \= A2).
atom_prec(A1, A2) :-
        atom_codes(A1, C1), atom_codes(A2, C2),
        struct_prec(C1, C2).
```


Magasabbrendű eljárások

Magasabbrendű eljárás

- ha eljárásként értelmezi valamelyik argumentumát
- szokás meta-eljárásnak (meta predicate) is nevezni
- pl. `findall`, `bagof`, `setof`

Listaműveletek `findall` segítségével

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

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


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

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

Listakezelő meta-eljárások

Meta-eljárás deklarálása

- csak többmodulos program esetén lényeges;
- a : jelzi a hívás argumentumot;
- meta-argumentum meghívása: a `call(Cél)` beépített eljárással: a `Cél` (atom vagy struktúra) kifejezést meghívja;
- hívásként változó ekvivalens `call`-lal.

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

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

% Az L lista X elemeit Pred Y-ba képezi le.
% A kapott Y értékek listája ML.
:- meta_predicate map(+, ?, :, ?, -).
map(L, X, Pred, Y, ML) :-
        findall(Y, (member(X, L), Pred), ML).

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


Rekurzív meta-eljárások

Részlegesen paraméterezett eljárások

- segédeszközök: `call1/2`, `call2/3`, ... eljárások
- a `call<I>` eljárások sok Prolog megvalósításban beépítettek, de SICStusban nem

`call<I>` eljárások definíciói

```
% Pred az A utolsó argumentummal meghívva igaz.
call1(Pred, A) :-
    Pred =.. FArgs, append(FArgs, [A], FArgs1),
    Pred1 =.. FArgs1, call(Pred1).

% Pred az A és B utolsó argumentumokkal
% meghívva igaz.
call2(Pred, A, B) :-
    Pred =.. FArgs, append(FArgs, [A,B], FArgs2),
    Pred2 =.. FArgs2, call(Pred2).

% Pred az A, B és C utolsó argumentumokkal
% meghívva igaz.
call3(Pred, A, B, C) :-
    Pred =.. FArgs,
    append(FArgs, [A,B,C], FArgs3),
    Pred3 =.. FArgs3, call(Pred3).
```


Rekurzív meta-eljárások, folyt.

`map` rekurzív definíciója

```
% map(Xs, Pred, Ys): Az Xs lista elemeire
% a Pred transzformációt
% alkalmazva kapjuk az Ys listát.
map([X|Xs], Pred, [Y|Ys]) :-
        call2(Pred, X, Y),
        map(Xs, Pred, Ys).
map([], _, []).

negyzet(X, Y) :- Y is X*X.

| ?- map([1,2,3,4], negyzet, L).
L = [1,4,9,16] ? ;
no
```


Rekurzív meta-eljárások, folyt.

`foldl` és `foldr` definíciója

```
% foldl(Xs, Pred, Y0, Y): Az Xs elemeire balról
% jobbra alkalmazott, a Pred által leírt
% kétargumentumú függvény Y0 kezdőértékre
% alkalmazott eredménye Y.
foldl([X|Xs], Pred, Y0, Y) :-
        call3(Pred, X, Y0, Y1),
        foldl(Xs, Pred, Y1, Y).
foldl([], _, Y, Y).

% foldr(Xs, Pred, Y0, Y): Az Xs elemeire jobbról
% balra alkalmazott, a Pred által leírt függvény
% Y0 kezdőértékre alkalmazott eredménye Y.
foldr([X|Xs], Pred, Y0, Y) :-
        foldr(Xs, Pred, Y0, Y1),
        call3(Pred, X, Y1, Y).
foldr([], _, Y, Y).

jegyhozzá(A, J, E0, E) :- E is E0*A+J.

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

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

Modulok SICStus Prologban

Predikátumok és modulok

- minden predikátum valamelyik modulba tartozik,
- alapértelmezés: a `user` modul,
- Modul-állomány: olyan, amelynek első direktívája:
  `:- module(`*név*`, [`*exportált eljárás*`,...]).`
- Példa: `:- module(plató, [fennsík/3]).`

Modulok importálása

- Formátum:
  `:- use_module(`*file*`).`
  `:- use_module(`*file*`, [`*importált eljárás*`, ...]).`
- Példák:
  `:- use_module('plato.pl').`
  `:- use_module(library(lists), [last/2]).`

Modul-kvalifikálás (Module qualification)

- explicit modulnév megadás: *Modul*`:`*Hívás*
- Nincs szigorú név-takarás, pl. meghívható:
  `plató:első_fennsík([2,2,1], 4, F, H, L).`

Modulok és meta-predikátumok

Modulnév-kiterjesztés (module name expansion)

- a meta-argumentumok automatikus kvalifikálása
- meta-argumentum: hívásként használandó kifejezés
- Példa: `findall` stb. második argumentuma.

```
| ?- listing(páros_elemei).

páros_elemei(A, B) :-
    findall(C, user:(member(C,A),C mod 2=:=0), B).
```

Felhasználói meta-predikátumok

- a meta-argumentum(ok) kijelölése:
  `:- meta_predicate` *eljárás*`(`*spec*$_1$`, ...).`
- *spec*$_i$ lehet kettőspont (`:`) — meta-argumentum, vagy bármi más (általában módjelzés) — nem meta-argumentum.
- a meta-argumentum kezelésekor a modul-kvalifikálást is figyelembe kell venni.

```
:- meta_predicate call1(:, +).
call1(M:Pred, A) :-
    Pred =.. FArgs, append(FArgs, [A], FArgs1),
    Pred1 =.. FArgs1, call(M:Pred1).
```


Meta-predikátumok — példa

A `mylib.pl` állomány tartalma

```
:- module(my_library, [filter/4]).
:- use_module(library(lists)).

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

Példafutás

```
| ?- [mylib].
{module my_library imported into user}
{module lists imported into my_library}
{consulted mylib.pl in module my_library, 20 msec 2552 bytes}
yes
| ?- [user].
| jo(X) :- X > 0, X mod 2 =:= 0.
| {consulted user in module user, 0 msec 264 bytes}
yes
| ?- trace, filter([0,1,2,3,4], X, jo(X), L).
{The debugger will first creep -- showing everything (trace)}
   1   1 Call: filter([0,1,2,3,4],_187,user:jo(_187),_244) ?
   2   2 Call: findall(_187,
                  my_library:(member(_187,[0,1,2,3,4]),
                              user:jo(_187)),  _244) ?  n
L = [2,4] ?
```


Dinamikus adatbáziskezelés

Dinamikus eljárások

- futási időben hozzáadhatunk, és elvehetünk klózokat
- általában lassabak, mint a statikus eljárások
- deklaráció: `:- dynamic(Eljárásnév/Argumentumszám)`

`asserta(:@Klóz)`

Hatás: A `Klóz` kifejezést klózként értelmezve felveszi a programba, mégpedig az adott predikátum első klózaként. ∎

`assertz(:@Klóz)`

Hatás: A `Klóz` kifejezést klózként értelmezve felveszi a programba, mégpedig az adott predikátum utolsó klózaként. ∎

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

Dinamikus adatbáziskezelés

`retract(:@Klóz)`

Argumentumok:

`Klóz` Egy klózként értelmezhető kifejezés.

Jelentés: A `retract(Klóz)` hívás igaz, ha létezik egy dinamikus eljárás, amelynek van `Klóz`-zal egyesíthető klóza.

Hatás: Illeszti `Klóz`-zal az első megfelelő klózt az adott definícióból majd kitörli a klózt. *Többszörösen sikerülhet!* Visszalépéskor újabb klózt keres, illeszti, majd kitörli stb. ∎

```
| ?- retract((p(2,_):-_)), listing(p),
     write(----------), nl, fail.
p(1, A) :-
        q(A).
p(2, A) :-
        r(A).
----------
p(1, A) :-
        q(A).
----------
no

% retractall(Fej): kitörli az összes klózt,
% amelynek feje illeszthető Fej-jel.
retractall(Head) :- retract((Head :- _)), fail.
retractall(_).
```


Dinamikus predikátumok — példa

Egyszerű `findall`

- skatulyázva nem működik

```
:- dynamic(solution/1).

findall1(Templ, Goal, _Sols) :-
        call(Goal),
        asserta(solution(Templ)),  fail.
findall1(_Templ, _Goal, Sols) :-
        collect_sols([], Sols).

% A solution tényállítások argumentumában tárolt
% kifejezések megfordított listája L-L0.
collect_sols(L0, L) :-
        retract(solution(S)), !,
        collect_sols([S|L0], L).
collect_sols(L, L).

| ?- findall1(Y, (member(X, [1,2,3]), Y is X*X), S).
                        S = [1,4,9] ? ;
no
```


Dinamikus klózok elővétele

`clause(:+Fej, ?Törzs)`

Argumentumok:

`Fej` Az argumentum meghívható kifejezés.
`Törzs` Az argumentum meghívható kifejezés.

Jelentés: Igaz, ha létezik egy interpretált (F:-T) klóz, amely egyesíthető a (`Fej`:-`Törzs`) struktúrával.

Hatás: A (`Fej :- Törzs`) klózzal illeszthető első klózt megkeresi és illeszti. *Többszörösen sikerülhet!* Visszalépéskor újabb klózt keres, illeszti stb. ∎

Egy egyszerű nyomkövető interpreter

```
interp(true, _) :- !.
interp((G1, G2), D) :- !,
        interp(G1, D), interp(G2, D).
interp(G, D) :-
        (   trace(G, D, call)
        ;   trace(G, D, fail), fail
        ),
        D2 is D+2,
        clause(G, B), interp(B, D2),
        (   trace(G, D, exit)
        ;   trace(G, D, redo), fail
        ).

trace(G, D, Port) :-
        tab(D), write(Port), write(': '), write(G), nl.
```


Interpreter - példafutás

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

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

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

| ?- interp(app(_, [b,c], L, [c,b,c,b]), 0).
call: app(_117,[b,c],_167,[c,b,c,b])
  call: app(_117,_572,[c,b,c,b])
  exit: app([],[c,b,c,b],[c,b,c,b])
  call: app([b,c],_167,[c,b,c,b])
  fail: app([b,c],_167,[c,b,c,b])
  redo: app([],[c,b,c,b],[c,b,c,b])
    call: app(_779,_572,[b,c,b])
    exit: app([],[b,c,b],[b,c,b])
  exit: app([c],[b,c,b],[c,b,c,b])
  call: app([b,c],_167,[b,c,b])
    call: app([c],_167,[c,b])
      call: app([],_167,[b])
      exit: app([],[b],[b])
    exit: app([c],[b],[c,b])
  exit: app([b,c],[b],[b,c,b])
exit: app([c],[b,c],[b],[c,b,c,b])

L = [b] ?
```

A legfontosabb beépített eljárások

- **Vezérlési eljárások**
- **Aritmetika**
- **Kifejezések osztályozása és összehasonlítása**
- **Kifejezések egyesítése**
- **Listakezelés**
- **Kifejezések szétszedése és összereakása**
- **Dinamikus adatbáziskezelés**
- **Összes megoldás keresése**
- **Hibakezelés (kivételkezelés)**
- **Kiírás és beolvasás**
- **Bevitel/kiírás szervezése**
- **Programfejlesztés**

A predikátumleírások formátuma

A predikátumleírás részei

Hívási minta

Jelentés: mikor igaz?

Hatás: mi történik, ha meghívjuk?

Kompatibilitás: ISO vagy más?

Megjegyzések: megjegyzések ▪

Jelek a hívási mintákban

`@` tisztán bemenő paraméter

`+` bemenő paraméter

`?` kimenő/bemenő paraméter

`-` tisztán kimenő paraméter

`:`*X* ahol *X* a fenti karakterek közül való. A predikátum egy meta-predikátum és az adott argumentum modul-kiterjesztésen esik át.

Vezérlési eljárások

`!` (vágó)

Hatás: Megszünteti az összes választási pontot egészen a szülő célig, azt is beleértve. ▪

`Első, Második`

Jelentés:

Igaz, ha `Első` és `Második` is igaz. ▪

`(Első; Második)`

Jelentés:

Igaz, ha `Első` vagy `Második` igaz. ▪

`(Ha -> Akkor)` (if–then)

Hatás: A rendszer megkeresi a `Ha` első megoldását. Ha ez sikerül, eldobja `Ha` többi megoldását és meghívja `Akkor`-t, egyébként meghiúsul. ▪

`(Ha -> Akkor; Egyébként)` (if–then–else)

Hatás: A rendszer először célként meghívja `Ha`-t. Ha az sikerül, eldobja `Ha` többi megoldását és meghívja `Akkor`-t, egyébként meghívja `Egyébként`-et. ▪

A fenti eljárások átlátszóak a vágó számára.

`call(:+X)` (`X` változó célként)

Hatás: A rendszer `X`-et célként meghívja és annak sikere dönti el az eljárás sikerességét. ▪

Vezérlési eljárások

`fail`

Jelentés: Hamis. ▪

`true`

Jelentés: Igaz. ▪

`\+ :+Cél` (nem bizonyítható)

Jelentés: `\+ Cél` igaz, ha Cél nem igaz.

Hatás: Meghívja `Cél`-t és ha az meghiúsul, akkor sikerül, egyébként meghiúsul.

Megjegyzések: A végrehajtási módszerből adódóan siker esetén sem helyettesít be változókat. ▪

`repeat`

Jelentés: Igaz.

Hatás: Híváskor választási pontot hoz létre, majd sikerül. Ilyen módon visszalépések során végtelen sokszor képes sikerülni. ▪

Definíciója:

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

Ezt az eljárást mindig vágóval párban használjuk, pl.:

```
fociklus:-
        repeat,
        read(X),
        feldolgoz(X),
        X = end_of_file, !.
```

Aritmetikai kifejezések

Aritmetikai kifejezésekben felhasználható funktorok:

Infix operátorok

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

Prefix operátorok

- `-` negáció
- `\` bitenkénti negáció

Függvény jelölésűek

`abs/1`	abszolút érték
`ceiling/1`	felső egészrész
`sign/1`	előjel függvény
`sin/1`	szinusz
`float_integer_part/1`	lebegőpontos egészrész
`cos/1`	koszinusz
`float_fractional_part/1`	lebegőpontos törtrész
`float/1`	lebegőpontos konverzió
`atan/1`	arkusz tangens
`floor/1`	alsó egészrész
`exp/1`	exponenciális függvény
`truncate/1`	csonkolás
`log/1`	természetes alapú logaritmus
`round/1`	kerekítés
`sqrt/1`	négyzetgyök
`max/2`	maximum[1]
`min/2`	minimum[1]

[1] SICStus Prolog kiterjesztés

Aritmetikai eljárások

`?X is @AKif`

Argumentumok:

`X` Az argumentum egy tetszőleges kifejezés.
`AKif` Az argumentum egy aritmetikai kifejezés.

Jelentés:

Igaz, ha `X` az `AKif` aritmetikai kifejezés értéke. ∎

`(<)/2`, `(>)/2`, `(=<)/2`, `(>=)/2`, `(=:=)/2`, `(=\=)/2`

Jelentés:

Az eljárások szemantikáját az alábbi táblázat definiálja.

hívás	igaz, ha
`AKif1 < AKif2`	val(`AKif1`) $<$ val(`AKif2`)
`AKif1 > AKif2`	val(`AKif1`) $>$ val(`AKif2`)
`AKif1 =< AKif2`	val(`AKif1`) $\leq$ val(`AKif2`)
`AKif1 >= AKif2`	val(`AKif1`) $\geq$ val(`AKif2`)
`AKif1 =\= AKif2`	val(`AKif1`) $\neq$ val(`AKif2`)
`AKif1 =:= AKif2`	val(`AKif1`) $=$ val(`AKif2`)

Ezen eljárások minden argumentuma tisztán bemenő.

Argumentumok:

`AKif1` Az argumentum egy aritmetikai kifejezés.
`AKif2` Az argumentum egy aritmetikai kifejezés.

∎

Kifejezések osztályozása és összehasonlítása

`var/1`, `nonvar/1`, `integer/1`, `float/1`, `number/1`, `atom/1`, `atomic/1`, `compound/1`

Jelentés:

hívás	sikerül, ha `X`	hívás	sikerül, ha `X`
`var(X)`	változó	`nonvar(X)`	nem változó
`integer(X)`	egész	`atom(X)`	név
`float(X)`	valós	`atomic(X)`	konstans
`number(X)`	szám	`compound(X)`	összetett

∎

`(==)/2`, `(\==)/2`, `(@<)/2`, `(@=<)/2`, `(@>)/2`, `(@>=)/2`

Jelentés:

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

Ezen eljárások minden argumentuma tetszőleges kifejezés, tisztán bemenő. ∎

Kifejezések egyesítése

`?Kif1 = ?Kif2` — egyesítés

Jelentés: Igaz, ha `Kif1` és `Kif2` egyesíthető.

Megjegyzések:

Definíciója: `X = X.` ∎

`@Kif1 \= @Kif2` — egyesíthetetlenség

Jelentés: Igaz, ha `Kif1` és `Kif2` nem egyesíthető.

Megjegyzések:

Definíciója: `X \= Y :- \+ X = Y.` ∎

`unfify_with_occurs_check(?Kif1, ?Kif2)`

Jelentés: Igaz, ha `Kif1` és `Kif2` előfordulás-ellenőrzéssel egyesíthető. ∎

Példák

```
| ?- X+a = b+Y.                    X = b, Y = a ?
| ?- X+a = Y+b.                    no
| ?- X = 1.                        X = 1 ?
| ?- X \= 1.                       no
| ?- X+a \= Y+b.                   yes
| ?- X = f(X).                     X = f(f(f(...))) ?
| ?- unify_with_occurs_check(
         X, f(X)).                 no
```

Variációk az egyenlőségre

(Az alábbi példákban `X` egy behelyettesítetlen változó.)

A Prolog egyenlőség-szerű beépített eljárásai

- *U* `=` *V*: *U* egyesítendő *V*-vel. Soha sem jelez hibát. Pl. `X = 1+2` eredménye az `X = 1+2` behelyettesítés.
- *U* `==` *V*: *U* azonos *V*-vel. Soha sem jelez hibát és soha sem helyettesít be. Pl. `X == 1+2` meghiúsul.
- *U* `is` *V*: *U* egyesítendő a *V* aritmetikai kifejezés értékével. Hiba, ha *V* nem (tömör) aritmetikai kifejezés. Pl. `X is 1+2` eredménye az `X = 3` behelyettesítés.
- *U* `=:=` *V*: Az *U* és *V* aritmetikai kifejezések értéke megegyezik. Hibát jelez, ha *U* vagy *V* nem (tömör) aritmetikai kifejezés. Pl. `X =:= 1+2` hibát jelez.

A Prolog nemegyenlő-szerű beépített eljárásai

- Az alábbi eljárások egyike sem helyettesít be változót.
- *U* `\=` *V*: *U* nem egyesíthető *V*-vel. Soha sem jelez hibát. Pl. `X \= 1+2` meghiúsul.
- *U* `\==` *V*: *U* nem azonos *V*-vel. Soha sem jelez hibát. Pl. `X \== 1+2` sikerül.
- *U* `=\=` *V*: Az *U* és *V* aritmetikai kifejezések értéke különbözik. Hibát jelez, ha *U* vagy *V* nem (tömör) aritmetikai kifejezés. Pl. `X =\= 1+2` hibát jelez.

Példák az egyenlőség-szerű eljárások használatára

Jelmagyarázat

- igen ⇒ siker.
- nem ⇒ meghiúsulás.
- `\==` az `==` eljárásnak, `=\=` pedig az `=:=` eljárásnak a tagadása (igen ⇒ nem, nem ⇒ igen, hiba ⇒ hiba).
- `X` és `Y` behelyettesítetlen változók.

U	*V*	*U* `=` *V*	*U* `\=` *V*	*U* `==` *V*	*U* `is` *V*	*U* `=:=` *V*
1	2	nem	igen	nem	nem	nem
a	b	nem	igen	nem	hiba	hiba
1+2	+(1,2)	igen	nem	igen	nem	igen
1+2	2+1	nem	igen	nem	nem	igen
1+2	3	nem	igen	nem	nem	igen
3	1+2	nem	igen	nem	igen	igen
X	1+2	X=1+2	nem	nem	X=3	hiba
X	Y	X=Y	nem	nem	hiba	hiba
X	X	igen	nem	igen	hiba	hiba

Listakezelés

`length(?L, +N)`
`length(@L, -N)`

Argumentumok:

- `L` Az argumentum egy tetszőleges kifejezés.
- `N` Az argumentum egy egész szám.

Jelentés: Igaz, ha `L` lista hossza `N`.

Megjegyzések: Hajlandó adott hosszúságú, csupa különböző változóból álló listát létrehozni.

Kompatibilitás: SICStus Prolog kiterjesztés. ▪

`sort(@L, ?S)`

Argumentumok:

- `L` Az argumentum tetszőleges kifejezések listája.
- `S` Az argumentum egy tetszőleges kifejezés.

Jelentés: Igaz, ha `L` lista `@<` szerinti rendezése `S`, (`==/2` szerint azonos elemek ismétlődését kiszűrve).

Kompatibilitás: SICStus Prolog kiterjesztés. ▪

`keysort(@L, ?S)`

Argumentumok:

- `L` Az argumentum `(-)/2` funktorú kifejezések listája.
- `S` Az argumentum egy tetszőleges kifejezés.

Jelentés: Igaz, ha az `L` lista `Key-Value` párokból áll és `S` az `L` lista `Key` értékei szerinti szabványos rendezése. ▪

Kifejezések szétszedése és összerakása

`functor(-Kif, +Nev, +ArgSzam)`
`functor(+Kif, ?Nev, ?ArgSzam)`

Jelentés: Igaz, ha `Kif` egy `Nev`/`ArgSzam` funktorú kifejezés. ▪

`arg(+Sorszam, +Kif, ?Arg)`

Jelentés: A `Kif` struktúra `Sorszam`-adik argumentuma `Arg`. ▪

`+Kif =.. ?Lista`
`-Kif =.. +Lista`

Jelentés: Igaz, ha `Kif = rekord(`A_1`,...,` A_n`)` és `Lista = [rekord,`A_1`,...` A_n`]`. ▪

`atom_codes(+Atom, ?KódLista)`
`atom_codes(-Atom, +KódLista)`

Jelentés:

Igaz, ha `Atom` egyes karakterkódjainak a listája `KódLista`. ▪

`number_codes(+Szám, ?KódLista)`
`number_codes(-Szám, +KódLista)`

Jelentés:

Igaz, ha `Szám` tizes számrendszerbeli alakjában az egyes karakterkódoknak a listája `KódLista`. ▪

Dinamikus adatbáziskezelés

`asserta(:@Klóz)`

Hatás: A `Klóz` kifejezést klózként értelmezve felveszi a programba, mégpedig az adott predikátum első klózaként. ∎

`assertz(:@Klóz)`

Hatás: A `Klóz` kifejezést klózként értelmezve felveszi a programba, mégpedig az adott predikátum utolsó klózaként. ∎

`retract(:@Klóz)`

Hatás: Illeszti `Klóz`-zal az első megfelelő klózt az adott definícióból majd kitörli a klózt. *Többszörösen sikerülhet!* Visszalépéskor újabb klózt keres, illeszti, majd kitörli stb. ∎

`clause(:+Fej, ?Törzs)`

Jelentés: Igaz, ha létezik egy dinamikus (F:-T) klóz, amely egyesíthető a (`Fej`:-`Törzs`) struktúrával.

Megjegyzések: *Többszörösen sikerülhet!* Visszalépéskor újabb klózt keres, illeszti stb. ∎

Összes megoldás keresése

`findall(?Gyűjtő, :+Cél, ?Lista)`

Jelentés: A `Cél` összes megoldására `Gyűjtő` értéke listába gyűjtve `Lista`. ∎

`bagof(?Gyűjtő, :+Cél, ?Lista)`

Jelentés: `Lista` az összes olyan `Gyűjtő` behelyettesítés nem üres listája, amely a `Cél` egy megoldását adja. ∎

`setof(?Gyűjtő, :+Cél, ?Lista)`

Jelentés: Ugyanaz mint `bagof`, de `Lista` rendezett (ld. `sort/2`), ismétlődések nélkül. ∎

Hibakezelés

Hibakezelés

- Kapd el és dobd (Catch and throw)

  A rendszer visszalép (felgöngyölíti a vermeit) ameddig egy megfelelő hibakezelő eljáráshívásig nem ér.

`throw(@HibaKif)`
`raise_exception(@HibaKif)`

Jelentés: Se nem igaz, se nem hamis.

Hatás: Kiváltja a `HibaKif` hibahelyzetet.

Kompatibilitás: A két eljárás szinonima, a `raise_exception/1` SICStus Prolog kiterjesztés. ∎

Hibakezelés

`catch(:+Cél, ?Minta, :+Hibaág)`
`on_exception(?Minta, :+Cél, :+Hibaág)`

Argumentumok:

`Cél` Az argumentum egy meghívható kifejezés,
`Minta` Az argumentum egy tetszőleges kifejezés,
`Hibaág` Az argumentum egy meghívható kifejezés,

Jelentés:

Igaz, ha `call(Cél)` igaz, vagy ha a hívását megszakította `throw/1` (vagy `raise_exception/1`) hívása egy olyan argumentummal, ami egyesíthető `Minta`-val és `call(Hibaág)` igaz.

Kompatibilitás: `on_exception/3` SICStus Prolog kiterjesztés. ∎

```
% safe_is(X, Expr): X az Expr kifejezés értéke,
% vagy 0, ha Expr nem érvényes aritmetikai kifejezés.
safe_is(X, Expr) :-
        catch(X is Expr, Err, (write(Err), nl, X = 0)).

| ?- safe_is(X, Expr).
instantiation_error(_250 is _251,2)

X = 0 ?

yes
```

Kifejezések kiírása

`write(@X)`

Hatás: Kiírja `X`-et, ha szükséges operátorokat, zárójeleket használva. ∎

`writeq(@X)`

Hatás: Mint `write(X)`, csak gondoskodik, hogy szükség esetén az atomok idézőjelek közé legyenek téve. ∎

`write_canonical(@X)`

Hatás: Mint `writeq(X)`, csak operátorok nélkül, minden struktúra szabványos alakban jelenik meg. ∎

Argumentumok:

`X` Az argumentum egy tetszőleges kifejezés.

Példák

```
| ?- write('Helló világ').              Helló világ

| ?- writeq('Helló világ').             'Helló világ'

| ?- write('*' - '%').                  * -%

| ?- write_canonical('*' - '%').        -(*,'%')

| ?- write_canonical([1,2]).            '.'(1,'.'(2,[]))
```


Kifejezések kiírása

`print(@X)`

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

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

`portray(@Kif)`

Jelentés: Igaz, ha `Kif` kifejezést a Prolog rendszernek nem kell kiírnia.

Hatás: Alkalmas formában kiírja a `Kif` kifejezést.

Megjegyzések: Ez egy felhasználó által definiálandó (*kampó*) eljárás (hook predicate). ∎

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

| ?- X = [[1,2,3],[4,5,6]].
X =
[1,2,3]
[4,5,6] ?
```


Formázott kifejezés-kiírás

`format(@Formátum, @AdatLista)`

Argumentumok:

`Formátum` egy név vagy karakterkódok listája.
`AdatLista` tetszőleges kifejezések listája.

Hatás:

A `Formátum`-nak megfelelő módon kiírja `AdatLista`-t.
A formázójelek alakja: *~<szám esetleg> < formázójel>*.
∎

A legfontosabb formázójelek

	Adattal		*Adat nélkül*
`d`	(decimális) egész szám	`t`	tabuláció
`D`	(decimális) szám, csoportosítva	`n`	újsor
`f`	lebegőpontos szám	`|`	abszolút tabulátorpozició
`w`	tetszőleges kifejezés, mint write	`+`	relativ tabulátorpozició
`q`	tetszőleges kifejezés, mint writeq		
`p`	tetszőleges kifejezés, mint print		

```
time(Text, Goal):-
          statistics(runtime, [T0,_]),  call(Goal), !,
          statistics(runtime, [T1,_]), T is T1 - T0,
          format('~w:~t ~3d ~30|seconds. ~n', [Text,T]).

| ?- time('Fibci 200', fibci(200, _)),
     time('Fibci 4000', fibci(4000, _)).
Fibci 200:              0.040 seconds.
Fibci 4000:            15.360 seconds.
```


Karakterek kiírása

`put_code(@Kód)`

Argumentumok:

`Kód` Az argumentum egy karakterkód.

Hatás: Kiírja az adott kódú karaktert. ∎

`tab(@N)`

Argumentumok:

`N` Az argumentum egy egész szám.

Hatás: Kiír `N` szóközt feltéve, hogy $N > 0$.

Kompatibilitás: SICStus Prolog kiterjesztés. ∎

`nl`

Hatás: Kiír egy soremelést. ∎

Példa

```
| ?- nl, tab(10), put_code(0'a),
     tab(3), put_code(49), nl.

          a   1
```

Kifejezések beolvasása

`read(?Kif)`

Argumentumok:

`Kif` Az argumentum egy tetszőleges kifejezés.

Hatás:

Beolvas egy ponttal lezárt kifejezést és egyesíti `Kif`-fel.

Megjegyzések:

File végénél `Kif = end_of_file`. ∎

```
consult_body :-
        repeat, read(Term),
        (   Term == end_of_file -> true
        ;   assertz(Term), fail
        ), !.

| ?- consult_body.
|: pp(X) :- qq(X), rr(X).
|: ^D
yes
| ?- listing([pp/1]).
p(A) :-
        qq(A),
        rr(A).
yes
```


Karakterek beolvasása

`get_code(?Kód)`

Jelentés: Igaz, ha a beolvasott karakterkód `Kód`.

Hatás: Beolvas egy karaktert, és (karakterkódját) egyesíti `Kód`-dal.

Megjegyzések: File végénél `Kód = -1`. ∎

`peek_code(?Kód)`

Argumentumok:

`Kód` Az argumentum egy karakterkód.

Jelentés: Igaz, ha a beolvasható karakterkód `Kód`.

Hatás: Megnézi a soronkövetkező karaktert, és karakterkódját egyesíti `Kód`-dal. A karaktert nem távolítja el a bemenetről.

Megjegyzések: File végénél `Kód = -1`. ∎

Példa: számbeolvasás

```
% szambe(Szam): a Szam szám következik az
% input-folyamban.
szambe(Szam) :-
        szamjegy(Ertek), szambe(Ertek, Szam).

% Az eddig beolvasott Szam0-val együtt az
% input-folyamban következő szám értéke Szam.
szambe(Szam0, Szam) :-
        szamjegy(E), !, Szam1 is Szam0*10+E,
        szambe(Szam1, Szam).
szambe(Szam, Szam).

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

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


Bevitel/kiírás szervezése

Csatorna

- bármi, amiből olvasni vagy amibe írni lehet
- minden időpontban egy aktuális kimenet/bemenet
- eddig ezekre irtunk, ezekről olvastunk

Csatorna megnyitása

`open(@Filenév, @Mód, -Csatorna)`

Argumentumok:

`Filenév` Az argumentum egy atom, ami egy fájl nevét adja.

`Mód` Az argumentum a `read`, `write`, `append` atomok valamelyike.

`Csatorna` Az argumentum egy csatorna.

Hatás:

Megnyitja a `Filenév` nevű file-t `Mód` módban. A `Csatorna` argumentumban visszaadja a megnyitott csatorna nevét. ∎

Bevitel/kiírás szervezése

Az aktuális csatorna állítása/lekérdezése

```
set_input(@Csatorna)
set_output(@Csatorna)
```

Hatás: A `Csatorna` lesz a jelenlegi beviteli/kiviteli csatorna. ∎

```
current_input(?Csatorna)
current_output(?Csatorna)
```

Jelentés: Igaz, ha a jelenlegi beviteli/kiviteli csatorna `Csatorna`. ∎

Argumentumok:

`Csatorna` Az argumentum egy csatorna.

Csatorna lezárása

```
close(@Csatorna)
```

Argumentumok:

`Csatorna` Az argumentum egy csatorna.

Hatás: Lezárja a `Csatorna` csatornát. ∎

Explicit csatornamegadás

```
write(@Csatorna, @Kif)
writeq(@Csatorna, @Kif)
write_canonical(@Csatorna, @Kif)
print(@Csatorna, @Kif)
format(@Csatorna, @Formátum, @AdatLista)
read(@Csatorna, ?Kif)
put_code(@Csatorna, @Kód)
tab(@Csatorna, @N)
nl(@Csatorna)
get_code(@Csatorna, ?Kód)
peek_code(@Csatorna, ?Kód)
```

Hatás:

Azonos az első argumentumok elhagyásával keletkező eljárásokéval, azzal az eltéréssel, hogy nem az aktuális ki/bemenetet használják, hanem az első argumentumban megadottat. ∎

DEC10 I/O

```
see(@Filenév)
tell(@Filenév)
```

Argumentumok:

`Filenév` Az argumentum egy atom.

Hatás: Első hívásakor megnyitja a `Filenév` file-t olvasásra/írásra és a jelenlegi beviteli/kiviteli csatornává teszi. Későbbi híváskor csak a jelenlegi csatornává teszi. ∎

```
seeing(?Filenév)
telling(Filenév)
```

Argumentumok:

`Filenév` Az argumentum egy tetszőleges kifejezés.

Jelentés: Igaz, ha `Filenév` a jelenlegi beviteli/kiviteli csatorna neve. ∎

```
seen
told
```

Hatás: Lezárja a jelenlegi beviteli/kiviteli csatornát, a terminál lesz a jelenlegi beviteli/kiviteli csatorna. ∎

Példa: Egyszerű consult variánsok

```
consult1(File) :-
        open(File, read, S),
        repeat, read(S, Term),
        (   Term == end_of_file -> close(S)
        ;   Term = (:- Goal) -> execute(Goal), fail
        ;   assertz(Term), fail
        ), !.

consult2(File) :-
        current_input(S),
        see(File),
        repeat, read(Term),
        (   Term == end_of_file -> seen
        ;   Term = (:- Goal) -> execute(Goal), fail
        ;   assertz(Term), fail
        ), !,
        set_input(S).

execute(Goal) :- call(Goal), !.
execute(_).
```

Az itt ismertetett eljárások nem részei a szabványnak, bár a legtöbb Prolog rendszerben megtalálhatók.

Jelzők

`set_prolog_flag(+Jelző, @Érték)`

Argumentumok:

`Jelző` Az argumentum egy Prolog jelző.
`Érték` Az argumentum egy a jelzőnek megfelelő kifejezés.

Hatás: `Jelző` értkékét `Érték`-re állítja. ∎

`current_prolog_flag(?Jelző, ?Érték)`

Argumentumok:

`Jelző` Az argumentum egy Prolog jelző.
`Érték` Az argumentum egy tetszőleges kifejezés.

Jelentés: Igaz, ha `Jelző` egy érvényes Prolog jelző és pillanatnyi értkéke `Érték`. ∎

A legfontosabb Prolog jelzők

- `language` Lehetséges értékei: `sicstus` (ez az alapértelmezés) vagy `iso`. Azt adja meg, hogy éppen milyen módban vagyunk.
- `source_info` Lehetséges értékei: `on`, `off`. Azt adja meg, hogy a rendszer gyűjtsön-e forrás szintű nyomkövetési információt (`on`) vagy se (`off`). Használata elsősorban a GNU Emacs környezetben kényelmes, ilyenkor a rendszer a forrásfile megfelelő sorát szép zölden kivilágítja és jelzi, hogy milyen kapunál járunk.

`consult(@Files)`
`[@File,...]`

Argumentumok:

`Files` Az argumentum egy név vagy nevek listája.
`File` Az argumentum egy név.

Hatás: Beolvassa a `File`(ok)at.

Megjegyzések: SICStusban interpretált alakot hoz létre.
∎

`compile(@Files)`

Argumentumok:

`Files` Az argumentum egy név vagy nevek listája.

Hatás: Beolvassa a `File`(ok)at, leforditott alakot hozva létre. ∎

`listing`
`listing(@EljárásSpec)`

Argumentumok:

`EljárásSpec` Az argumentum egy eljárás specifikáció.

Hatás: Kiírja az összes/megnevezett interpretált eljárás(oka)t az aktuális kimenetre. ∎

Eljárás specifikáció

- *név* Minden predikátum, aminek ez a neve, tetszőleges aritással.
- *név*/*aritás* A *név* nevű, *aritás* aritású eljárás.
- *név*/*alsó*-*felső* Minden *név* nevű eljárás aminek az aritása *alsó* és *felső* közé esik.
- *modul*:*spec* Minden *modul* modulbeli a *spec* eljárás specifikációnak megfelelő eljárás.
- $[spec_1,\ldots,spec_n]$ Minden a $spec_i$ specifikációk által meghatározott eljárás.

Statisztika

`statistics`

Hatás: Különféle statisztikákat ír ki az aktuális kimenetre. ∎

`statistics(?Fajta, ?Ertek)`

Argumentumok:

`Fajta` Az argumentum egy mennyiség neve.
`Ertek` Az argumentum egy tetszőleges kifejezés.

Jelentés: Igaz, ha `Ertek` a `Fajta` fajtájú mennyiség pillanatnyi értéke. ∎

Példa

`statistics(runtime, E)` eredménye `E=[Tdiff, T]`, ahol `Tdiff` az előző lekérdezés óta eltelt idő, `T` a rendszerindítás óta eltelt idő, mindkettő ezredmásodpercben.

`break`

Hatás: Egy új interakciós szintet hoz létre. ∎

`abort`

Hatás: Kilép a legkülső interakciós szintre. ∎

`halt`

Hatás: Kilép a Prolog rendszerből.

Kompatibilitás: ISO szabvány szerinti eljárás. ∎

`trace`

Hatás: Elindítja az interaktív nyomkövetést. ∎

`debug`
`zip`

Hatás: Elindítja a szelektív nyomkövetést (spion-pontok, lásd alább).

Megjegyzések: A két eljárás között annyi a különbség, hogy `zip` módban a rendszer gyorsabb (majdnem olyan gyors, mint ha nem is lenne nyomkövetés), de nem gyűjt annyi információt mint `debug` módban. ∎

`nodebug`
`notrace`
`nozip`

Hatás: Leállítja a nyomkövetést. ∎

`spy(@EljárásSpec)`

Hatás: Spion-pontot tesz az `EljárásSpec` által megadott eljárásokra. ∎

`nospy(@EljárásSpec)`

Hatás: Megszünteti az `EljárásSpec` által megadott eljárásokra kiadott Spion-pontokat. ∎

Argumentumok:

`EljárásSpec` Az argumentum egy eljárás specifikáció.

`nospyall`

Hatás: Az összes spion-pontot megszünteti. ∎

`leash(@Kapulista)`

Argumentumok:

`Kapulista` Az argumentum kapuk listája.

Hatás: A `Kapulista` meghatározza, hogy teljes nyomkövetéskor mely kapuknál álljon meg a rendszer.

Megjegyzések: A listában a következő nevek szerepelhetnek: `call`, `exit`, `redo`, `fail`, `exception`. Alapértelmezésben a rendszer minden kapunál megáll. ∎

Fejlettebb nyelvi és rendszerelemek

Nyelvtani elemzés

Külső nyelvi interfész

Hasznos beépített eljárások

Fejlett vezérlési lehetőségek

SICStus könyvtárak

Nyelvtani elemzés Prologban

Definit klóz nyelvtan (DCG — Definite Clause Grammar)

- egy általános nyelvtani formalizmus,
- egyszerűen Prologra fordítható,
- a legtöbb Prolog rendszer része (bár a szabványnak nem).

Példa — bináris számok

⟨szám⟩ ::= ⟨számjegy⟩ ⟨számmaradék⟩
⟨számmaradék⟩ ::= ⟨számjegy⟩ ⟨számmaradék⟩ | ϵ
⟨számjegy⟩ ::= 0 | 1

DCG (Definite Clause Grammar) jelöléssel

```
szám -->
        számjegy, számmaradék.

számmaradék -->
        számjegy, számmaradék.
számmaradék --> [].

számjegy --> [0'1].  számjegy --> [0'0].
% vagy:
számjegy --> "0" ; "1".   % "0" == [0'0]
```


Nyelvtani elemzés *(folytatás)*

A DCG szabályok lefordított alakja

```
| ?- [user].
| szám --> számjegy, számmaradék.

| számmaradék -->
        számjegy, számmaradék.
| számmaradék --> "".              % "" == []

| számjegy --> "0" | "1".
{consulted user in module user, 10 msec 912 bytes}

| ?- listing.
szám(A, B) :-
        számjegy(A, C), számmaradék(C, B).

számmaradék(A, B) :-
        számjegy(A, C), számmaradék(C, B).
számmaradék(A, B) :- B=A.

számjegy(A, B) :-   (   'C'(A, 48, B)
                    ;   'C'(A, 49, B)
                    ).

| ?- szám("101", []).    % "101" == [0'1,0'0,0'1]
yes
| ?- szám("102", []).
no
```


Nyelvtani elemzés *(folytatás)*

A lefordított alak értelmezése

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

% L0-ról leelemezhető egy <számmaradék>, marad L.
számmaradék(L0, L) :-
        számjegy(L0, L1), számmaradék(L1, L).
számmaradék(L, L).

% Leelemezhető egy <számjegy>.
számjegy(L0, L) :-   (   'C'(L0, 48, L)
                     ;   'C'(L0, 49, L)
                     ).
```

A `'C'/3` beépített eljárás

```
% L0-ból leelemezhető egy E terminális elem, marad L
'C'(L0, E, L) :- L0 = [E|L].
```

A SICStus fordító a `'C'(L0, E, L)` hívást pontosan ugyanúgy fordítja, mint az `L0 = [E|L]` hívást.

Nyelvtani elemzés *(folytatás)*

DCG-ben használható vezérlési szerkezetek

- vágó
- diszjunkció
- negáció
- feltételes diszjunktív szerkezet

Példák

```
% Leelemezhető számjegyek egy MAXIMÁLIS
% (esetleg üres) listája.
számmaradék -->
        számjegy, !, számmaradék.
számmaradék --> [].

% Ugyanez feltételes szerkezettel
számmaradék -->
        (   számjegy -> számmaradék.
        ;   []
        ).
```


Nyelvtani elemzés *(folytatás)*

Decimális számjegyek elemzése

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

% Ugyanez általánosabban és egyszerűbben:
számjegy --> [K], {decimális_kód(K)}.

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

Prolog megfelelő

```
% Leelemezhető egy számjegy kódja.
számjegy(L0, L) :- L0 = [K|L], decimális_kód(K).
```


Nyelvtani elemzés *(folytatás)*

Az elemző kiegészítése argumentumokkal

```
% leelemezhető az Sz decimális szám
szám(Sz) -->
        számjegy(J), számmaradék(J, Sz).

% leelemezhető számjegyek egy esetleg üres listája,
% amelynek decimális értékét Sz0 mögé rakva Sz-t kapjuk.
számmaradék(Sz0, Sz) -->
        számjegy(J), !, {Sz1 is Sz0*10+J},
        számmaradék(Sz1, Sz).
számmaradék(Sz0, Sz0) --> [].

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

| ?- szám(Sz, "1024 56", L).

        L = [32,53,54],  Sz = 1024 ? ; no

| ?- listing(számmaradék).

számmaradék(A, B, C, D) :-
        számjegy(E, C, F), !, G is A*10+E,
        számmaradék(G, B, F, D).
számmaradék(A, A, B, C) :-
        C=B.
```


DCG nyelvtanok

A DCG nyelvtani szabályok szerkezete

- Nem-terminális: tetszőleges *hívható* kifejezés (atom vagy struktúra).
- Terminális: *tetszőleges* Prolog kifejezés; a 0, 1 vagy több terminális sorozata listaként helyezhető el a DCG szabályokban.
- Feltételek: tetszőleges Prolog hívás elhelyezhető `{}` zárójelekbe zárva.
- A DCG szabály alakja: Baloldal `-->` Jobboldal .
- Baloldal: egy nem-terminális, amit opcionálisan terminálisok listája követhet.
- Jobboldal: Egymás után írás (`,`), diszjunkció (`;`), ha-akkor és ha-akkor-egyébként (`->`) és negáció (`\+`) segítségével épül fel terminálisokból és nem-terminálisokból.

Általános példa

`p(A,...)` `-->`
`q`$_0$`(B,...), ...[X], q`$_i$`(C,...), ..., {Cel}, ...q`$_n$`(D,...)`

lefordított alakja:

`p(A,...,`L_0`,L):-`
`q`$_0$`(B,...,`L_0`,`L_1`), ...`L_{i-1} `= [X|`L_i`], q`$_i$`(C,...,`L_i`,`L_{i+1}`),...`
`Cel, ..., q`$_n$`(D,...,`L_n`,L)`

Példa: kifejezés kiértékelése

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

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

| ?- kif(Z, "10*10-6*6", "").
                Z = 64 ? ;  no
| ?- kif(Z, "10*10-6*6", L).
                L = [], Z = 64 ? ;
                L = [42,54], Z = 94 ? ;
                L = [45,54,42,54], Z = 100 ? ;
                L = [42,49,48,45,54,42,54], Z = 10 ? ;
                no
% Vigyázat: a fenti nyelvtan jobbról balra zárójelez:
| ?- kif(Z, "4-2+1", []).
                Z = 1 ?

% Egy lehetséges javítás:
kif(Z) --> tag(X), kifmaradék(X, Z).

kifmaradék(X0, Z) -->
    "+", tag(X1), {X is X0 + X1}, kifmaradék(X, Z).
% ...
```


Példa: “természetes” nyelvű beszélgetés

```
:- use_module(library(lists)).
mondat(én, Áll) --> perm(én, Áll).
mondat(te, Áll) --> perm(te, Áll).
mondat(Alany, Áll) --> szó(Alany), szavak(Áll).

perm(Ki, Áll) --> névmás(Ki), létige(Ki), szavak(Áll).
perm(Ki, Áll) --> névmás(Ki), szavak(Áll), létige(Ki).
perm(Ki, Áll) --> szavak(Áll), létige(Ki), névmás(Ki).
perm(Ki, Áll) --> szavak(Áll), létige(Ki).

névmás(én) --> "én", köz.        névmás(te) --> "te", köz.
létige(én) --> "vagyok", köz.    létige(te) --> "vagy", köz.

köz --> " " -> köz ; [].

szó([B|Sz]) --> betű(B), szómaradék(Sz), köz.

szómaradék([B|Sz]) --> betű(B), !, szómaradék(Sz).
szómaradék([]) --> [].

betű(K) --> [K], {non_member(K, " .?")}.

szavak([Sz|Szk]) --> szó(Sz),
     (   szavak(Szk)
     ;   {Szk = []}
     ).
```


Beszélgetős DCG példa — folytatás

```
menet(kijelent(Alany,Áll)) -->
    mondat(Alany, Áll), ".".
menet(kérdez(Alany)) -->
    mondat(Alany, [Szó]), "?", {kérdő(Szó)}.
menet(un) --> "unlak.".

kérdő("mi").   kérdő("ki").   kérdő("kicsoda").

párbeszéd :-
     repeat, rd_line(L),
     (   menet(M, L, []) -> feldolgoz(M)
     ;   write('Nem értem.\n'), fail
     ), M = un, !.

feldolgoz(un) :- write('Én is.\n').
feldolgoz(kijelent(Alany, Áll)) :-
     assertz(tudom(Alany,Áll)), write('Felfogtam.\n').
feldolgoz(kérdez(Alany)) :-
     tudom(Alany, _), !, válasz(Alany).
feldolgoz(kérdez(_)) :- write('Nem tudom.\n').

válasz(Alany) :- tudom(Alany, Áll),
     (   member(Szó, Áll), format('~s ', [Szó]), fail
     ;   nl, fail
     ).
válasz(_).

% rd_line(L): L a következő sor karakterkódjainak listája.
% lásd a jegyzet 126. oldalán.
```


Beszélgetős DCG példa — egy párbeszéd

```
| ?- párbeszéd.
|: magyar legény vagyok én.
Felfogtam.
|: ki vagyok én?
magyar legény
|: Péter kicsoda?
Nem tudom.
|: Péter tanuló.
Felfogtam.
|: Péter jó tanuló.
Felfogtam.
|: Péter kicsoda?
tanuló
jó tanuló
|: boldog vagyok.
Felfogtam.
|: én vagyok Jeromos.
Felfogtam.
|: te egy Prolog program vagy.
Felfogtam.
|: ki vagyok én?
magyar legény
boldog
Jeromos
|: okos vagy.
Felfogtam.
|: te vagy a világ közepe.
Felfogtam.
|: ki vagy  te?
egy Prolog program
okos
a világ közepe
|: valóban?
Nem értem.
|: unlak.
Én is.
```

DCG használata elemzésen kívül

A DCG formalizmus kényelmesen használható

- listák akkumulálására
- tetszőleges kifejezések akkumulálására
- az utóbbi esetben az elemi akkumulálási lépést DCG-n kívül kell megírni

```
% M az L lista N-nél nagyobb elemeinek listája.
nagyobb(L, N, M) :- nagyobb(L, N, M, []).

% nagyobb(L, N, M, M0): M-M0 az L lista N-nél
% nagyobb elemeinek listája.
nagyobb([], _) --> [].
nagyobb([X|L], N) --> {X > N}, !, [X], nagyobb(L, N).
nagyobb([_|L], N) --> nagyobb(L, N).

% Az L lista összege S.
sum(L, S) :- sum1(L, 0, S).

% sum1(L, S0, S): Az L lista összege S-S0.
sum1([]) --> [].
sum1([X|L]) --> add(X), sum1(L).

% X+S0= S.
add(X, S0, S) :- S is S0+X.
```


Külső nyelvi interfész

Hagyományos (pl. C nyelvű) programrészek meghívása

- A Prolog rendszer elvégzi az átalakítást a Prolog alak és a külső nyelvi alak között. Kényelmesebb, biztonságosabb mint a másik módszer, de kevésbé hatékony. Többnyire csak egyszerű adatokra (egész, valós, atom). (MProlog)
- A külső nyelvi rutin pointereket kap Prolog adatstruktúrákra, valamint hozzáférési algoritmusokat ezek kezelésére. Nehézkesebb, veszélyesebb, de jóval hatékonyabb mint az előző megoldás. Összetett adatok adás-vételére is jó. (SWI, SICStus)

Külső nyelvi interfész példa

```
| ?- [ixtest].
| ?- index_keys(f(+, -, +, +),
                f(12.3, _, s(1, _, z(2)), t),
                L, X).
L = [12.3,s,3,t], X = 3 ?
yes
```

Az `ixtest.pl` file.

```
foreign(ixkeys, index_keys(+term, +term, -term, [-integer])).
foreign_resource(ixkeys, [ixkeys]).
:- load_foreign_resource(ixkeys).
```

A C programot elő kell készíteni a Prolog számára az `splfr` eszköz segítségével:

```
splfr ixkeys ixtest.pl +c ixkeys.c
```


Az `ixkeys.c` file.

```
#include <sicstus/sicstus.h>

#define NA -1                   /* not applicable */
#define NI -2                   /* instantiatedness */

long ixkeys(SP_term_ref spec, SP_term_ref term, SP_term_ref list)
{
  unsigned long sname, tname, plus;
  int sarity, tarity, i;
  long ret = 0;
  SP_term_ref arg = SP_new_term_ref(), tmp = SP_new_term_ref();

  SP_get_functor(spec, &sname, &sarity);
  SP_get_functor(term, &tname, &tarity);
  if (sname != tname || sarity != tarity) return NA;

  plus = SP_atom_from_string("+");
  for (i = sarity; i > 0; --i) {
    unsigned long t;

    SP_get_arg(i, spec, arg);
    SP_get_atom(arg, &t);       /* no error checking */
    if (t != plus) continue;

    SP_get_arg(i, term, arg);
    switch (SP_term_type(arg)) {
    case SP_TYPE_VARIABLE:
      return NI;
    case SP_TYPE_COMPOUND:
      SP_get_functor(arg, &tname, &tarity);
      SP_put_integer(tmp, (long)tarity);
      SP_cons_list(list, tmp, list);
      SP_put_atom(arg, tname);
      break;
    }
    SP_cons_list(list, arg, list);
    ++ret;
  }
  return ret;
}
```

Hasznos lehetőségek SICStus Prolog-ban

- Tetszőleges nagyságú egész számok
  pl.:

```
| ?- fakt(100,F).

F = 933262154439441526816992388562667004907159682643816214685929638952175999932299156089414639761565182862536979208272237582511852109168640000000000000000000000000 ?
```

- Globális változók (Blackboard)

  `bb_put(Kulcs, Érték)`

  A `Kulcs` kulcs alatt eltárolja `Érték`-et, az előző értéket, ha van, törölve. (`Kulcs` egy (kis) egész szám vagy atom lehet.)

  `bb_get(Kulcs, Érték)`

  Előhívja `Érték`-be a `Kulcs` értékét.

  `bb_delete(Kulcs, Érték)`

  Előhívja `Érték`-be a `Kulcs` értékét, majd kitörli.

Hasznos lehetőségek SICStus Prolog-ban *(folytatás)*

- Visszaléptethető módon változtatható kifejezések

  `create_mutable(Adat, ValtKif)`

  `Adat` kezdőértékkel létrehoz egy új változtatható kifejezést, ez lesz `ValtKif`. `Adat` nem lehet üres változó.

  `get_mutable(Adat, ValtKif)`

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

  `update_mutable(Adat, ValtKif)`

  A `ValtKif` változtatható kifejezés új értéke `Adat` lesz. Ez a változtatás visszalépéskor visszacsinálódik. `Adat` nem lehet üres változó.

- Takarító eljárás

  `call_cleanup(Hivas, Tiszito)`

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

Fejlett vezérlési lehetőségek SICStusban: Blokk-deklarációk

Példa:

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

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

Ugyanarra az eljárásra több vagylagos feltétel is szerepelhet, pl.

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

Végtelen választási pontok kiküszöbölése blokkdeklarációval

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

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


Blokk-deklarációk *(folytatás)*

Generál-és-ellenőriz típusú programok gyorsítása

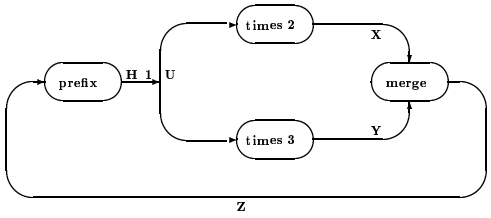
- általában nem hatékonyak (pl megrajzolja_1), mert túl sok visszalépést használnak
- korutinszervezéssel a generáló és ellenőrző rész "automatikusan" összefésülhető
- ehhez az ellenőrző részt kell előre tenni és megfelelően blokkolni

Korutinszervezésre épülő programok

Példa: egyszerűsített Hamming feladat

- keressük a $2^i * 3^j (i \geq 1, j \geq 1)$ alakú számok közül az első `N` darabot nagyság szerint rendezve.
- "stream-and-parallelism" közelítésmódot használva korutinszervezéssel egyszerűen lehet megoldani

Hamming probléma




```
% A H lista az első N, csak a 2 és 3 tényezőkből álló szám.
hamming(N, H) :-
     U = [1|H], times(U, 2, X), times(U, 3, Y),
     merge(X, Y, Z), prefix(N, Z, H).

% times(X, M, Z): A Z lista az X elemeinek M-szerese
:- block times(-, ?, ?).
times([A|X], M, Z) :-
     B is M*A, Z = [B|U], times(X, M, U).
times([], _, []).
```


Hamming probléma (folyt.)

```
% merge(X, Y, Z): Z az X és Y összefésülése.
:- block merge(-, ?, ?), merge(?, -, ?).
% Csak akkor fusson, ha az első két argumentum ismert
merge([A|X], [B|Y], V) :-
     A < B, !, V = [A|Z], merge(X, [B|Y], Z).
merge([A|X], [B|Y], V) :-
     B < A, !, V = [B|Z], merge([A|X], Y, Z).
merge([A|X], [A|Y], [A|Z]) :-
     merge(X, Y, Z).
merge([], X, X) :- !.
merge(_, [], []).

% prefix(N, X, Y): Az X lista első N eleme Y.
prefix(0, _, []) :- !.
prefix(N, [A|X], [A|Y]) :-
     N > 0, N1 is N-1, prefix(N1, X, Y).
```


Korutinszervező eljárások

`freeze(X, Hivas)`

`Hivas`t felfüggeszti mindaddig, amig `X` behelyettesítetlen változó.

`frozen(X, Hivas)`

Az `X` változó miatt felfüggesztett hívás(oka)t egyesíti `Hivas`-sal.

`dif(X, Y)`

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

`call_residue(Hivas, Maradék)`

`Hivas`-t végrehajtja, és ha a sikeres lefutás után maradnak felfüggesztett hívások, akkor azokat visszaadja `Maradékban`. Pl.

```
| ?- call_residue(dif(X, f(Y)), Maradek).

Maradek = [[X]-(prolog:dif(X,f(Y)))] ?

yes
| ?- call_residue((dif(X, f(Y)), X=f(Z)), Maradek).

X = f(Z),
Maradek = [[Y,Z]-(prolog:dif(f(Z),f(Y)))] ?

yes
```


SICStus könyvtárak

Könyvtár betöltése

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

Könyvtárak

- `arrays` Logaritmikus elérési idejű kiterjeszthető tömbök megvalósítását tartalmazza.
- `assoc` AVL fák segítségével valósítja meg az „asszociációs listák”, azaz véges Prolog kifejezéshalmazokon definiált kiterjeszthető leképezések fogalmát.
- `atts` tetszőleges attributumokat enged a Prolog változókhoz rendelni, ezeket tárolórekeszként és a Prolog egyesítési mechanizmusának módosítására is engedi használni.
- `heaps` A bináris kazal (heap) fogalmát valósítja meg, amely főként prioritásos sorok (priority queue) megvalósítására használható.
- `lists` Biztosítja a listakezelő alapműveleteket.
- `terms` Különböző kifejezéskezelő eljárásokat tartalmaz.
- `ordsets` Halmazműveleteket definiál, ahol a halmazokat a Prolog szabványos rendezése szerint (`compare`) rendezett listákkal ábrázolja.

- `queues` Sorokra (queue, FIFO store) vonatkozó műveleteket definiál.
- `random` Egy véletelenszám-generátort tartalmaz.
- `system` Különféle operációsrendszer-szolgáltatások elérését biztosítja.
- `trees` Az `arrays` könyvtárhoz hasonló, de nem-kiterjeszthető logaritmikus elérési idejű tömbfogalmat valósít meg, bináris fák segítségével (kicsit hatékonyabb mint az `arrays` könyvtár).
- `ugraphs` Irányított és irányítatlan gráffogalmat valósít meg, élcimkék nélkül.
- `wgraphs` Olyan irányított és irányítatlan gráffogalmat valósít meg, ahol minden él egy egészértékű súllyal rendelkezik.
- `sockets` A socket-ek kezelésére szolgáló eljárásokat biztosít.
- `linda/client` és `linda/server` Linda-szerű processzkommunikációs eszközöket ad.
- `db` Felhasználó által definiált többszörös indexelést lehetővé tevö, Prolog kifejezések lemezen való tárolására szolgáló adatbázis-rendszer.
- `clpb` Boole-értékekre vonatkozó feltétel-megoldó (constraint solver).
- `clpq` és `clpr` Feltétel-megoldó a Q (racionális számok) ill. R (valós számok) tartományán.

- `clpfd` Véges tartományokra vonatkozó feltétel-megoldó (constraint solver).
- `objects` A logikai és objektum-orientált paradigmák kombinációját biztosítja.
- `gcla` A Prolog ún. GCLA (Generalized Horn Clause Language) általánosításán alapuló specifikációs eszköz.
- `tcltk` A *Tcl/Tk* nyelv és eszközkészlet elérését biztosítja.
- `gauge` Prolog programok a profilirozására szolgáló, a `tcltk` -n alapuló grafikus interfésszel rendelkező eszköz.
- `charsio` Karaktersorozatból olvasó ill. abba író be- és kiviteli eljárások gyüjteménye.
- `flinkage` Segédprogram a külső nyelvi interfész összekötő kódjának generálására.
- `timeout` Lehetőséget ad arra, hogy célok futási idejét korlátozzuk.
- `xref` A nyomkövetés és a program-analízis segítésére használható keresztreferencia készítő program.

Fordítóprogramírás Prologban

David H. D. Warren

Logic Programming and Compiler Writing
Software Practice and experience, Vol 10, 97-125 (1980)
c. cikke alapján

A forrásnyelv és a célnyelv

A fordítóprogram szerkezete

Kódgenerálás

Becímzés

Nyelvtani elemzés

Lexikai elemzés

Kifrás

Forrásnyelv

egyszerű Algol-szerű nyelv

- értékadás
- IF utasítás
- WHILE utasítás
- READ és WRITE utasítások

Például a következő forrásprogram szolgálhat a faktoriális kiszámítására:

```
READ value;
count := 1;
result := 1;
WHILE count < value DO
      (count := count+1;
      result := result*count);
WRITE result
```

Célnyelv

egyszerű egycímes gépi nyelv

- Aritmetikai stb utasitások
  (literális operandussal):

  `ADDC, SUBC, MULC, DIVC, LOADC`

  (memória operandussal):

  `ADD, SUB, MUL, DIV, LOAD, STORE`

- Ugró utasitások:

  `JUMP, JUMPEQ, JUMPNE, JUMPLT,`
  `JUMPGT, JUMPLE, JUMPGE`

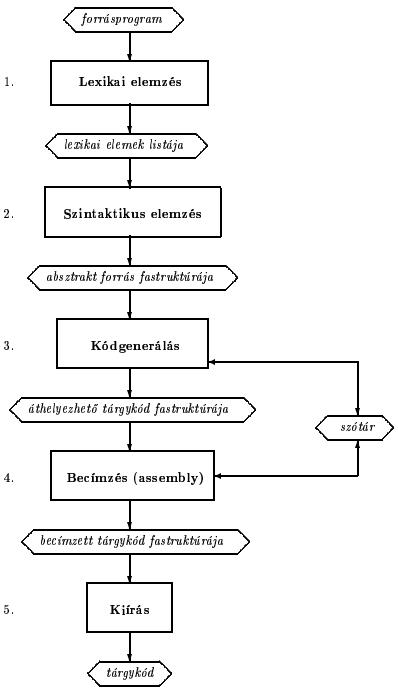
- I/O etc:

  `READ, WRITE, HALT`

A faktoriális programból generált kód

```
        1:  READ    21  value
        2:  LOADC    1
        3:  STORE   19  count
        4:  LOADC    1
        5:  STORE   20  result
L1      6:  LOAD    19  count
        7:  SUB     21  value
        8:  JUMPGE  16  L2
        9:  LOAD    19  count
       10:  ADDC     1
       11:  STORE   19  count
       12:  LOAD    20  result
       13:  MUL     19  count
       14:  STORE   20  result
       15:  JUMP     6  L1
L2     16:  LOAD    20  result
       17:  WRITE    0
       18:  HALT     0
count  19:  BLOCK    3
result 20:
value  21:
```


A fordítóprogram szerkezete



A forrásnyelv absztrakt szintaxisa

<program> ::= <statement>

<statement> ::= `assign(`<name>`,` <expr>`)` |
`if(`<test>`,` <statement>`,` <statement>`)` |
`while(`<test>`,` <statement>`)` |
`read(` <name> `)` |
`write(`<expr>`)` |
<statement>`;` <statement>

<test> ::= `test(`<comparison_op>`,` <expr>`,` <expr>`)`

<expr> ::= `expr(`<op>`,` <expr>`,` <expr>`)` |
`const(`<number>`)` |
`name(`<name>`)`

<comparison_op> ::= `=` | `<` | `>` | `=<` | `>=` | `\=`

<op> ::= `+` | `-` | `*` | `/`

Ugyanez Mercury-szerű típusdefiníciókkal:

```
:- type program == statement.

:- type statement --->  assign(name, expr)
        ;               if(test, statement, statement)
        ;               while(test, statement)
        ;               read(name)
        ;               write(expr)
        ;               { statement ; statement } .
```

A (becímzetlen) tárgykód struktúrája

<target> ::= <code>; block(<integer>)

<code> ::= <instr> |
<code>; <code>

<instr> ::= instr(<mnem>, <addr>) |
label(<label>)

<mnem> ::= LOAD | STORE | ...

<addr> ::= <integer> | <label>

<label> ::= <uninstantiated Prolog variable>

Ugyanez Mercury-szerű típusdefiníciókkal:

```
:- type target --->     instr(mnem, addr)
        ;               label(label)
        ;               block(int)
        ;               { target ; target } .

:- type mnem --->       load ; store ; ... .
:- type addr == int .
:- type label == int .
```


Példa

Forrás:

```
while count<value do
        (count := count+1;
        result := result*count)
```

Absztrakt alak:

```
while(
        test(<, name(count), name(value)),
        (assign(name(count), expr(+, name(count), const(1))) ;
         assign(name(result), expr(*, name(result),
                                    name(count)))
        )
      )
```

Tárgykód:

```
  label(L1) ;
        instr(load, CountAddr) ;
        instr(sub, ValueAddr) ;
        instr(jumpge, L2) ;
        instr(load, CountAddr) ;
        instr(addc, 1) ;
        instr(store, CountAddr) ;
        instr(load, ResultAddr) ;
        instr(mul, CountAddr) ;
        instr(store, ResultAddr) ;
        instr(jump, L1) ;
  label(L2)
```


A szótár struktúrája

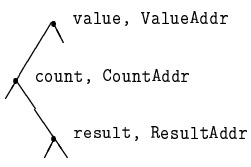
Rendezett bináris fa

<dictionary> ::=
dic(<name>, <value>, <dictionary>, <dictionary>)
| void

A bővíthetőség érdekében a void név helyett üres változót használunk (hasonlóan az üres végű listákhoz). Példa (a korábbi kóddarabnak megfelelő szótár):

```
      dic(value, ValueAddr,
dic(count, CountAddr, _,
            dic(result, ResultAddr, _, _)), _)
```

A fastruktúra grafikus képe:



Keresés/bevitel a szótárba

```
% lookup(Name, Dict, Value):  First solution of the goal:
% the (Name, Value) pair is in the dictionary Dict.

lookup(Name, dic(Name, Value, _, _), Value):-!.
lookup(Name, dic(Name1, _, Before, _), Value):-
      Name @< Name1, lookup(Name, Before, Value).
lookup(Name, dic(Name1, _, _, After), Value):-
      Name @> Name1, lookup(Name, After, Value).
```

Ugyanez (üres végű) listákkal:

```
lookup(Name, Dict, Value):-
      memberchk(Name-Value, Dict).
```

Kódgenerálás: Értékadás fordítása

Forrás: `assign(name(X), Expr)` pl. `c:=c+1`

Tárgykód: <Expr kódja>
STORE <X címe>

```
% encode_statement(St, Dict, Code):  Code is the
% translated form of statement St with respect
% to dictionary Dict.

encode_statement(assign(name(X), Expr), Dict,
        (ExprCode;
         instr(store, Addr))
             ):-
      lookup(X, Dict, Addr),
      encode_expr(Expr, Dict, ExprCode).
```


Kifejezések fordítása

```
% encode_expr(Expr, Dict, Code):  Code is the
% translated form of expression Expr
% with respect to dictionary Dict.

encode_expr(Expr, Dict, Code):-
      encode_subexpr(Expr, 0, Dict, Code).
```

A bevezetett `encode_subexpr` eljárásnak egy további paramétere van, a kifejezés fordításakor nem-használható (már elhasznált) segédváltozók száma.

Az egyes kifejezésfajták fordítása a következő:

Forrás: `const(C)` Tárgykód: `LOADC` <C értéke>
Forrás: `name(X)` Tárgykód: `LOAD` <X címe>

```
% encode_subexpr(Expr, N, Dict, Code):  Code is
% the translated form of expression Expr with
% respect to dictionary Dict, with auxiliary
% variables below N not used in the code.

encode_subexpr(const(C), _, _, instr(loadc, C)).
encode_subexpr(name(X), _, Dict, instr(load, Addr)):-
      lookup(X, Dict, Addr).
```


Kifejezések fordítása *(folytatás)*

Forrás: `expr(Op,Expr1,Expr2)` pl. `c*x+1`
Tárgykód: <Expr1 kódja>
<Op utasítás> <Expr2 vagy Expr2 címe>
feltéve, hogy Expr2 egyszerű (szám vagy név)

```
encode_subexpr(expr(Op, Expr1, Expr2), N, Dict,
        (Expr1Code;
         Instruction)
            ):-
      apply(Op, Expr2, Dict, Instruction),
      encode_subexpr(Expr1, N, Dict, Expr1Code).

% apply(Op, Expr, Dict, Instr):  Expr is a
% simple expression such that the operation
% 'Acc := Acc Op Expr' can be encoded into a single
% instruction Instr, with respect to dictionary Dict.
apply(Op, const(C), _, instr(Opcode, C)):-
      literalop(Op, Opcode).
apply(Op, name(Name), Dict, instr(Opcode, Addr)):-
      memoryop(Op, Opcode),
      lookup(Name, Dict, Addr).
```


Kifejezések fordítása *(folytatás)*

```
% literalop(Op, Mnem):  Mnem is the mnemonic of the
% instruction Acc := Acc Op <Literal Operand>.
literalop(+, addc).
literalop(-, subc).
literalop(*, mulc).
literalop(/, divc).

% memoryop(Op, Mnem):  Mnem is the mnemonic of the
% instruction Acc := Acc Op <Memory Operand>.
memoryop(+, add).
memoryop(-, sub).
memoryop(*, mul).
memoryop(/, div).
```

Kifejezések fordítása *(folytatás)*

Forrás:	`expr(Op,Expr1,Expr2)`	pl. `x+c*2`
Tárgykód:	<`Expr2` kódja>	
	`STORE` <Segédváltozó>	
	<`Expr1` kódja>	
	<`Op` utasítás> <Segédváltozó>	
	feltéve, hogy `Expr2` összetett	

```
encode_subexpr(expr(Op, Expr1, Expr2), N, Dict,
        (Expr2Code;
         instr(store, Addr);
         Expr1Code;
         instr(Opcode, Addr))
             ):-
      complex(Expr2),
      lookup(N, Dict, Addr),
      encode_subexpr(Expr2, N, Dict, Expr2Code),
      N1 is N+1,
      encode_subexpr(Expr1, N1, Dict, Expr1Code),
      memoryop(Op, Opcode).


% complex(Expr):  Expr is a complex expression

complex(expr(_,_,_)).
```


Feltételes utasítások fordítása

Forrás:	`if(Test,Then,Else)`
Tárgykód:	<`Test` kódja>
	<`Then` kódja>
	`JUMP` <cimke_2>
	<cimke_1>:
	<`Else` kódja>
	<cimke_2>:

ahol <`Test` kódja> a <cimke_1>-re ugrik, ha a vizsgálat hamis eredményt ad.

```
encode_statement(if(Test, Then, Else), Dict,
        (TestCode;
         ThenCode;
         instr(jump, L2);
        label(L1);
         ElseCode;
        label(L2))   ):-
      encode_test(Test, Dict, L1, TestCode),
      encode_statement(Then, Dict, ThenCode),
      encode_statement(Else, Dict, ElseCode).
```


Feltételes utasítások fordítása *(folytatás)*

```
% encode_test(Test, Dict, Label, Code):  Code is the
% translated form of test Test with respect to
% dictionary Dict.  Code jumps to Label if Test is
% not satisfied.
encode_test(test(Op, Arg1, Arg2), Dict, Label,
        (ExprCode;
         instr(JumpIf, Label))   ):-
      encode_expr(expr(-, Arg1, Arg2), Dict, ExprCode),
      unlessop(Op, JumpIf).

% unlessop(Op, Mnem):  Mnem is the mnemonic of
% the jump instruction jumping if
% not(Acc Op 0) relation holds.
unlessop(=, jumpne).
unlessop(<, jumpge).
unlessop(>, jumple).
unlessop(\=, jumpeq).
unlessop(=<, jumpgt).
unlessop(>=, jumplt).
```


További utasítások fordítása

```
encode_statement(while(Test, Do), Dict,
        (label(L1);
          TestCode;
          DoCode;
          instr(jump, L1);
         label(L2))
            ):-
      encode_test(Test, Dict, L2, TestCode),
      encode_statement(Do, Dict, DoCode).
encode_statement(read(name(Name)), Dict,
        instr(read, Addr)):-
      lookup(Name, Dict, Addr).
encode_statement(write(Expr), Dict,
        (ExprCode;
         instr(write, 0))
            ):-
      encode_expr(Expr, Dict, ExprCode).
encode_statement((S1;S2), Dict, (Code1;Code2):-
      encode_statement(S1, Dict, Code1),
      encode_statement(S2, Dict, Code2).
```

Becímzés

```
% compile(Source, Code):  Code is the
% compiled form of Source.

compile(Source,
        (Code;
         instr(halt, 0);
         block(L))
      ):-
      encode_statement(Source, Dict, Code),
      assemble(Code, 1, N0),
      N1 is N0+1,
      allocate(Dict, N1, N),
      L is N-N1.

% assemble(Code, N0, N): Code can be positioned so
% that it starts at location N0 and the first
% unused location is N.
assemble((Code1;Code2), N0, N):-
      assemble(Code1, N0, N1),
      assemble(Code2, N1, N).
assemble(instr(_, _), N0, N):-
      N is N0+1.
assemble(label(N), N, N).
```


Becímzés *(folytatás)*

```
% allocate(Dict, N0, N): Locations can be assigned
% to variables in Dict in such a way that the
% first variable is assigned N0, the next N0+1, etc,
% and the first unused location is N.
allocate(void, N, N):-
      !.
allocate(dic(_Name, N1, Before, After), N0, N):-
      allocate(Before, N0, N1),
      N2 is N1+1,
      allocate(After, N2, N).
```


Nyelvtani elemzés

Gyors megoldás: a Prolog elemző használata

```
:-op(990, xfx, :=).      :-op(995, fy, while).
:-op(995, fy, if).       :-op(995, xfy, do).
:-op(995, xfy, then).    :-op(995, fx, read).
:-op(995, xfy, else).    :-op(995, fx, write).

read_program(Prog):-
    read(Term),  parse_statement(Term, Prog).

% parse_statement(Kif, Absz): Absz a Kif Prolog
% kifejezésnek megfelelő absztrakt alak.
parse_statement((V:=E), assign(name(V), Expr)):-
    parse_expr(E, Expr).
...

% Megszorítás:
%     if a=1 then if b=1 then c:=1 else c:=2 else c:=3
% nem elemezhető helyesen, zárojeleket kell használni:
%     if a=1 then (if b=1 then c:=1 else c:=2) else c:=3}
```


Nyelvtani elemzés *(folytatás)*

Teljesértékű megoldás: DCG használata

- Lexikai elemző: karakterfolyam átalakítása lexikai elemek (token) listájává (jegyzet A.5 függeléke).
- Nyelvtani elemző: a lexikaielem-lista átalakítása az absztrakt programformátumra (jegyzet A.4 függeléke).
- DCG használatával ez nagyon egyszerű és természetes.

Példa

```
% statement(Absz, L0, L): L lexikaielem-listáról
% leelemezhető egy <statement>, amelynek absztrakt
% alakja Absz.
statement(assign(V,Expr)) -->
    name(V),[:=],expr(Expr).
statement(if(Test,Then,Else)) -->
    [if], test(Test), [then], statement(Then), [else],
    statement(Else).
...
```

A generált kód kiírása

Lásd a jegyzet A.3 függelékét

Új irányzatok a logikai programozásban

Párhuzamos megvalósítások

- Aurora
- Andorra

Nagyhatékonyságú megvalósítások

- Mercury

CLP — Constraint Logic Programming

- CLP(R)
- CLP(B)
- CLP(FD)

Prolog — előnyök és hátrányok

Miért jó?

- deklaratív, könnyen ellenőrizhető
- 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
- egyes beépített eljárások algoritmikusak
- gyenge következtetési képesség

Hogyan tovább?

- CLP — korlát logikai programozás (constraint logic programming), megkísérli az utóbbi három probléma megoldását ...
- annotációk, típusok — Mercury
- rugalmasabb vezérlés — Oz
- párhuzamos, elosztott végrehajtás — Aurora, Andorra, Oz

Egy Prolog/CLP példasor: Lovagok és lókötők

A feladat

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

Lovagok és lókötők

A Prolog megoldás elvei

- Készítünk egy egyszerű formális nyelvet a bennszülöttek kijelentéseire, pl. `Alfréd mondja Alfréd = lókötő vagy Béla = lókötő`
- A bennszülöttek nevei (pl. `Alfréd`) Prolog változók, amelyek a `lovag` vagy `lókötő` értéket veszik fel.
- A nyelv egyetlen alap-relációja az `=`.
- Az összekötő jeleket (`mondja, és, vagy, nem`) Prolog operátornak deklaráljuk.
- Egy egyszerű Prolog programmal definiáljuk a "bennszülött logikát", azaz a nyelv állításainak igazságértékét.
- A feladat: egy adott mondat esetén megkeresni azokat a változó-behelyettesítéseket, amelyekre a mondat a "bennszülött logika" szerint igaz lesz.

Lovagok és lókötők: Prolog változat

```
:- op(700, fy, nem).     :- op(900, yfx, vagy).
:- op(800, yfx, és).     :- op(950, xfy, mondja).

% Az A bennszülött mondhatja az Áll állítást.
A mondja Áll :- értéke(A mondja Áll, 1).

% értéke(Állítás, Érték): Az Állítás igazságértéke
% Érték (1 = igaz, 0 = hamis).
értéke(X = X, 1).
értéke(X = Y, 0) :-           különböző(X, Y).
értéke(lovag mondja M, E) :-  értéke(M, E).
értéke(lókötő mondja M, E) :- értéke(nem M, E).
értéke(M1 és M2, E) :-        értéke(M1, E1), értéke(M2, E2),
                              E is E1 /\ E2.
értéke(M1 vagy M2, E) :-      értéke(M1, E1), értéke(M2, E2),
                              E is E1 \/ E2.
értéke(nem M, E) :-           értéke(M, E1), E is 1-E1.

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

```
| ?- Alfréd mondja Alfréd = lókötő vagy Béla = lókötő.
                            Béla = lókötő, Alfréd = lovag ?
| ?- A mondja B = C.
                            A = lovag, C = B ? ;
                            A = lókötő, B = lovag, C = lókötő ? ;
                            A = lókötő, B = lókötő, C = lovag ?
| ?- A mondja B mondja C mondja A = C.
                            A = lovag, B = lovag, C = lovag ? ;
                            A = lovag, B = lovag, C = lókötő ? ;
                            A = lókötő, B = lovag, C = lovag ? ;
                            A = lókötő, B = lovag, C = lókötő ?
```


A CLP alapgondolata

A CLP($\mathcal{X}$) séma

Prolog + egy valamilyen $\mathcal{X}$ adattartományra és azon értelmezett korlátokra (relációkra) vonatkozó „erős” következtetési mechanizmus.

Példák az $\mathcal{X}$ tartomány megválasztására

$\mathcal{X}$ = Q vagy R: a racionális vagy valós számok
korlátok = lineáris egyenlőségek és egyenlőtlenségek
következtetési mechanizmus = Gauß elimináció és szimplex módszer

$\mathcal{X}$ = FD (Finite Domain): egész számok véges tartománya
korlátok = különféle aritmetikai és kombinatorikus relációk
következtetési mechanizmus = MI CSP–módszerek (CSP = Constraint-Satisfaction Problem)

$\mathcal{X}$ = B: 0 és 1 Boole értékek
korlátok = ítéletkalkulusbeli relációk
következtetési mechanizmus = MI SAT-módszerek (SAT = Boole kielégíthetőség)

...

Példa: A SICStus clpq könyvtára

A Prologba ágyazás szintaxisa:

{*Korlát*} a *Korlát* felvétele

Példafutás

```
| ?- use_module(library(clpq)).
{loading .../library/clpq.ql...}
...

| ?- {X=Y+4, Y=Z-1, Z=2*X-9}.  % lineáris egyenlet
X = 6, Y = 2, Z = 3 ?

| ?- {X+Y+9<4*Z, 2*X=Y+2, 2*X+4*Z=36}.   % egyenlőtlenség is lehet
{X=1+1/2*Y}, {Z=17/2-1/4*Y}, {Y<48/5} ?

| ?- {(Y+X)*(X+Y)/X = Y*Y/X+100}.  % lineárissá egyszerűsíthető

{X=100-2*Y} ?

| ?- {(Y+X)*(X+Y) = Y*Y+100*X}. % így már nem...
clpq:{2*(X*Y)-100*X+X^2=0} ?
```


A CLP($\mathcal{X}$) séma megvalósítási elvei

A korlát-tár

- *konzisztens* korlátok halmaza (konjunkciója)
- elemei az ún. primitív korlátok (a megengedett korlátok egy részhalmaza)
- az előremenő végrehajtás során a tár csak bővülhet (pontosabb lehet)
- ha a tár inkonzisztenssé válna, visszalépés történik (és a tár is visszaáll)
- a nem-primitív korlátok ágensként (démonként) várakoznak arra, hogy:
  a. primitív korláttá váljanak
  b. a tárat egy primitív korláttal bővíthessék (az ún. erősítés)

A CLP($\mathcal{X}$) séma megvalósítási elvei (folyt.)

Példa várakozó ágensre

```
| ?- {X =< Y}, {X*(Y+1) > X*X+Z},
       (   Z = X*(Y-X), {Y < 0}
       ;   Y = X
       ).
  Y = X,  {X-Z>0} ? ; no
```

A végrehajtás részletei

```
| ?- {X =< Y}, {X*(Y+1) > X*X+Z}.

  {X-Y=<0}, clpq:{Z-X-Y*X+X^2<0} ?

| ?- {X =< Y}, {X*(Y+1) > X*X+Z}, Z = X*(Y-X).

  Z = X*(Y-X), {X-Y=<0}, {X>0} ?

| ?- {X =< Y}, {X*(Y+1) > X*X+Z}, Z = X*(Y-X), {Y < 0}.

  no
| ?- {X =< Y}, {X*(Y+1) > X*X+Z}, Y = X.

  Y = X, {X-Z>0} ?
```


A SICStus `clpb` könyvtár

- **Tartomány**: logikai értékek (`1` és `0`, igaz és hamis)
- **Függvények** (egyben korlát-relációk):

`~ P`	`P` hamis (*negáció*).
`P * Q`	`P` és `Q` mindegyike igaz (*konjunkció*).
`P + Q`	`P` és `Q` legalább egyike igaz (*diszjunkció*).
`P # Q`	`P` és `Q` pontosan egyike igaz (*kizáró vagy*).
`P =:= Q`	Ugyanaz mint `~(P # Q)`.

- **Constraint-megoldó algoritmus:** Boole-egyesítés.

A `library(clpb)` könyvtár eljárásai

- `sat(`*Kifejezés*`)`, ahol *Kifejezés* változókból, a `0` `1` konstansokból és atomokból (ún. szimbolikus konstansok) a fenti müveletekkel felépített logikai kifejezés. Hozzáveszi *Kifejezés*t a korlát-tárhoz.
- `labeling(`*Változók*`)`. Behelyettesíti a *Változók*at `0` `1` értekekre, úgy, hogy a tár teljesüljön. Visszalépéskor felsorolja az összes lehetséges értéket.

Lovagok és lókötők: CLP(B) változat

(A bennszülöttek típusát numerikusan jelöljük: `lovag` →`1`, `lókötő` →`0`.)

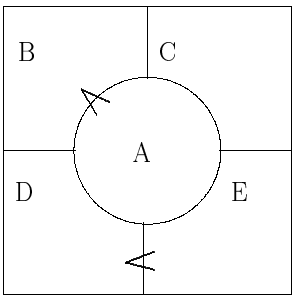
```
:- use_module(library(clpb)).

% értéke(Állítás, Érték): Az Állítás igazságértéke Érték.
értéke(X = Y, E) :-
        sat((X =:= Y) =:= E).
értéke(X mondja M, E) :-
        értéke(M, E0), sat((E0 =:= X) =:= E).
értéke(M1 és M2, E) :-
        értéke(M1, E1), értéke(M2, E2), sat(E =:= E1*E2).
értéke(M1 vagy M2, E) :-
        értéke(M1, E1), értéke(M2, E2), sat(E =:= E1+E2).
értéke(nem M, E) :-
        értéke(M, E0), sat(E =:= ~E0).

| ?- Alfréd mondja Alfréd = 0 vagy Béla = 0.
Béla = 0, Alfréd = 1 ? ;
no
| ?- A mondja B mondja C mondja A = C.
                               B = 1 ? ; no
| ?- A mondja B = C.
                               sat(B=\=C#A) ? ; no
| ?- A mondja B = C, labeling([A,B,C]).
                               A = 0, B = 1, C = 0 ? ;
                               A = 0, B = 0, C = 1 ? ;
                               A = 1, B = 0, C = 0 ? ;
                               A = 1, B = 1, C = 1 ? ; no
```


Véges tartományú korlátok háttere:

CSP problémák

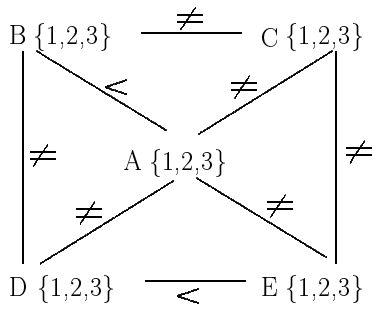
Példa

Az alábbi térkép kiszínezése három színnel, azaz az 1, 2, 3 számok beírása úgy, hogy a szomszédos mezők különböző értéket kapjanak és a beírt relációk is teljesüljenek.



CSP problémák (folyt.)

Az előbbi térképnek megfelelő korlát-gráf:



A CSP megoldás lépései

- a változók tartományainak felvétele
- a korlátok felállítása
- a tartományok szűkítése
- címkézés — visszalépéses keresés

CLP(FD) — A SICStus clpfd könyvtára

Könyvtári eljárások

- változók tartományának beállítása `domain(`*VáltLista*, *Tól*, *Ig*)
- aritmetikai korlátok felállítása *Kif1 Rel Kif2*, ahol *Rel* lehet `#=`, `#<`, `#>`, `#=<`, `#>=`, `#\=`
- címkézés: `labeling(`*Opciólista*, *Változólista*)

A térképszinezési példa SICStus-ban

```
| ?- use_module(library(clpfd)).
| ?- domain([A,B,C,D,E], 1, 3),
     A #> B, A #\= C, A #\= D, A #\= E,
     B #\= C, B #\= D, C #\= E, D #< E.
A in 2..3, B in 1..2,
C in 1..3, D in 1..2, E in 2..3 ?

| ?- domain([A,B,C,D,E], 1, 3),
     A #> B, A #\= C, A #\= D, A #\= E,
     B #\= C, B #\= D, C #\= E, D #< E,
     labeling([], [A,B,C,D,E]).
A = 3, B = 2, C = 1, D = 1, E = 2 ?

| ?- domain([A,B,C,D,E], 1, 3),
     all_distinct([A,B,C]), all_distinct([A,C,E]),
     A #> B, E #> D.
A = 3, B = 2, C = 1, D = 1, E = 2 ?
```


CLP(FD) — A korlát-tár működése

A CLP(FD) korlát-tár

- primitiv korlátok: X `in` *Tartomány*, pl. `X in 2..3 \/ {5}`
- A CLP(FD) tár nem más, mint egy hozzárendelés a változók és lehetséges értékek között
- Új primitív korlát felvétele: egy változó tartományának szűkítése
- A felhasználói korlátok többsége ágens lesz: szűkít, elalszik, aktiválódik, szűkít, ...
- Az ágenseket a korlátbeli változók tartományának változása aktiválja

A korlát-tár működése — példák

`A #\= B`

- Aktiválás: ha vagy `A` vagy `B` konkrét értéket kap.
- Szűkítés: az érték kihagyása a másik változó tartományából
- Folytatás: az ágens befejezi működését

`A #< B`

- Aktiválás: ha `A` alsó határa (min `A`) vagy `B` felső határa (max `B`) változik
- Szűkítés: `A` tartományából kihagyja az $X \geq$ max `B` értékeket,
  `B` tartományából kihagyja az $Y \leq$ min `A` értékeket
- Folytatás: ha max `A` $<$ min `B`, akkor lefut, különben újra elalszik

`all_distinct([`A_1`,...])`

- Jelentése: $A_1, \ldots$ mind különböző értékek.
- Aktiválás: ha bármelyik változó tartománya változik
- Szűkítés: minden olyan értéket elhagy, amelyek esetén a korlát nem állhat fenn. Példa:

```
| ?- domain([A,B], 1, 2), C in 1..3, all_distinct([A,B,C]).
                                C = 3, A in 1..2, B in 1..2 ?
```

- Folytatás: ha a tartományok mind diszjunktak, akkor leáll, különben újra elalszik.

Lovagok és lókötők: CLP(FD) változat

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

% értéke(Állítás, Érték): Az Állítás igazságértéke Érték.
értéke(X = Y, E) :-
        X in 0..1, Y in 0..1,
        E #<=> (X #= Y).
értéke(X mondja M, E) :-
        X in 0..1, értéke(M, E0), E #<=> (E0 #= X).
értéke(M1 és M2, E) :-
        értéke(M1, E1), értéke(M2, E2), E #<=> E1 #/\ E2.
értéke(M1 vagy M2, E) :-
        értéke(M1, E1), értéke(M2, E2), E #<=> E1 #\/ E2.
értéke(nem M, E) :-
        értéke(M, E0), E #<=> #\E0.

| ?- Alfréd mondja Alfréd = 0 vagy Béla = 0.
                        Alfréd in 0..1, Béla in 0..1 ? ; no
| ?- Alfréd mondja Alfréd = 0 vagy Béla = 0,
        labeling([], [Alfréd,Béla]).
                        Béla = 0, Alfréd = 1 ? ; no
| ?- A mondja B = C, labeling([], [A,B,C]).
                        A = 0, B = 0, C = 1 ? ;
                        A = 0, B = 1, C = 0 ? ;
                        A = 1, B = 0, C = 0 ? ;
                        A = 1, B = 1, C = 1 ? ; no
```


Lovagok, lókötők és normálisak

(A bennszülöttek típusa:
`normális` →2, `lovag` →1, `lókötő` →0.)

```
% értéke(Állítás, Érték): Az Állítás igazságértéke Érték.
értéke(X = Y, E) :-
        X in 0..2, Y in 0..2,
        E #<=> (X #= Y).
értéke(X mondja M, E) :-
        X in 0..2, értéke(M, E0), E #<=> (X #= 2 #\/ E0 #= X).
értéke(M1 és M2, E) :-
        értéke(M1, E1), értéke(M2, E2), E #<=> E1 #/\ E2.
értéke(M1 vagy M2, E) :-
        értéke(M1, E1), értéke(M2, E2), E #<=> E1 #\/ E2.
értéke(nem M, E) :-
        értéke(M, E0), E #<=> #\E0.

% http://www.math.wayne.edu/~boehm/Probweek2w99sol.htm
% A: I am normal      B: A is telling the truth
% C: I am not normal
% We are given 3 people, A, B, C, one of whom is a knight,
% one a knave, and one a normal. What are A, B, and C?

| ?- all_distinct([A,B,C]),
     A mondja A = 2, B mondja A = 2, C mondja nem C =2,
     labeling([], [A,B,C]).

A = 0, B = 2, C = 1 ? ;  no
```

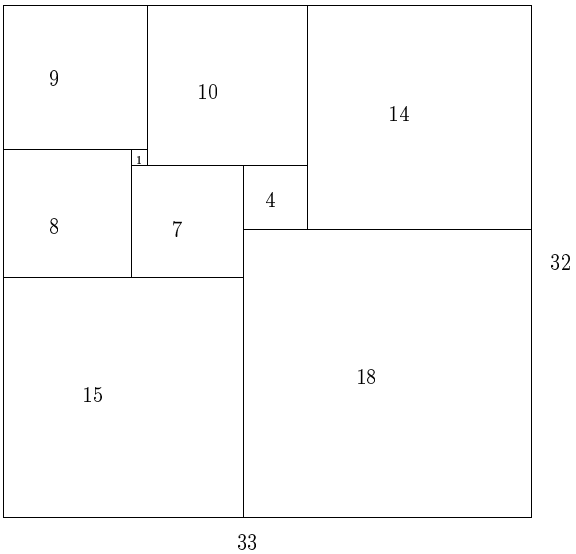

Példa: tökéletes téglalapok

A feladat

- egy olyan téglalap keresése
- amely kirakható páronként különböző oldalú négyzetekből

Egy megoldás

(a legkevesebb, 9 darab négyzet felhasználásával)



Tökéletes téglalapok — CLP(Q) megoldás

```
% Rectangle 1 x Width is covered by distinct squares of size Ss.
filled_rectangle(Width, Ss) :-
  { Width >= 1 }, distinct_squares(Ss),
  filled_hole([-1,Width,1], _, Ss, []).

distinct_squares([]).
distinct_squares([S|Ss]) :-
  { S > 0 }, outof(Ss, S), distinct_squares(Ss).

outof([],     _).
outof([S|Ss], S0) :- { S =\= S0 }, outof(Ss, S0).

% filled_hole(L0, L, Ss0, Ss): Hole in line L0
% filled with squares Ss0-Ss (diff list) gives line L.
filled_hole(L, L, Ss, Ss) :-
  L = [V|_], {V >= 0}.
filled_hole([V|HL], L, [S|Ss0], Ss) :-
  { V < 0 }, placed_square(S, HL, L1),
  filled_hole(L1, L2, Ss0, Ss1), { V1=V+S },
  filled_hole([V1,S|L2], L, Ss1, Ss).

% placed_square(S, HL, L): placing a square on
% HL horizontal line gives (vertical) line L.
placed_square(S, [H,V,H1|L], L1) :-
  { S > H, V=0, H2=H+H1 },
  placed_square(S, [H2|L], L1).
placed_square(S, [S,V|L], [X|L]) :-
  { X=V-S }.
placed_square(S, [H|L], [X,Y|L]) :-
  { S < H, X= -S, Y=H-S }.

?- Ss = [_,_|_], filled_rectangle(Width, Ss).
    Ss = [15/32,9/16,1/4,7/32,1/8,7/16,1/32,5/16,9/32],
    Width = 33/32 ?

/* CPU time: 134 sec */
```

CLP rendszerek a nagyvilágban

Néhány implementáció

- clp(R) — az első CLP(X) rendszer (Monash Univ, Australia, IBM Yorktown Heights és CMU)
- CHIP — FD, Q és B (ECRC, München, Cosytec, Franciao.); CHARME (Bull); Decision Power (ICL)
- Prolog III, Prolog IV (Marseille), Q (nem-lineáris is), B, FD, listák, intervallumok
- ILOG solver (Franciao.) — C++ könyvtár: R (nem-lineáris is), FD, halmazok
- SICStus Prolog (Svédo.) — R/Q, FD, B
- GNU Prolog — FD (C-re fordít)
- Oz (Saarbrücken, Németo.) — constraint alapú elosztott funkcionális nyelv.

CLP rendszerek a nagyvilágban (folytatás)

Kommerciális rendszerek (a fentiek között)

- ILOG, CHIP, Prolog III–IV, SICStus
- a szakma óriása: ILOG
  - szakterület: CLP + vizualizációs eszközök + szabályalapú eszközök
  - felvásárolta az egyik vezető operációkutatási céget, a CPLEX-et
  - 400 munkatárs 7 országban
  - 55M USD éves bevétel
  - NASDAQ-on jegyzett

Mire használják a CLP rendszereket

Ipari erőforrás optimalizálás

- termék- és gépkonfiguráció
- gyártásütemezés
- emberi erőforrások ütemezése
- logisztikai tervezés

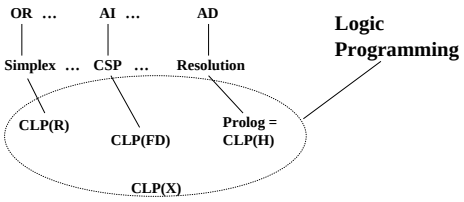
Közlekedés, szállítás

- repülőtéri allokációs feladatok (beszállókapu, poggyászszalag stb.)
- repülő-személyzet járatokhoz rendelése
- menetrendkészítés
- forgalomtervezés

Távközlés, elektronika

- GSM átjátszók frekvencia-kiosztása
- lokális mobiltelefon-hálózat tervezése
- áramkörtervezés és verifikálás

A CLP mint integrációs paradigma



„A CLP a lopás tudománya."

(Mats Carlsson, SICS)