

## 4. fejezet: Prolog programozási módszerek

---

- Az előző előadás-blokk (jegyzetbeli 3. fejezet) célja volt:
  - a Prolog nyelv alapjainak bemutatása,
  - a logikailag „tisztá” résznyelvre koncentrálni.
- A jelen előadás-blokk (jegyzetben a 4. fejezet) célja: olyan
  - beépített eljárások,
  - programozási technikák
 bemutatása, amelyekkel
  - hatékony Prolog programok készíthetők,
  - esetleg a tiszta logikán túlmutató eszközök alkalmazásával.

## II. RÉSZ

### Prolog programozási módszerek: tartalomjegyzék

---

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

## A KERESÉSI TÉR SZŰKÍTÉSE

## Prolog nyelvi eszközök a keresési tér szűkítésére

- **Eszközök**
  - a vágó beépített eljárás: ! (az első Prolog rendszerektől kezdve)
  - feltételes diszjunktív szerkezet (későbbi kiterjesztés): ( if -> then ; else )
- **Feltételes szerkezet — procedurális szemantika (ismétlés)**  
 A ( felt->akkor ; egyébként ) , folyt célsorozat végrehajtása:
  - Végrehajtjuk a felt hívást (egy önálló végrehajtási környezetben).
  - Ha felt sikeres, akkor az akkor , folyt célsorozatra redukáljuk a fenti célsorozatot, a felt első megoldása által eredményezett behelyettesítésekkel. A felt cél **többi megoldását nem keressük meg**.
  - Ha felt sikertelen, akkor az egyébként , folyt célsorozatra redukáljuk.
- **Feltételes szerkezet — alternatív procedurális szemantika:**
  - A feltételes szerkezetet egy speciális diszjunktiónak tekintjük:
 

```
( felt, {vágás}, akkor
; egyébként
)
```
  - A {vágás} jelentése: megszünteti a felt-beli választási pontokat, és egyébként választását is letiltja.

## Feltételes szerkezet: választási pontok a feltételben

- Eddig a főleg determinisztikus (választásmentes) feltételeket mutattunk.
- Példafeladat: első\_poz\_elem(L, P): P az L lista első pozitív eleme.
  - Első megoldás, rekurzióval (mérnöki :-))
 

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

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

van_poz_eleme(L) :- member(P, L), P > 0.
```
  - Harmadik megoldás, feltételes szerkezettel (gyorsprogramozás — Prolog hekker :-))
 

```
első_poz_elem(L, EP) :-
    ( member(EP, L), EP > 0 -> true
; fail % ez a sor elhagyható
).
```
- Figyelem: a harmadik megoldás épít a member / 2 felsorolási sorrendjére!

## A vágó eljárás

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

```
első_poz_elem([EP|_], EP) :- EP > 0, !.
első_poz_elem([X|L], EP) :- X =< 0, első_poz_elem(L, EP).
```
  - Prolog hekker megoldása:
 

```
első_poz_elem(L, EP) :- member(EP, L), EP > 0, !.
```
- **Miért vágunk le ágakat a keresési térben?**
  - mert mi tudjuk, hogy ott nincs megoldás, de a Prolog megvalósítás nem — zöld vágás, szemantikailag „ártalmatlan”
    - (Például, a legtöbb Prolog megvalósítás „nem tudja”, hogy a  $X > 0$  és  $X \leq 0$  feltételek kizárják egymást, lásd indexelés.)
  - ténylegesen eldobunk megoldásokat — vörös vágás, a program jelentését megváltoztatja
    - (Vörös vágás sokszor úgy keletkezik, hogy egy zöld vágót tartalmazó programban a „felesleges” feltételeket elhagyjuk (pl. az  $X =< 0$  feltételt a fenti 2. klózban)

## 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): L utolsó eleme E. (lists könyvtárbeli)
last([E], E) :- !. % zöld vágó
last([_|L], E) :- last(L, E).

% pozitívak(+L, -P): P az L pozitív elemeiből áll.
pozitívak([], []).
pozitívak([E|Ek], [E|Pk]) :-
    E > 0, !, % vörös vágó
    pozitívak(Ek, Pk).
pozitívak([_E|Ek], Pk) :-
    /* \+ _E > 0, */ pozitívak(Ek, Pk).
```

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

## A vágó definíciója

### • Segédfogalom

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

### • A vágó végrehajtása:

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

### • A vágás kétféle választási pontot szüntet meg:

```
r(X) :- s(X), !.    % az s(X)-beli választási pontokat --- a vágót megelőző
                  % cél(ok)nak az első megoldására szorítkozunk
r(X) :- t(X).      % az r(X) többi klózának választását --- a vágót tartalmazó
                  % klóz mellett kötelezzük el magunkat (commit)
```

- A vágó szemléltetése a 4-kapus doboz modellben: a vágó **Újra** kapujából egyenesen a körülvevő (szülő) doboz **Meghiúsulási** kapujára megyünk.

## A vágó által megszüntetett választási pontok

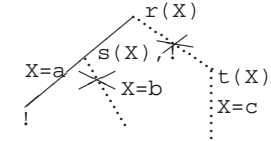
% vágó nélküli példa

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

% ugyanaz a példa vágóval

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

```
s(a).      s(b).      t(c).
```



% a vágó nélküli példa futása

```
:- q(X), write(X), fail.
      ---> abc
```

% a vágót tartalmazó példa futása

```
:- r(X), write(X), fail.
      ---> a
```

## A diszjunktív feltételes szerkezet megvalósítása a vágó segítségével

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

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

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

- Az egyébként alternatíva elmaradhat, ilyenkor a megfelelő klóz is elmarad.
- SICStus módban a `felt` részekben vágó nem lehet, ISO módban lehet, de hatásköre (szülője) a `felt` rész.
- Az `akkor` részekben lehet vágó. Ennek hatásköre, a `->` nyílból generált vágóval ellentétben, a teljes `p` predikátum (ilyenkor a Prolog megvalósítás egy speciális, ún. távolbaható vágót használ).
- Vágót rendkívül ritkán szükséges feltételes szerkezetben szerepeltetni.

## Példák a diszjunktív feltételes szerkezet használatára

% `fakt(+N, ?F): N! = F.`

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

% `last(+L, ?E): az L nem üres lista utolsó eleme E.`

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

% `pozitívak(+L, ?Pk): Pk az L pozitív elemeiből áll.`

```
pozitívak([], []).
pozitívak([E|_], Pk) :-
( E > 0 -> Pk = [E|Pk0]
; Pk = Pk0
),
pozitívak([], Pk0).
```

## A vágás első alapesete — klóz mellett való elkötelezés

- A klóz melletti elkötelezés általában egyszerű feltételes szerkezetet jelent.

```
szülő :- feltétel, !, akkor.
szülő :- egyébként.
```

- A vágó szükségtelessé teszi a feltétel negációjának végrehajtását a többi klózban. A logikailag tiszta, de nem hatékony alak:

```
szülő :- feltétel, akkor.
szülő :- \+ feltétel, egyébként.
```

A fenti két alak csak akkor ekvivalens, ha feltétel egyszerű, nincs benne választás.

- Analogia: ha  $a$ ,  $b$  és  $c$  Boole-értékű változók, akkor  
 $\text{if } a \text{ then } b \text{ else } c \equiv a \wedge b \vee \neg a \wedge c$
- A vágó által kiváltott negált feltételt célszerű kommentként jelezni:

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

## Feltételes szerkezetek

### Feltételes szerkezet — példa

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

### Általános alak

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

### Diszjunktív feltételes szerkezet

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

### Általános alak

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

## Feltételes szerkezetek és fejillesztés

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

```
% a vágó előtt fej-egyesítés:      % az egyesítés explicitté téve:
abs(X, X) :- X >= 0, !.              abs(X, A) :- A = X, X >= 0, !.
abs(X, A) :- A is -X.                abs(X, A) :- A is -X.
```

- A fej-egyesítés gondot okozhat, ha az eljárást ellenőrzésre használjuk:

```
| ?- abs(10, -10). ---> yes
```

- A megoldás a **vágás alapszabálya**:

- A kimenő paraméterek értékadását mindig a vágó után végezzük!

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

- Ez nemcsak általánosabban használható, hanem hatékonyabb kódot is ad: csak akkor helyettesíti be a kimenő paramétert, ha már tudja, mi az értéke (nincs „előre-behelyettesítés”, mint a fenti első két példában).
- („kimenő” paraméterek — vágó alkalmazásakor általában nincs többirányú használat :-)

## A bevezető példának a vágás alapszabályát betartó változata

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

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

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

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

Példasor:  $\max(X, Y, Z)$ : X és Y maximuma Z.

- 1. változat, tiszta Prolog. Lassú (előre-behelyettesítés, két hasonlítás), választási pontot hagy.

```
max(X, Y, X) :- X >= Y.
max(X, Y, Y) :- Y > X.
```

- 2. változat, zöld vágóval. Lassú (előre-behelyettesítés, két hasonlítás), nem hagy választási pontot.

```
max(X, Y, X) :- X >= Y, !.
max(X, Y, Y) :- Y > X.
```

- 3. változat, vörös vágóval. Gyorsabb (előre-behelyettesítés, egy hasonlítás), nem hagy választási pontot, de nem használható ellenőrzésre, pl. `| ?- max(10, 1, 1)` sikerül.

```
max(X, Y, X) :- X >= Y, !.
max(X, Y, Y).
```

- 4. változat, vörös vágóval. Helyes, nagyon gyors (egy hasonlítás, nincs előre-behelyettesítés) és nem is hoz létre választási pontot.

```
max(X, Y, Z) :- X >= Y, !, Z = X.
max(X, Y, Y) /* :- Y > X */.
```

## A vágás második alapesete — első megoldásra való megszorítás

- Mikor használjuk az első megoldásra megszorító vágót?
  - behelyettesítést nem okozó, eldöntendő kérdés esetén;
  - feladatspecifikus optimalizálásra;
  - végtelen választási pontot létrehozó eljárások hasznosítására.
- Eldöntendő kérdés: eljáráshívás csupa bemenő paraméterrel
 

```
% van_eleg_hosszu_ut(+N, +A, +B, +Min):
% A és B között van N lépéses út,
% amelynek összhossza legalább Min km.
van_eleg_hosszu_ut(N, A, B, Min) :-
    utvonal(N, A, B, Hossz), Hossz >= Min, !.
```
- Eldöntendő kérdés esetén általában nincs értelme többszörös választ adni/várni.

## Feladatspecifikus optimalizálás

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

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

/*
% egyformák(Ek, E): Az Ek lista minden eleme E.
egyformák([], _).
egyformák([E|Ek], E) :-
    egyformák(Ek, E).

*/
| ?- kezdethossz([1,1,1,2,3,5], H).
H = 3 ? ; no
```

Végtelen választás megszelidítése: `memberchk` (lists könyvtár)

- `memberchk/2` definíciója:

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

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

- `memberchk/2` használata

- Eldöntő kérdésben (visszalépéskor nem keresi végig a lista maradékát.)

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

- Nyílt végű lista elemévé tesz, pl.:

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

## Nyílt végű listák kezelése memberchk segítségével: szótárprogram

```
szótaraz(Sz):-
    read(M-A), !,
    % A read(X) beépített eljárás egy kifejezést
    % olvas be és egyesíti X-szel
    memberchk(M-A,Sz),
    write(M-A), nl,
    szótaraz(Sz).
szótaraz(_).
```

Egy futása:

```
| ?- szótaraz(Sz).
|: alma-apple.           |: alma-X.
alma-apple              alma-apple
|: korte-pear.          |: X-pear.
korte-pear              korte-pear
                        |: vege.    % nem egyesíthető M-A-val

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

## VEZÉRLÉSI ELJÁRÁSOK

### Vezérlési eljárások, a call/1 beépített eljárás

- Vezérlési eljárás: A Prolog végrehajtáshoz kapcsolódó beépített eljárás (pl. vágó, if-then-else).
- A vezérlési eljárások többsége **magasabbrendű** eljárás, azaz olyan eljárás, amely egy vagy több argumentumát eljáráshívásként értelmezi. (A magasabbrendű Prolog eljárásokat szokás **meta-eljárásnak** is hívni.)
- A meta-eljárások fő képviselője és alapvető építőeleme a call/1:
  - Hívási minta: call(+Cél)
  - Cél egy „meghívható kifejezés” (callable, vö. callable/1), azaz struktúra, vagy névkonstans.
  - Jelentése (deklaratív szemantika): Cél igaz.
  - Hatása (procedurális szemantika): a Cél kifejezést eljáráshívássá alakítja és meghívja.
- A klóztörzsben célként megengedett egy X változó használata, ezt a rendszer egy call(X) hívássá alakítja át.

```
| kétszer(Hívás) :- call(Hívás), Hívás.
| ?- kétszer(write(ba)), nl.      ---> baba
| ?- listing(kétszer).          ---> kétszer(A) :-
                                call(user:A), call(user:A).
```

### Vezérlési szerkezetek mint eljárások

- A call/1 argumentumában szerepelhetnek vezérlési szerkezetek is, mert ezek maguk beépített eljárásként is jelen vannak a Prolog rendszerben:
  - (',' )/2: konjunkció.
  - (;)/2: diszjunkció.
  - (->)/2: if-then.
  - ( ; )/2: if-then-else.
- A call-ban szereplő vezérlési szerkezetek lényegében ugyanúgy futnak, mint az interpretált (consult-tal betöltött) kód.
- Példák:

```
| ?- _Cél = (kétszer(write(ba)), write(' ')), kétszer(_Cél), nl.
baba baba
| ?- kétszer((member(X, [a,b,c,d]), write(X), fail ; nl)).
abcd
abcd
```

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

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

A call/1 viszonylag költséges: egy 1414 hosszú lista megfordítása nrev-vel (kb. 1 millió append hívás), minden append körül egy felesleges call-lal ill. anélkül:

	call nélkül	call-lal	Lassulás
lefordítva	0.140 sec	1.680 sec	12.00
interpretálva	1.710 sec	3.520 sec	2.06

## További beépített vezérlési eljárások

- \+ Cél: Cél „nem bizonyítható”. A beépített eljárás definíciója:

```
\+ X :- call(X), !, fail.
\+ _X.
```

- once (Cél): Cél igaz, és csak az első megoldását kérjük. Definíciója:

```
once(X) :- call(X), !.
```

- true: azonosan igaz (mindig sikertül), fail: azonosan hamis (mindig megghiúsul).

- repeat: végtelen sokszor igaz (egy végtelen választási pontot hoz létre). Definíciója:

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

- A repeat eljárást legtöbbször egy mellékhatásos eljárás ismétlésére használjuk. A végtelen választási pontot kötelező egy vágóval semlegesíteni.

- Példa (egyszerű kalkulátor):

```
bc :- repeat, read(Expr),
    (
        Expr = end_of_file -> true
    ;
        Res is Expr, write(Expr = Res), nl, fail
    ),
    !.
```

## Példa: magasabbrendű reláció definiálása

- Az implikáció ( $P \Rightarrow Q$ ) megvalósítása negáció segítségével:

```
% P minden megoldása esetén Q igaz.
forall(P, Q) :-
    \+ (P, \+Q). % Szintaktikus emlékeztető:
                % az első \+ után kötelező a szóköz!
```

```
| ?- _L = [1,2,3],
    % _L minden eleme pozitív:
    forall(member(X, _L), X > 0).
true ?
| ?- _L = [1,-2,3], forall(member(X, _L), X > 0).
no
| ?- _L = [1,2,3],
    % _L szigorúan monoton növe:
    forall(append(_, [A,B|_], _L), A < B).
true ?
```

- forall/2 csak eldöntendő kérdés esetén használható.

## DETERMINIZMUS ÉS INDEXELÉS

## Determinizmus

- Egy eljáráshívás **determinisztikus**, ha (legfeljebb) egyféleképpen sikerülhet.
- Egy eljáráshívásnak egy sikeres végrehajtása **determinisztikusan futott le**:
  - ha nem hagyott választási pontot a híváshoz tartozó részében, azaz
    - vagy választásmentesen futott le, azaz létre sem hozott választási pontot (figyelem: ez a Prolog megvalósítástól függ!);
    - vagy létrehozott ugyan választási pontot, de megszüntette (kimerítette, levágta).
- A SICStus Prolog nyomonkövetésében ? jelzi a **nem**determinisztikus lefutást:

```

p(1, a). | ?- p(1, X). % det. hívás,
p(2, b). |           1 1 Exit: p(1,a) % det. lefutás
p(3, b). | ?- p(Y, a). % det. hívás,
          | ? 1 1 Exit: p(1,a) % nemdet. lefutás
          | ?- p(Y, b), Y > 2. % nemdet. hívás
          | ? 1 1 Exit: p(2,b) % nemdet. lefutás
          |           1 1 Exit: p(3,b) % det. lefutás
    
```

## A determinisztikus lefutás

- Mi a determinisztikus lefutás haszna?
  - a futás gyorsabb lesz,
  - a tárigény csökken,
  - más optimalizálások (pl. jobbrekurzió) alkalmazható.
- Hogyan ismeri fel a fordító azt, hogy nem kell választási pont?
  - indexelés (indexing)
  - vágók és feltételes szerkezetek
- Az alábbi definíciók esetén a  $p(Nonvar, Y)$  hívás nem hoz létre választási pontot:

```

p(1, a). | p(1, Y) :- !, | p(X, Y) :-
p(2, b). | Y = a. | ( X == 1 -> Y = a
          | p(_, b). | ; Y = b
          |           | ).
    
```

## Indexelés — ismétlés

- Mi az indexelés?
  - egy adott hívásra illeszhető klózok gyors kiválasztása,
  - egy eljárás klózainak fordítási idejű csoportosításával.
- A legtöbb Prolog rendszer, így a SICStus Prolog is, az első fej-argumentum alapján indexel (first argument indexing).
- Az indexelés alapja az első fejjargumentum külső funktora:
  - C szám vagy névkonstans esetén C/0;
  - R nevű és N argumentumú struktúra esetén R/N;
  - változó esetén nem értelmezett.
- Az indexelés megvalósítása:
  - Fordításkor a funktorokhoz elkészítjük az illeszhető klózok részhalmazát.
  - Futáskor lényegében konstans idő alatt választunk a részhalmazok közül.
  - **Fontos:** ha egyelemű a részhalmaz, nem hozunk létre választási pontot!

## Példa indexelésre

```

p(0, a). | /* (1) */ | q(1).
p(X, t) :- q(X). | /* (2) */ | q(2).
p(s(0), b). | /* (3) */ |
p(s(1), c). | /* (4) */ |
p(9, z). | /* (5) */ |
    
```

- A  $p(A, B)$  hívással illesztendő klózhalmaz:
  - $\{(1) (2) (3) (4) (5)\}$  ha A változó;
  - $\{(1) (2)\}$  ha  $A = 0$ ;
  - $\{(2) (3) (4)\}$  ha A fő funktora s/1;
  - $\{(2) (5)\}$  ha  $A = 9$ ;
  - $\{(2)\}$  minden más esetben.
- Példák hívásokra:
  - $p(1, Y)$  nem hoz létre választási pontot.
  - $p(s(1), Y)$  létrehoz választási pontot, de determinisztikusan fut le.
  - $p(s(0), Y)$  nemdeterminisztikusan fut le.



## Struktúrák, változók a fejargumentumban

- Azonos funktorú struktúrák az első fejargumentumban:

- Ha a klózok szétválasztásához szükség van az első (struktúra) argumentum részeire is, akkor érdemes segédjelrást bevezetni.

- Például  $p/2$  és  $q/2$  ekvivalens, de  $q(Nonvar, Y)$  determinisztikusan fut le!

<code>p(0, a).</code>	<code>q(0, a).</code>	<code>q_seged(0, b).</code>
<code>p(s(0), b).</code>	<code>q(s(X), Y) :-</code>	<code>q_seged(1, c).</code>
<code>p(s(1), c).</code>	<code>q_seged(X, Y).</code>	
<code>p(9, z).</code>	<code>q(9, z).</code>	

- Fejlesztés kiváltása egyenlőséggel (vö. rétegelt minta)

- Az indexelés figyelembe veszi a törzs elején szereplő egyenlőséget:

```
p(X, ...) :- X = K if, ... esetén K if funktora szerint indexel.
```

- Példa: lista hosszának reciproka, üres lista esetén 0:

```
rhossz([], 0).
rhossz(L, RH) :- L = [_|_], length(L, H), RH is 1/H.
% rhossz([X|L], RH) :- length([X|L], H), RH is 1/H.
%           % kevésbé hatékony, mert újra felépíti az [X|L] listát.
% rhossz(L, RH) :- L \= [], length(L, H), RH is 1/H.
%           % kevésbé hatékony, mert L=[] esetben választási pontot hagy.
```

## Indexelés — további tudnivalók

- Indexelés és aritmetika

- Az indexelés nem foglalkozik aritmetikai vizsgálatokkal.
- Pl. az  $N = 0$  és  $N > 0$  feltételek nem „zárják ki” egymást.
- Az alábbi `fakt/2` eljárás lefutása nem-determinisztikus:

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

- Indexelés és listák

- Gyakran kell az üres és nem-üres lista esetét szétválasztani.
- A bemenő lista-argumentumot célszerű az első argumentum-pozícióba tenni.
- Az `[]` és `[...|...]` eseteket az indexelés megkülönbözteti (funktork: `'[]' / 0` ill. `'.'/ 2`).
- A két klóz sorrendje nem érdekes (feltéve, hogy zárt listával hívjuk az első pozíció) — de azért tegyük a leálló klózt mindig előre.

## Listakezelő eljárások indexelése: példák

- Az `append/3` választásmentesen fut le, ha első argumentuma zárt végű lista.

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

- A `last/2` közvetlen megfogalmazása nemdeterminisztikusan fut le:

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

- Érdemes segédjelrást bevezetni, `last2/2` választásmentesen fut

```
last2([X|L], E) :- last2(L, X, E).

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

- Az utolsó listaelemet választásmentesen felsoroló `member` (lists könyvtárból):

```
member(E, [H|T]) :- member_(T, H, E).

% member_(L, X, E): Az [X|L] lista eleme E.
member_(_, E, E).
member_([H|T], _, E) :- member_(T, H, E).
```

## Az indexelés és a vágó kölcsönhatása

- Hogyan vehető figyelembe a vágó az indexelés fordításakor?

- Példa: a `p(1, A)` hívás választásmentes, de a `q(1, A)` nem!

<code>p(1, Y) :- !, Y = 2.</code>	<code>% (1)</code>	<code>q(1, 2) :- !.</code>	<code>% (1)</code>
<code>p(X, X).</code>	<code>% (2)</code>	<code>q(X, X).</code>	<code>% (2)</code>
<code>Arg1=1 → (1), Arg1≠1 → (2)</code>		<code>Arg1=1 → {(1),(2)}, Arg1≠1 → (2)</code>	

- A fordító figyelembe veszi a vágót az indexelésben, ha garantált, hogy egy adott fő funktor esetén a vágót elérjük. Ennek feltételei:

- az első argumentum változó, konstans, vagy csak változókat tartalmazó struktúra legyen,
- a további argumentumok változók legyenek,
- a fejben az összes változóelőfordulás különböző legyen,
- a törzs első hívása a vágó (megengedve a fejlesztést kiváltó egyenlőséget).

- Ilyenkor a fordító az adott funktorhoz tartozó listából kihagyja a vágót követő klózokat.

- Példa: `p(X, D, E) :- X = s(A, B, C), !, ... p(X, Y, Z) :- ...`

- Ez egy újabb érv a vágás alapszabálya mellett:

**A kimenő paraméterek értékadását mindig a vágó után végezzük!**

## A vágó és az indexelés hatékonysága

- Egy Fibonacci-szerű sorozat:  $f_1 = 1; f_2 = 2; f_n = f_{\lfloor 3n/4 \rfloor} + f_{\lfloor 2n/3 \rfloor}, n > 2$

% determinisztikus	% determ. lefutású	% választásmentes
fib(1, 1).	fibc(1, 1) :- !.	fibci(1, F) :- !, F = 1.
fib(2, 2).	fibc(2, 2) :- !.	fibci(2, F) :- !, F = 2.
fib(N, F) :-	fibc(N, F) :-	fibci(N, F) :-
N > 2,	N > 2,	N > 2,
N2 is N*3//4,	N2 is N*3//4,	N2 is N*3//4,
N3 is N*2//3,	N3 is N*2//3,	N3 is N*2//3,
fib(N2, F2),	fibc(N2, F2),	fibci(N2, F2),
fib(N3, F3),	fibc(N3, F3),	fibci(N3, F3),
F is F2+F3.	F is F2+F3.	F is F2+F3.

- Futási idők  $N = 2000$  esetén

	fib	fibc	fibci
futási idő	990 msec	890 msec	830 msec
meghiúsulási idő	440 msec	30 msec	0 msec
összesen	1430 msec	920 msec	830 msec
nyom-verem mérete	4.1Mbyte	2.0 Mbyte	256 byte

- fibc esetén a meghiúsulási idő azért nem 0, mert a rendszer a nyom-vermet (trail-stack) dolgozza fel. A nyom-verem tárolja a változó-értékekadások visszacsinálási információit.

## META-LOGIKAI ELJÁRÁSOK

## Választás-mentesség diszjunktív feltételes szerkezetek esetén

- Feltételes szerkezet végrehajtásakor általában választási pont jön létre.
- A **SICStus Prolog** a „( felt -> akkor ; egyébként )” szerkezetet választásmentesen hajtja végre, ha a felt konjunkció tagjai csak:
  - aritmetikai összehasonlító eljárás-hívások (pl. <, =, =:=), és/vagy
  - kifejezés-típust ellenőrző eljárás-hívások (pl. atom, number), és/vagy
  - általános összehasonlító eljárás-hívások (ld. később, pl. @<, @=<, ==).
- Analóg módon választásmentes kód keletkezik a „fej :- felt, !, akkor.” klózból, ha fej argumentumai különböző változók, és felt olyan mint fent.
- Például választásmentes kód keletkezik az alábbi definíciókból:

vektorfajta(X, Y, Fajta) :- ( X :=: 0, Y :=: 0 % X = 0, Y = 0 <b>nem lenne jó</b> -> Fajta = null ; Fajta = nem_null ).	vektorfajta(X, Y, Fajta) :- X :=: 0, Y :=: 0, !, Fajta = null. vektorfajta(_X, _Y, nem_null).
--	--

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

- A Prolog kifejezések pillanatnyi behelyettesítettségi állapotát vizsgáló eljárások (értelemszerűen sorrendfüggetlek):
  - kifejezések osztályozása (1)
 

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

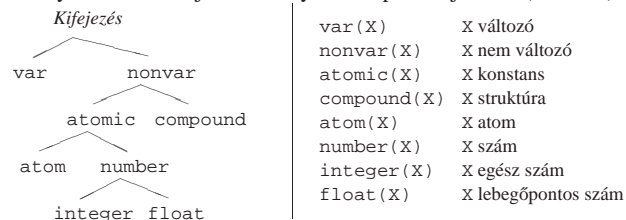
```
| ?- X @< 3 /* X megelőzi 3-t? */, X = 4. => X = 4
  % a változók megelőzik a nem változó kifejezéseket
| ?- X = 4, X @< 3. => no
```
- Prolog kifejezéseket szétszedő vagy összerakó eljárások:
  - (struktúra) kifejezés  $\iff$  név és argumentumok (2)
 

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

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

## Kifejezések osztályozása

- Kifejezés-osztályok fastruktúrája — osztályozó beépített eljárások (ismétlés)



- SICStus-specifikus osztályozó eljárások:

- simple(X): X nem összetett (konstans vagy változó);
- ground(X): X tömör, azaz nem tartalmaz behelyettesíthető változót.

- Az osztályozó eljárások használata — példák

- var, nonvar — többirányú eljárásokban a különböző irányok elágaztatása
- number, atom, ... — nem-megkülönböztetett úniók feldolgozása (pl. szimbolikus deriválás)

## Osztályozó eljárások: elágaztatás behelyettesíthettség alapján

- Példa: a length/2 beépített eljárás megvalósítása (SICStus kód!)

```
% length(?L, ?N): Az L lista N hosszú.
length(L, N) :- var(N), !, length(L, 0, N).
length(L, N) :-                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). (length/3) ==> Len = 2 ? ; no
| ?- length([1,2], 3). (dlength/3) ==> no
| ?- length(L, 3). (dlength/3) ==> L = [_A,_B,_C] ? ;no
| ?- length(L, Len). (length/3) ==> L = [], Len = 0 ? ;
| L = [_A], Len = 1 ? ; L = [_A,_B], Len = 2 ?
```

## Struktúrák szétszedése és összerakása: az univ eljárás

- Az univ eljárás hívási mintái:
  - +Kif =.. ?Lista

- Kif =.. +Lista

- Az eljárás jelentése: Igaz, ha

- Kif = Fun(A<sub>1</sub>, ..., A<sub>n</sub>) és Lista = [Fun, A<sub>1</sub>, ..., A<sub>n</sub>], ahol Fun egy névkonstans és A<sub>1</sub>, ..., A<sub>n</sub> tetszőleges kifejezések; vagy
- Kif = C és Lista = [C], ahol C egy konstans.

- Példák

```
| ?- el(a,b,10) =.. L.      ==> L = [el,a,b,10]
| ?- Kif =.. [el,a,b,10]. ==> Kif = el(a,b,10)
| ?- alma =.. L.          ==> L = [alma]
| ?- Kif =.. [1234].      ==> Kif = 1234
| ?- Kif =.. L.           ==> hiba
| ?- f(a,g(10,20)) =.. L. ==> L = [f,a,g(10,20)]
| ?- Kif =.. [/,X,2+X].   ==> Kif = X/(2+X)
| ?- [a,b,c] =.. L.       ==> L = ['.',a,[b,c]]
```

## Struktúrák szétszedése és összerakása: a functor eljárás

- functor/3: kifejezés funktorának, adott funktorú kifejezésnek az előállítás

- Hívási minták: functor(-Kif, +Név, +Argszám)
- functor(+Kif, ?Név, ?Argszám)
- Jelentése: igaz, ha Kif egy Név/Argszám funktorú kifejezés.

- A konstansok 0-argumentumú kifejezésnek számítanak.
- Ha Kif kimenő, az adott funktorú legáltalánosabb kifejezéssel egyesíti (argumentumaiban csupa különböző változóval).

- Példák:

```
| ?- functor(el(a,b,1), F, N). ==> F = el, N = 3
| ?- functor(E, el, 3).       ==> E = el(_A,_B,_C)
| ?- functor(alma, F, N).    ==> F = alma, N = 0
| ?- functor(Kif, 122, 0).   ==> Kif = 122
| ?- functor(Kif, el, N).    ==> hiba
| ?- functor(Kif, 122, 1).   ==> hiba
| ?- functor([1,2,3], F, N). ==> F = '.', N = 2
| ?- functor(Kif, ., 2).     ==> Kif = [_A|_B]
```

## Struktúrák szétszedése és összerakása: az arg eljárás

- `arg/3`: kifejezés adott sorszámú argumentuma.
  - Hívási minta: `arg(+Sorszám, +StrKif, ?Arg)`
  - Jelentése: A `StrKif` struktúra `Sorszám`-adik argumentuma `Arg`.
  - Végrehajtása: `Arg`-ot az adott sorszámú argumentummal **egyesíti**.
  - Az `arg/3` eljárás így nem csak egy argumentum elővételére, hanem a struktúra változóargumentumának behelyettesítésére is használható (ld. a 2. példát alább).

### Példák:

```
| ?- arg(3, el(a, b, 23), Arg).    => Arg = 23
| ?- K=el(_,_,_), arg(1, K, a),
    arg(2, K, b), arg(3, K, 23). => K = el(a,b,23)
| ?- arg(1, [1,2,3], A).          => A = 1
| ?- arg(2, [1,2,3], B).          => B = [2,3]
```

- Az *univ* visszavezethető a functor és `arg` eljárásokra (és viszont), például:

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

## Az univ alkalmazása: ismétlődő sémák összevonása

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

### 1. megoldás, *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 ? ; no
```

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

- Kifejezés-egyszerűsítés, 2. megoldás, *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).
```

- Kifejezés-egyszerűsítés, általánosítás tetszőleges *tömör* kifejezésre:

```
egyszl(Kif, EKif) :-
    Kif =.. [M|ArgL], egyszl_lista(ArgL, EArgL), EKif0 =.. [M|EArgL],
    % catch(:Cél,?Kiv,:KCél): ha Cél kivételt dob, KCél-t futtatja:
    catch(EKif is EKif0, _, EKif = EKif0).

egyszl_lista([], []).
egyszl_lista([K|Kk], [E|Ek]) :-
    egyszl(K, E), egyszl_lista(Kk, Ek).

| ?- egyszl(f(1+2+a, exp(3,2), a+1+2), E). => E = f(3+a,9.0,a+1+2)
```

## Univ alkalmazása általános kifejezés-bejárásra: kiírás

- A feladat: egy tetszőleges kifejezés kiírása úgy, hogy

- a kétargumentumú operátorok zárójellelt infix formában,
- minden más alap-struktúra alakban jelenjék meg.

```
ki(Kif) :-
    compound(Kif), !, Kif =.. [Func, A1|ArgL],
    ( % kétargumentumú kifejezés, funktora infix operátor
      ArgL = [A2], current_op(_, Kind, Func), infix_fajta(Kind)
    -> write('('), ki(A1),
        write(' '), write(Func), write(' '), ki(A2), write(')')
      ; write(Func),
        write('('), ki(A1), arglistaki(ArgL), write(')')
    ).
ki(Kif) :- write(Kif).

% infix_fajta(F): F egy infix operátorfajta.
infix_fajta(xfx). infix_fajta(xfy). infix_fajta(yfx).

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

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

## Univ alkalmazása általános kifejezés-bejárásra: változómentesítés

- A SICStus Prologban beépített `numbervars(?Kif, +N0, ?N)` eljárás hatása:
  - A tetszőleges `Kif` minden változóját '`$VAR`' (I) alakú kifejezéssel helyettesíti,  $I = N0, \dots, N-1$  (azaz `Kif`-ben  $N-N0$  különböző változó van).
- A '`$VAR`'(0), '`$VAR`'(1), ... kifejezések `write`-tal való kiírásakor változónévként (A, B...) jelennek meg.
- A `write_term(Kif, Opciók)` beépített eljárás kiírja a `Kif` kifejezést, az `Opciók` által meghatározott módon.
- A `numbervars/3` által létrehozott '`$VAR`'/1 struktúrák „eredetiben” is megjeleníthetők:
 

```
| ?- _K = [f(_X),g(_),_X], numbervars(_K, 0, N), write(_K), nl,
           write_term(_K, [quoted(true),numbervars(false)]), nl.
===>    [f(A),g(B),A]
           [f('$VAR'(0)),g('$VAR'(1)),$VAR'(0)]
           N = 2
```
- A feladat: elkészítendő egy `numbervars1/3` eljárás, amely '`$VAR`' helyett '`$myvar`' funktort használ.

## Általános kifejezés-bejárás univ-val: változómentesítés

- A változómentesítés egy saját megvalósítása:
 

```
% A Term kifejezésben levő változókat '$myvar(I)' stb.
% struktúrákkal helyettesíti be, I = N0, ... N-1.
numbervars1(Term, N0, N) :-
    var(Term), !,
    Term = '$myvar'(N0), N is N0+1.
numbervars1(Term, N0, N) :-
    Term =.. [_|Args],
    numbervars1_list(Args, N0, N).

% numbervars1_list(L, N0, N): Az L listában levő változókat
% '$myvar(I)' stb. struktúrákkal helyettesíti be, I = N0, ... N-1.
numbervars1_list([], N, N).
numbervars1_list([A|As], N0, N) :-
    numbervars1(A, N0, N1), numbervars1_list(As, N1, N).

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

## numbervars1 egy alkalmazása

### Két kifejezés azonossága

- A kifejezések azonosak, ha változó-behelyettesítés *nélkül* egyesíthetők;
- 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).

% azonos2/2 és azonos/2 teljesen ekvivalens.
% \+ \+ X : csakkor sikeres amikor X, de változóbehelyettesítést nem okoz
azonos2(X, Y) :-
    \+ \+ (numbervars1(foo(X,Y), 0, _), X = Y).

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

## Univ alkalmazása: részkifejezések keresése

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

```
% kif_szám(?Kif, ?N, ?Kiv): Kif Kiv kiválasztójú része az N szám.
kif_szám(X, N, Kiv) :-
    number(X), !, N = X, Kiv = [].
kif_szám(X, N, [I|Kiv]) :-
    compound(X), % a változó kizárása miatt fontos!
    functor(X, _F, ArgNo), between(1, ArgNo, I), arg(I, X, X1),
    kif_szám(X1, N, Kiv).

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

## Atomok szétszedése és összerakása

- `atom_codes/2`: névkonstans és karakterkód-lista közötti átalakítás
  - Hívási minták: `atom_codes(+Atom, ?KódLista)`  
`atom_codes(-Atom, +KódLista)`
  - Jelentése: Igaz, ha `Atom` karakterkódjainak a listája `KódLista`.
  - Végrehajtása:
    - Ha `Atom` adott (bemenő), és a  $c_1c_2\dots c_n$  karakterekből áll, akkor `KódLista`-t egyesíti a  $[k_1, k_2, \dots, k_n]$  listával, ahol  $k_i$  a  $c_i$  karakter kódja.
    - Ha `KódLista` egy adott karakterkód-lista, akkor ezekből a karakterekből összerak egy névkonstans, és azt egyesíti `Atom`-mal.

- Példák:

```
| ?- atom_codes(ab, Cs).           => Cs = [97,98]
| ?- atom_codes(ab, [0'a|L]).     => L = [98]
| ?- Cs="bc", atom_codes(Atom, Cs). => Cs = [98,99], Atom = bc
| ?- atom_codes(Atom, [0'a|L]).   => hiba
```

## Atomok szétszedése és összerakása — alkalmazási példák

- Keresés névkonstansokban

```
% Atom-ban a Rész nem üres részatom kétszer ismétlődik.
dadogó_rész(Atom, Rész) :-
    atom_codes(Atom, Cs), dadogó(Cs, Ds), atom_codes(Rész, Ds).
```

```
% L-ben a D nem üres részlista kétszer ismétlődik (lásd korábban).
dadogó(L, D) :- D = [_|_],
    append(_, Farok, L), append(D, Vég, Farok), append(D, _, Vég).
```

```
| ?- dadogó_rész(babaruhaha, R).   => R = ba ? ; R = ha ? ; no
```

- Atomok összefűzése

```
% atom_concat(+A, +B, ?C): A és B névkonstansok összefűzése C.
% (Szabványos beépített eljárás atom_concat(?A, ?B, +C) módban is.)
atom_concat(A, B, C) :-
    atom_codes(A, Ak), atom_codes(B, Bk),
    append(Ak, Bk, Ck),
    atom_codes(C, Ck).
```

```
| ?- atom_concat(abra, kadabra, A). => A = abrakadabra ?
```

## Számok szétszedése és összerakása

- `number_codes/2`: szám és karakterkód-lista közötti átalakítás
  - Hívási minták: `number_codes(+Szám, ?KódLista)`  
`number_codes(-Szám, +KódLista)`
  - Jelentése: Igaz, ha `Szám` tízes számrendszerbeli alakja a `KódLista` karakterkód-listának felel meg.
  - Végrehajtása:
    - Ha `Szám` adott (bemenő), és a  $c_1c_2\dots c_n$  karakterekből áll, akkor `KódLista`-t egyesíti a  $[k_1, k_2, \dots, k_n]$  kifejezéssel, ahol  $k_i$  a  $c_i$  karakter kódja.
    - Ha `KódLista` egy adott karakterkód-lista, akkor ezekből a karakterekből összerak egy számot (ha nem lehet, hibát jelez), és azt egyesíti `Szám`-mal.

- Példák:

```
| ?- number_codes(12, Cs).           => Cs = [49,50]
| ?- number_codes(0123, [0'1|L]).     => L = [50,51]
| ?- number_codes(N, "-12.0e1").      => N = -120.0
| ?- number_codes(N, "12e1").         => hiba (nincs .0)
| ?- number_codes(120.0, "12e1").     => no (a szám adott! :-)
```

## Kifejezések rendezése: szabványos sorrend

- A Prolog szabvány definiálja két tetszőleges Prolog kifejezés szabványos sorrendjét.
- Jelölés:  $X \prec Y$  — az  $X$  kifejezés megelőzi az  $Y$  kifejezést a szabványos sorrendben.
- A szabványos sorrend definíciója:
  1. Ha  $X$  és  $Y$  azonos, akkor sem  $X \prec Y$  sem  $Y \prec X$  nem igaz és fordítva.
  2. Ha  $X$  és  $Y$  különböző kifejezésosztályba tartozik, akkor az osztály dönt: *változó*  $\prec$  *lebegőpontos szám*  $\prec$  *egész szám*  $\prec$  *név*  $\prec$  *struktúra*.
  3. Ha  $X$  és  $Y$  változó, akkor az eredmény 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 a lexikografikus (abc) sorrenddel.
  6. Ha  $X$  és  $Y$  struktúra:
    - 6.1. Ha  $X$  és  $Y$  aritása ( $\equiv$  argumentumszáma) különböző,  $X \prec Y \Leftrightarrow X$  aritása kisebb mint  $Y$  aritása.
    - 6.2. Egyébként, ha a rekordok neve különböző,  $X \prec Y \Leftrightarrow X$  neve  $\prec Y$  neve.
    - 6.3. Egyébként (azonos név, azonos aritás) balról az első nem azonos argumentum dönt.
- (A SICStus Prologban kiterjesztésként megengedett végtelen (ciklikus) kifejezésekre a fenti rendezés nem érvényes.)

## Kifejezések összehasonlítása — beépített eljárások

- Két tetszőleges kifejezés összehasonlítását végző eljárások:

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

- Az összehasonlító eljárások logikailag nem tiszták:

```
| ?- X @< 3, X = 4. => X = 4
| ?- X = 4, X @< 3. => no
```

- Az összehasonlítás mindig a belső ábrázolás szerint történik:

```
| ?- [1, 2, 3, 4] @< struktúra(1, 2, 3). => sikerül (6.1 szabály)
```

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

```
% T1 megelőzi T2-t a szabványos sorrendben. (Ekvivalens T1 @< T2 -vel, kivéve
% a változókat, ezek rendezése a T1-T2-beli előfordulásuk szerint történik.)
precedes(T1, T2) :-
```

```
\+ \+ (numbervars(T1-T2, 0, _), prec(T1, T2)).
```

```
% class(+T, -C): A T kifejezés a C-edik kifejezésszintre tartozik.
```

```
class(T, C) :-
    ( T='$VAR'(_) -> C=0           % változó
    ; float(T) -> C=1           % lebegőpontos szám
    ; integer(T) -> C=2         % egész szám
    ; atom(T) -> C=3            % névkonstans
    ; compound(T) -> C=4        % összetett kifejezés
    ).
```

```
% T1 megelőzi T2-t, a változók már '$VAR'(n) struktúrákra vannak lecserélve.
```

```
prec(T1, T2) :-
    class(T1, C1), class(T2, C2),
    ( C1 == C2 ->
        ( C1 == 1 -> T1 < T2     % 4. szabály (lebegőpontos szám)
        ; C1 == 2 -> T1 < T2     % 4. szabály (egész szám)
        ; struct_prec(T1, T2)    % 3., 5. és 6. szabály
        )
        % (változó, név, struktúra)
    ; C1 < C2                    % 2. szabály
    ).
```

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

```
% S1 megelőzi S2-t (S1 és S2 struktúra-kifejezés vagy névkonstans).
```

```
struct_prec(S1, S2) :-
    functor(S1, F1, N1), functor(S2, F2, N2),
    ( N1 < N2 -> true
    ; N1 = N2,
      ( F1 = F2 -> args_prec(1, N1, S1, S2)
      ; atom_prec(F1, F2)
      )
    ).
```

```
% Az S1 struktúra-kifejezés N0, ..., N sorszámú argumentumai
% lexikografikusan megelőzik S2 azonos sorszámú argumentumait.
```

```
args_prec(N0, N, S1, S2) :-
    N0 =< N,
    arg(N0, S1, A1), arg(N0, S2, A2),
    ( A1 = A2 -> N1 is N0+1, args_prec(N1, N, S1, S2)
    ; prec(A1, A2)
    ).
```

```
% Az A1 névkonstans megelőzi az A2 névkonstanst.
```

```
atom_prec(A1, A2) :-
    atom_codes(A1, C1), atom_codes(A2, C2), struct_prec(C1, C2).
```

## JOBBREKURZIÓ ÉS AKKUMULÁTOROK



## Jobbrekurzió (farok-rekurzió, tail-recursion) optimalizálás

- Az általános rekurzió költséges, helyben és időben is.
- Jobbrekurzióról beszélünk, ha
  - a rekurzív hívás a klóztörzs utolsó helyén van, vagy az utolsó helyen szereplő diszjunkció egyik ágának utolsó helyén stb., és
  - a rekurzív hívás pillanatában nincs választási pont a predikátumban (a rekurzív hívást megelőző célok determinisztikusan futottak le, nem maradt nyitott diszjunkciós ág).
- Jobbrekurzió optimalizálás: az utolsó hívás végrehajtása **előtt** a predikátum által lefoglalt hely felszabadul ill. szemétygyűjtésre alkalmassá válik.
- Ez az optimalizálás nemcsak rekurzív hívás esetén, hanem minden **utolsó** hívás esetén megvalósul — a pontos név: utolsó hívás optimalizálás (last call optimisation).
- A jobbrekurzió így tehát nem növeli a memória-igényt, korlátlan mélységig futhat — mint a ciklusok az imperatív nyelvekben. Példa:

```
ciklus(Állapot) :- lépés(Állapot, Állapot1), !, ciklus(Állapot1).
ciklus(_Állapot).
```

## Predikátumok jobbrekurzív alakra hozása — listaösszeg

- A listaösszegzés „természetes”, nem jobbrekurzív definíciója:
 

```
% sum(+L, ?S): Az L számlista elemeinek összege S (S = 0+Ln+Ln-1+...+L1).
sum([], 0).
sum([X|L], S):- sum(L,S0), S is S0+X.
```
- Első jobbrekurzív változat, csak ellenőrzésre használható:
 

```
% sum1(+L, +S): Az L számlista elemeinek összege S (S-L1-L2-...-Ln = 0).
sum1([], 0).
sum1([X|L], S) :- /* S is S0+X helyett: */ S0 is S-X, sum1(L, S0).
```
- Második jobbrekurzív változat, csak kiírni tudja az eredményt:
 

```
% sum2(+L): Az L számlista elemeinek összegét (0+L1+L2+...+Ln) kiírja.
sum2(L):- sum2(L, 0).

% sum2(+L, +S0): Az L lista S0-lal növelt összegét kiírja.
sum2([], S) :- write(S), nl.
sum2([X|L], S0):- S1 is S0+X, sum2(L, S1).
```
- Ahhoz, hogy az összeget **eredményként** ki tudjuk adni, szükséges egy további, kimenő argumentum.

## Jobbrekurzív listaösszeg — akkumulátorpár segítségével

- Harmadik változat: teljes értékű jobbrekurzív lista-összegző:
 

```
% sum3(+L, ?S): Az L számlista elemeinek összege S.
sum3(L, S):- sum3(L, 0, S).

% sum3(+L, +S0, ?S): L elemeit hozzáadva S0-hoz kapjuk S-et. (≡ σ L = S-S0)
sum3([], S, S).
sum3([X|L], S0, S):-
    S1 is S0+X, sum3(L, S1, S).
```
- A jobbrekurzív `sum3` eljárás több mint **3-szor gyorsabb** mint a nem jobbrekurzív `sum`!
- Az **akkumulátor** az imperatív (azaz megváltoztatható értékű) változó fogalmának deklaratív megfelelője:
  - A `sum3(L, S0, S)` predikátumban az `S0` és `S` argumentumok egy akkumulátorpárt alkotnak.
  - Az akkumulátorpár két része az adott változó mennyiség (a példában az összeg) különböző időpontokban vett értékeit mutatja:
    - `S0` az összeg értéke a `sum3/3` **meghívásakor**: az összegző változó kezdőértéke;
    - `S` az összeg értéke a `sum3/3` **lefutása után**: összegző változó végértéke.

## Az akkumulátorok használata

- Az akkumulátorokkal általánosan több egymás utáni változtatást is leírhatunk:
 

```
p(..., A0, A):-
    q0(..., A0, A1), ...,
    q1(..., A1, A2), ...,
    qn(..., An, A).
```
- A `sum3/3` második klóza ilyen alakra hozva:
 

```
sum3([X|L], S0, S):- plus(X, S0, S1), sum3(L, S1, S).

plus(X, S0, S) :- S is S0+X.
```
- Akkumulátorváltozók elnevezési konvenciója: kezdőérték: *Vá1t0*; közbülső értékek: *Vá1t1*, ..., *Vá1tn*; végérték: *Vá1t*.
- A Prolog akkumulátorpár nem más mint a funkcionális programozásból ismert gyűjtőargumentum és a függvény eredményének együttese.



## Akkumulátorok használata — folytatás

### • Három lista összege

```
% sum_3_lists(+L, +LL, +LLL, +S0, ?S): Az L, LL, LLL számlisták
% összegeinek összege S-S0
sum_3_lists(L, LL, LLL, S0, S) :-
    sum3(L, S0, S1), sum3(LL, S1, S2), sum3(LLL, S2, S).
```

Előrebocsátott megjegyzés: a fenti szabály DCG (Definite Clause Grammar) formája

```
sum_3_lists(L, LL, LLL) --> sum3(L), sum3(LL), sum3(LLL).
```

### • Többszörös akkumulálás — listák összege és négyzetösszege

```
% sum12(+L, +S0, ?S, +Q0, ?Q): S-S0 = Σ Li, Q-Q0 = Σ Li*Li
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).
```

### • Többszörös akkumulátorok összevonása

```
% sum12(+L, +S0/Q0, ?S/Q): S-S0 = Σ Li, Q-Q0 = Σ Li*Li
sum12([], SQ, SQ).
sum12([X|L], S0/Q0, SQ) :-
    S1 is S0+X, Q1 is Q0+X*X, sum12(L, S1/Q1, SQ).
```

## Különbséglisták

### • A revapp mint akkumuláló eljárás

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

### • Az L-L0 jelölés (különbséglista): azt a listát nevezi meg, amelyet úgy kapunk, hogy L végéről elhagyjuk L0-t (feltéve, hogy L0 szuffixuma L-nek).

### • Például az [1,2,3] listának megfelelő különbséglisták:

- [1,2,3,4]-[4], [1,2,3,a,b]-[a,b], [1,2,3]-[],...
- A legáltalánosabb (nyílt) különbséglistában a „kivonandó” változó: [1,2,3|L]-L

### • Egy nyílt különbséglista konstans időben összefűzhető egy másikkal:

```
% app_dl(DL1, DL2, DL3): DL1 és DL2 különbséglisták összefűzése DL3.
app_dl(L-L0, L0-L1, L-L1).
| ?- app_dl([1,2,3|L0]-L0, [4,5|L1]-L1, DL).
    => DL = [1,2,3,4,5|L1]-L1, L0 = [4,5|L1]
```

### • A nyílt különbséglista „egyszer használatos”, egy hozzáfűzés után már nem lesz nyílt!

## Különbséglisták (folyt.)

### • Példa: lineáris idejű listafordítás, nrev stílusában, különbséglistával:

```
% nrev(L, DR): Az L lista megfordítása a DR különbséglista.
nrev_dl([], L-L). % L-L ≡ üres különbséglista
nrev_dl([X|L], DR) :-
    nrev_dl(L, DR0),
    app_dl(DR0, [X|T]-T, DR). % [X|T]-T ≡ egyelemű különbséglista
% app_dl(DL1, DL2, DL3): DL1 és DL2 különbséglisták összefűzése DL3.
app_dl(L-L0, L0-L1, L-L1).
% Az L lista megfordítása R
rev(L, R) :-
    nrev_dl(L, R-[]).
```

### • Az nrev\_dl/2 eljárás törzsében érdemes a két hívást megcserélni (jobbrekurzió!).

### • nrev\_dl(L, R-R0) ⇒ rev2(L, R0, R) átalakítással és app\_dl kiküszöbölésével a fenti nrev\_dl/2 eljárásból kapunk egy rev2/3-t, amely azonos revapp/3-mal!

### • Ettől az átalakítástól kb **3-szor gyorsabb** lesz a program ⇒ érdemes a különbséglisták helyett akkumulátorpárokat használni!

### • A továbbiakban a különbséglista jelölést csak a fejkommentek megfogalmazásában használjuk.

## Az append mint akkumuláló eljárás

### • Írjunk egy eleje\_marad(Eleje, L, Marad) eljárást!

```
% eleje_marad(Eleje, L, Marad): Az L lista kezdetén az Eleje lista áll,
% annak L-ből való elhagyása után marad a Marad lista.
eleje_marad([], L, L).
eleje_marad([X|Xs], L0, L) :-
    L0 = [X|L1],
    eleje_marad(Xs, L1, L).
```

### • Az akkumulálási lépés: L0 = [X|L1], egy elem **elhagyása** a lista elejéről.

### • A 2. és 3. argumentum felcserélésével az eleje\_marad eljárás átalakul az append eljárássá!

### • Tehát az append is tekinthető akkumuláló eljárásnak (a 2. és 3. argumentum a szokásos akkumulátorpárokhoz képest fel van cserélve):

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

### • Az akkumulálási lépés: az L0 változó értékül kap egy listát, melynek farka L1, az akkumulálált mennyiség: az a változó, amelyben az összefűzés eredményét várjuk.

Egy mintafeladat:  $a^n b^n$  alakú sorozat előállítás● Első megoldás,  $3n$  lépés

```
% anbn(N, L): Az L lista N db a-ból
% és azt követő N db b-ből áll.
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, [A|L]) :-
    N > 0,
    N1 is N-1,
    an(N1, A, L0, L).
```

 $a^n b^n$  alakú sorozatok (folyt.)● Harmadik megoldás,  $n$  lépés

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

% anbn(N, L0, L): Az L-L0 lista N db a-ból és azt követő N db b-ből áll.
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
```

 $a^n b^n$  alakú sorozatok — más nyelvű megoldások

## ● C++ megoldás

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

## Összetettebb adatstruktúrák akkumulálása

## ● Az adatstruktúra:

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

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

● Egészek gyűjtése **rendezett** bináris fában

- beszur(BFa0, E, BFa): Az E egész számnak a BFa0 fába való beszúrása a BFa bináris fát eredményezi.
- Itt BFa0 és BFa egy akkumulátorpár, de az indexelés érdekében BFa0 az első argumentum-pozícióba kerül.

## ● Példafutás:

```
| ?- beszur(ures, 3, Fa0),
    beszur(Fa0, 1, Fa1),
    beszur(Fa1, 5, Fa2).
```

```
Fa0 = bfa(3,ures,ures),
Fa1 = bfa(3,bfa(1,ures,ures),ures),
Fa2 = bfa(3,bfa(1,ures,ures),bfa(5,ures,ures)) ?
```

## Akkumulálás bináris fákkal

## ● Elem beszúrása bináris fába

```
% beszur(BF0, E, BF): E beszúrása BF0 rendezett fába
% a BF rendezett fát adja
% :- pred beszur(bfa::in, 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 fává

```
% 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 listává

```
% 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
```

## IMPERATÍV PROGRAMOK ÁTÍRÁSA PROLOGBA

## Hogyan írjunk át imperatív nyelvű algoritmust Prolog programmá?

- Példafeladat: Hatékony hatványozási algoritmus
  - Alaplépés: a kitevő felezése, az alap négyzetre emelése.
  - Lényegében a kitevő kettes számrendszerbeli alakja szerint hatványoz.

- Az algoritmust megvalósító C nyelvű függvény:

```

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

```

- Az algoritmusban három változó van: a, h, e:
  - a és h végértékére nincs szükség,
  - e végső értéke szükséges (ez a függvény eredménye).

## A hatv C függvénynek megfelelő Prolog eljárás

- Egy kétargumentumú C függvénynek egy 2+1-argumentumú Prolog eljárás felel meg.
- A függvény eredménye a reláció utolsó argumentuma lesz:  $\text{hatv}(+A, +H, ?E): A^H = E$ .
- A ciklusnak segédeljárás felel meg:  $\text{hatv}(+A0, +H0, +E0, ?E): A0^{H0} * E0 = E$ .
- Az »a« és »h« C változóknak az »+A« és »+H« bemenő paraméterek (nem kell a végérték), az »e« C változónak az »+E0, ?E« akkumulátorpár felel meg (kezdőérték, végérték).

<pre> hatv(A, H, E) :-     hatv(A, H, 1, E).  hatv(A0, H0, E0, E) :- H0 &gt; 0, !,     ( H0 /\ 1 =:= 1       % /\ ≡ bitenkénti "és"     -&gt; E1 is E0*A0       ; E1 = E0     ),     H1 is H0 &gt;&gt; 1,     A1 is A0*A0,     hatv(A1, H1, E1, E).  hatv(_, _, E, E). </pre>	<pre> int hatv(int a, unsigned h) {     int e = 1;      ism: if (h &gt; 0)         { if (h &amp; 1)           e *= a;            h &gt;&gt;= 1;           a *= a;           goto ism;         }     else return e; } </pre>
---	---

## A C ciklus és a Prolog eljárás kapcsolata

- A ciklust megvalósító Prolog eljárás minden pontján minden C változónak megfeleltethető egy Prolog változó (pl. h-nak H0, H1, ...):
  - A ciklusmag elején a C változók a megfelelő Prolog argumentumban levő változónak felelnek meg.
  - Egy C értékadásnak egy új Prolog változó bevezetése felel meg, az ez után következő kódban az új változó felel meg a C változónak.
  - Ha a diszjunkció egyik ága megváltoztat egy változót, akkor a többi ágon is be kell vezetni az új Prolog változót, a régivel azonos értékkel (ld. `if (h & 1) ...`).
- A C ciklusmag végén a Prolog eljárást vissza kell hívni, argumentumaiban az egyes C változóknak pillanatnyilag megfeleltetett Prolog változóval.
- A C ciklus **ciklus-invariánsa** nem más mint a Prolog eljárás fejkommentje, a példában:
 
$$\% \text{hatv}(+A0, +H0, +E0, ?E): A0^{H0} * E0 = E.$$

## Programhelyesség-bizonyítás

- Egy algoritmus (függvény) specifikációja:
  - **előfeltételek:** a bemenő paramétereknek teljesíteniük kell ezeket,
  - **utófeltételek:** a paraméterek és az eredmény kapcsolatát írják le.
- Egy algoritmus **helyes**, ha minden, az előfeltételeket kielégítő adatra a függvény hibátlanul lefut, és eredményére fennállnak az utófeltételek.
- Példa:  $x = \text{mfoku\_gyok}(a, b, c)$ 
  - előfeltételek:  $b*b - 4*a*c \geq 0$ ,  $a \neq 0$
  - utófeltétel:  $a*x*x + b*x + c = 0$
  - a program:
 

```

double mfoku_gyok(a, b, c)
double a, b, c;
{ double d = sqrt(b*b-4*a*c);
  return (-b+d)/2/a;
}

```
- A program helyességének bizonyítása lineáris kódra viszonylag egyszerű.

## Ciklikus programok helyességének bizonyítása

- A ciklusokat „fel kell vágni” egy **ciklus-invariánssal**, amely:

- az előfeltételekből és a ciklust megelőző értékadásokból következnek,
- ha a ciklus elején fennáll, akkor a ciklus végén is (indukció),
- belőle és a leállási feltételből következik a ciklus utófeltétele.

```
int hatv(int a0, unsigned h0) /* utófeltétel: hatv(a0, h0) = a0h0 */
{ int e = 1, a = a0, h = h0;
  while /* ciklus-invariáns: a0h0 == e*ah */ (h > 0)
  {
    /* induláskor a kezdőértékek alapján triviálisan fennáll */
    if (h & 1) e *= a;          /* e' = e * ah&1 */
    h >>= 1;                  /* h' = (h-(h&1))/2 */
    a *= a;                   /* a' = a*a */
  }
  /* indukció: e'*ah' = ... = e*ah */
  return e;
  /* Az invariánsból h = 0 miatt következik az utófeltétel */
}
```

## Második példa: Fibonacci sorozat tagjainak hatékony számítása

- A C függvény

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

- Az (1) ciklusnak bemenő változói:  $n$ ,  $f$ ,  $fnxt$ , kimenő változója:  $f$ .
- A ciklusnak megfeleltetett Prolog eljárás:  $fib(N, F0, FNXT, F)$ : az  $F0$  és  $FNXT$  kezdőértékű Fibonacci sorozat  $N$ -edik tagja  $F$ .

<pre>% "betű szerinti" Prolog átírás: fib(N, F0, FNXT, F) :- N &gt; 0, !,   T = FNXT, FNXT1 is FNXT+F0,   F1 = T, N1 is N-1,   fib(N1, F1, FNXT1, F). fib(_, F0, _, F0).</pre>	<pre>% Leegyszerűsített alak: fib(N, F0, FNXT, F) :- N &gt; 0, !,   FNXT1 is FNXT+F0,   N1 is N-1,   fib(N1, FNXT, FNXT1, F). fib(_, F0, _, F0).</pre>
--	--

## Fibonacci sorozat — Prolog stílusban

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

```
fib(N, F) :-                               % unsigned fib(unsigned N)
  fib(N, 0, 1, F).                          % { unsigned F0=0, F1=1, F2;
                                             %
fib(N, F0, F1, F) :-                       % ism:
  N > 0, !,                                  % if (N > 0)
  N1 is N-1,                                 % { --N;
  F2 is F0+F1,                               % F2 = F0+F1;
                                             % F0 = F1; F1 = F2;
  fib(N1, F1, F2, F).                       % goto ism;
                                             % }
fib(_, F0, _, F0).                          % return F0;
                                             % }
```

MEGOLDÁSGYŰJTŐ BEÉPÍTETT ELJÁRÁSOK

## Keresési feladat Prologban — felsorolás vagy gyűjtés?

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

```
% Gyűjtés:
% páros_elemei(L, Pk): Pk az L
% lista páros elemeinek listája.
páros_elemei([], []).
páros_elemei([X|L], Pk) :-
    X mod 2 =\= 0, !,
    páros_elemei(L, Pk).
páros_elemei([_P|L], [_P|Pk]) :-
    páros_elemei(L, Pk).

% Felsorolás:
% páros_eleme(L, P): P egy páros
% eleme az L listának.
páros_eleme([X|L], P) :-
    X mod 2 =:= 0, P = X.
páros_eleme([_X|L], P) :-
    % _X akár páros, akár páratlan
    % folytatjuk a felsorolást:
    páros_eleme(L, P).

% egyszerűbb megoldás:
páros_eleme2(L, P) :-
    member(P, L), P mod 2 =:= 0.
```

Deklaratív programozás. BME VIK, 2010. őszi félév

(Logikai Programozás)

## Gyűjtés és felsorolás kapcsolata

- Vizsgáljuk meg, hogyan lehet egy felsoroló eljárást visszavezetni a gyűjtőre, és fordítva:

- felsorolás gyűjtésből: a member/2 könyvtári eljárás segítségével, pl.

```
páros_eleme(L, P) :-
    páros_elemei(L, Pk), member(P, Pk).
```

Természetesen ez így nem hatékony!

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

```
páros_elemei(L, Pk) :-
    findall(P, páros_eleme(L, P), Pk).
% A páros_eleme(L, P) cél
% összes P megoldásának listája Pk.
```

Deklaratív programozás. BME VIK, 2010. őszi félév

(Logikai Programozás)

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

- Az eljárás végrehajtása (procedurális szemantikája):
  - a Cél kifejezést eljáráshívásként értelmezi, meghívja (A : annotáció meta- (azaz eljárás) argumentumot jelez);
  - minden egyes megoldásához előállítja Gyűjtő egy *másolatát*, azaz a megoldásbeli változókat, ha vannak, szisztematikusan újjal helyettesíti;
  - Az összes Gyűjtő értéket egy listába összegyűjti, és ezt egyesíti Lista-val.
- Példák az eljárás használatára:

```
| ?- findall(X, (member(X, [1,7,8,3,2,4]), X>3), L).
    => L = [7,8,4] ? ; no
| ?- findall(X-Y, (between(1, 3, X), between(1, X, Y)), L).
    => L = [1-1,2-1,2-2,3-1,3-2,3-3] ? ; no
```

- Az eljárás jelentése (deklaratív szemantikája):  
 Lista = { Gyűjtő másolat | (∃X...Z)Cél igaz }  
 ahol X, ..., Z a findall hívásban levő szabad változók (azaz olyan, a hívás pillanatában behelyettesítetlen változók, amelyek a Cél-ban előfordulnak de a Gyűjtő-ben nem).

Deklaratív programozás. BME VIK, 2010. őszi félév

(Logikai Programozás)

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

- Az eljárás végrehajtása (procedurális szemantikája):
  - a Cél kifejezést eljáráshívásként értelmezi, meghívja;
  - összegyűjti a megoldásait (a Gyűjtő-t és a szabad változók behelyettesítéseit);
  - a szabad változók összes behelyettesítését *felsorolja* és mindegyikhez a Lista-ban megadja az összes hozzá tartozó Gyűjtő értéket.
- Példák az eljárás használatára:

```
gráf([a-b,a-c,b-c,c-d,b-d]).
```

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

- A bagof eljárás jelentése (deklaratív szemantikája):  
 Lista = { Gyűjtő | Cél igaz }, Lista ≠ [].

Deklaratív programozás. BME VIK, 2010. őszi félév

(Logikai Programozás)

## A bagof megoldásgyűjtő eljárás (folyt.)

- Explicit kvantorok
  - `bagof(Gyűjtő, V1 ^ ... ^ Vn ^ Cél, Lista)` alakú hívása a `V1, ..., Vn` változókat egzisztenciálisan kötöttnek tekinti, nem sorolja fel.
  - jelentése:  $Lista = \{ Gyűjtő \mid (\exists V1, \dots, Vn) Cél \text{ igaz} \} \neq []$ .  

```
| ?- gráf(_G), bagof(B, A^member(A-B, _G), VegP).
    => VegP = [b,c,c,d,d] ? ; no
```
- Egymásba ágyazott gyűjtések
  - szabad változók esetén a `bagof` nondeterminisztikus lehet, így skatulyázható:  

```
% A G irányított gráf fokszámlistája FL:
% FL = { A-N | N = |{ V | A-V ∈ G }| }
fokszámai(G, FL) :-
    bagof(A-N, Vk^(bagof(V, member(A-V, G), Vk),
                    length(Vk, N)
                    ), FL).
| ?- gráf(_G), fokszámai(_G, FL).
    => FL = [a-2,b-2,c-1] ? ; no
```

## A bagof megoldásgyűjtő eljárás (folyt.)

- Fokszámlista hatékonyabb előállítás
  - a vezérlési szerkezeteket célszerű elkerülni a meta-argumentumokban
  - segédeljárás bevezetésével a kvantor is szükségtelenné válik:  

```
% Az A pont foka a G irányított gráfban N, N>0.
pont_foka(A, G, N) :-
    bagof(V, member(A-V, G), Vk), length(Vk, N).
% A G irányított gráf fokszámlistája FL:
fokszámai(G, FL) :- bagof(A-N, pont_foka(A, G, N), FL).
```
- Példák a `bagof/3` és `findall/3` közötti kisebb különbségekre:
 

```
| ?- findall(X, (between(1, 5, X), X<0), L). => L = [] ? ; no
| ?- 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)] ? ; no
| ?- bagof(S, member(S, [f(X,X),g(X,Y)]), L).
    => L = [f(X,X),g(X,Y)] ? ; no
```
- A `bagof/3` logikailag tisztább mint a `findall/3`, de időigényesebb!

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

- az eljárás végrehajtása:
  - ugyanaz mint: `bagof(Gyűjtő, Cél, L0), sort(L0, Lista)`,
  - itt `sort/2` egy univerzális rendező eljárás (lásd később), amely
  - az eredménylistát rendezzi (az ismétlődések kiszűrésével).
- Példa a `setof/3` eljárás használatára:
 

```
gráf([a-b,a-c,b-c,c-d,b-d]).

% Gráf egy pontja P.
pontja(P, Gráf) :- member(A-B, Gráf), ( P = A ; P = B).

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

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

## EGYENLŐSÉGFAJTÁK — ÖSSZEFOGLALÁS

## A Prolog egyenlőség-szerű beépített eljárásai

<ul style="list-style-type: none"> <li>• <math>U = V</math>: <math>U</math> egyesítendő <math>V</math>-vel. Soha sem jelez hibát.</li> </ul>	<pre>?- X = 1+2.    =&gt; X = 1+2 ?- 3 = 1+2.    =&gt; no</pre>
<ul style="list-style-type: none"> <li>• <math>U == V</math>: <math>U</math> azonos <math>V</math>-vel. Soha sem jelez hibát és soha sem helyettesít be.</li> </ul>	<pre>?- X == 1+2.   =&gt; no ?- 3 == 1+2.   =&gt; no ?- +(1,2)==1+2 =&gt; yes</pre>
<ul style="list-style-type: none"> <li>• <math>U ::= V</math>: Az <math>U</math> és <math>V</math> aritmetikai kifejezések értéke megegyezik. Hibát jelez, ha <math>U</math> vagy <math>V</math> nem (tömör) aritmetikai kifejezés.</li> </ul>	<pre>?- X ::= 1+2.  =&gt; <b>hiba</b> ?- 1+2 ::= X.  =&gt; <b>hiba</b> ?- 2+1 ::= 1+2. =&gt; yes ?- 2.0 ::= 1+1. =&gt; yes</pre>
<ul style="list-style-type: none"> <li>• <math>U \text{ is } V</math>: <math>U</math> egyesítendő a <math>V</math> aritmetikai kifejezés értékével. Hiba, ha <math>V</math> nem (tömör) aritmetikai kifejezés.</li> </ul>	<pre>?- 2.0 is 1+1. =&gt; no ?- X is 1+2.    =&gt; X = 3 ?- 1+2 is X.    =&gt; <b>hiba</b> ?- 3 is 1+2.    =&gt; yes ?- 1+2 is 1+2. =&gt; no</pre>
<ul style="list-style-type: none"> <li>• <math>(U = . . V</math>: <math>U</math> „szétszedettje” a <math>V</math> lista)</li> </ul>	<pre>?- 1+2 =.. X.  =&gt; X = [+ , 1 , 2] ?- X =.. [f,1]. =&gt; X = f(1)</pre>

## A Prolog nem-egyenlőség jellegű beépített eljárásai

- A nem-egyenlőség jellegű eljárások soha sem helyettesítenek be változót!
  - $U \backslash= V$ :  $U$  nem egyesíthető  $V$ -vel.  
Soha sem jelez hibát.

```
?- X \= 1+2.    => no
?- +(1,2) \= 1+2. => no
```

- $U \backslash== V$ :  $U$  nem azonos  $V$ -vel.  
Soha sem jelez hibát.

```
?- X \== 1+2.   => yes
?- 3 \== 1+2.   => yes
?- +(1,2)\==1+2 => no
```

- $U \backslash= V$ : Az  $U$  és  $V$  aritmetikai kifejezések értéke különbözik.  
Hibát jelez, ha  $U$  vagy  $V$  nem (tömör) aritmetikai kifejezés.

```
?- X \= 1+2.    => hiba
?- 1+2 \= X.    => hiba
?- 2+1 \= 1+2.  => no
?- 2.0 \= 1+1.  => no
```

## A Prolog (nem-)egyenlőség jellegű beépített eljárásai — példák

$U$	$V$	Egyesítés		Azonosság		Aritmetika		
		$U = V$	$U \backslash= V$	$U == V$	$U \backslash== V$	$U ::= V$	$U \backslash= V$	$U \text{ is } V$
1	2	no	yes	no	yes	no	yes	no
a	b	no	yes	no	yes	error	error	error
1+2	+(1,2)	yes	no	yes	no	yes	no	no
1+2	2+1	no	yes	no	yes	yes	no	no
1+2	3	no	yes	no	yes	yes	no	no
3	1+2	no	yes	no	yes	yes	no	yes
X	1+2	X=1+2	no	no	yes	error	error	X=3
X	Y	X=Y	no	no	yes	error	error	error
X	X	yes	no	yes	no	error	error	error

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

## MEGOLDÁSOK GYŰJTÉSE ÉS FELSOROLÁSA



## Keresési feladat Prologban — felsorolás vagy gyűjtés (ism.)?

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

<pre><b>% Gyűjtés:</b> % páros_elemei(L, Pk): Pk az L % lista páros elemeinek listája. páros_elemei([], []). páros_elemei([X L], Pk) :-     X mod 2 =\= 0, !,     páros_elemei(L, Pk). páros_elemei([P L], [P Pk]) :-     páros_elemei(L, Pk).</pre>	<pre><b>% Felsorolás:</b> % páros_eleme(L, P): P egy páros % eleme az L listának. páros_eleme([X L], P) :-     X mod 2 =:= 0, P = X. páros_eleme([_X L], P) :-     % _X akár páros, akár páratlan     % folytatjuk a felsorolást:     páros_eleme(L, P).  % egyszerűbb megoldás: páros_eleme2(L, P) :-     member(P, L), P mod 2 =:= 0.</pre>
--	---

## Mi a közös a felsoroló és gyűjtő eljárásokban?

- Keressük meg a közös részt a páros\_elemei és páros\_eleme eljárásokban!
- Mindkettőben át kell lépni a páratlan elemeket, és meg kell keresni az első páros elemet a listában:

```
% köv_páros(L0, P, L) :- Az L0 első páros eleme P, a maradék L.
köv_páros([X|L0], P, L) :-
    X mod 2 =\= 0, !, köv_páros(L0, P, L).
köv_páros([P|L], P, L).
```

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

<pre>% páros_elemei(L, Pk): Pk az L % lista páros elemeinek listája. páros_elemei(L0, Pk) :-     köv_páros(L0, P, L1), !,     Pk = [P Pk1],     páros_elemei(L1, Pk1). páros_elemei(_, []).</pre>	<pre>% páros_eleme(L, P): P egy páros % eleme az L listának. páros_eleme(L0, P) :-     köv_páros(L0, P0, L1),     ( P = P0     ; páros_eleme(L1, P)     ).</pre>
---	--

## A gyűjtő és felsoroló sémák összehasonlítása

- A páros elemeket gyűjtő ill. felsoroló eljárások alapján adjunk meg egy általános sémát a kétféle eljárástípusra!
- Az általános esetben a keresésnek lehet egy vagy több Param paramétere. Például, kereshetjük a Param-mal osztható elemeket.
- A közös építőelem: következő(V0, Param, E, V1): A V0 kifejezéssel jellemzett keresési térben az első megoldás E, és a fennmaradó keresési tér V1, a Param paraméter-érték mellett.

A gyűjtő séma:

```
% A V0 keresési térben a Param
% paraméterű megoldások listája L.
megoldások(V0, Param, L) :-
    következő(V0, Param, E, V1), !,
    L = [E|L1],
    megoldások(V1, Param, L1).
megoldások(_, _, []).
```

A felsoroló séma:

```
% A V0 keresési térben E egy
% Param paraméterű megoldás.
megoldás(V0, Param, E) :-
    következő(V0, Param, E0, V1),
    ( E = E0
    ; megoldás(V1, Param, E)
    ).
```

## Egy összetettebb példa: fennsíkok felsorolása

- Egy listában fennsíknak nevezünk:
  - egy csupa azonos elemből álló, legalább kételemű, folytonos részlistát;
  - amely az ilyenek között maximális (egyik irányba sem kiterjeszthető).
- A feladat: felsorolandók egy lista fennsíkjai és kezdőpozíciójuk.
- fennsík(L, F, H): Az L listában az F (1-től számozott) pozíción egy H hosszú fennsík van.
- Egy gyorsprogramozási módszerrel készült (Prolog hekker) megoldás:

<pre>fennsík0(L, F, H) :-     Teste = [E,E _],     append(Eleje, Teste, L),     \+ last(Eleje, E),     length(Eleje, F0), F is F0+1,     kezdehossz(Teste, H). % kezdehossz/2 definícióját % lásd korábban</pre>	<pre>fennsík1(L, F, H) :-     Teste = [E,E _],     append(Eleje, Teste, L),     \+ last(Eleje, E),     length(Eleje, F0), F is F0+1,     % kezdehossz/2 kifejtve:     ( append(Ek, Farok, Teste),       \+ Farok = [E _] -&gt;         length(Ek, H)     ).</pre>
--	---

## Fennsíkok felsorolása — 2., hatékony megoldás

- Használjuk a megoldás-felsoroló sémát: megoldás(*VO*, *Param*, *E*)!

- *VO*: »L, P«, a bejárando lista és első elemének pozíciója;
- *Param*: üres;
- *E*: »F, H«, a megoldás-fennsík kezdőpozíciója és hossza.

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

```
% A P0-tól számozott L0 listában az F pozíción
% egy H hosszú fennsík van.
fennsík(L0, P0, F, H) :-
    % az első fennsík jellemzői F0 és H0,
    % a fennsík utáni maradéklista L1:
    első_fennsík(L0, P0, F0, H0, L1),
    ( F = F0, H = H0
    ; P1 is F0+H0, % L1 kezdőpozíciója, P1, nem más mint
                  % az előző megoldás kezdőpozíciója+hossza
      fennsík(L1, P1, F, H)
    ).
```

## MODULARITÁS

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

- Az első fennsík előállítására:

```
% első_fennsík(+L0, +P0, -F, -H, -L): A P0-tól számozott L0 listában az
% első fennsík az F. pozíción van és hossza H, a fennsík után fennmaradó
% rész pedig az L lista.
első_fennsík([E,E|L1], P0, F, H, L) :-
    !, F = P0, azonosak(L1, E, 2, H, L).
első_fennsík(_|L1], P0, F, H, L) :-
    P1 is P0+1,
    első_fennsík(L1, P1, F, H, L).
```

```
% azonosak(+L0, +E, +H0, -H, -L): Az L0 lista elejéről a maximális számú
% E-vel azonos elemet hagyva marad L, a hagyott elemek száma H-H0.
azonosak([X|L0], E, H0, H, L) :-
    E = X, !,
    H1 is H0+1,
    azonosak(L0, E, H1, H, L).
azonosak(L, _, H, H, L).
```

## Modulok definiálása SICStus Prolog nyelven

- A SICStus Prolog modulfogalmának jellemzői:

- Minden modul külön állományba kell kerülnön.

- Az állomány első programeleme egy modul-parancs kell legyen:

```
:- module( ModulNév, [ExpFunktor1, ExpFunktor2, ...]).
```

- *ExpFunktor* = az exportálandó eljárás funkтора (név/argumentumszám)

- Példa:

```
:- module(platók, [fennsík/3]). % plato állomány első sora
```

- Modul-betöltésre szolgáló beépített eljárások:

- use\_module(*ÁllományNév*)

- use\_module(*ÁllományNév*, [*ImpFunktor1*, *ImpFunktor2*, ...])

*ImpFunktor* — az importálandó eljárás funkтора

- *ÁllományNév* lehet névkonstans, vagy pl. library(*KönyvtárNév*):

```
:- use_module(plato). % a fenti modul betöltése
:- use_module(library(lists), [last/2]). % csak last/2 importált
```

- Modulqualifikált hívási forma: *Modul:Hívás* a *Modul*-ban futtatja *Hívás*-t.

- A modulfogalom nem szigorú, egy nem exportált eljárás is meghívható modulqualifikált formában, pl. `platók:első_fennsík(...)`.

## Meta-eljárások modularizált programban

- Eljárásparaméterek átadása gondot okozhat, ha modulközi hívásról van szó:

<pre>modul1.pl állomány: :- module(modul1, [kétszer/1]).  % :- meta_predicate kétszer(:). (*) kétszer(X) :-     X, X.  p :- write(bu).</pre>	<pre>modul2.pl állomány: :- module(modul2, [q/0,r/0]).  :- use_module(modul1).  q :- kétszer(p).  r :- kétszer(modul2:p).  p :- write(ba).</pre>
--	--

- Futtatás:

```
| ?- [modul1,modul2].
| ?- q.    => bubu
| ?- r.    => baba
```

- Automatikus modul-kvalifikáció meta-predikátum deklarációval:

Ha modul1.pl-ben elhagyjuk a (\*)-gal jelzett sor előtti % kommentjelet, akkor

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

## MAGASABBRENDŰ ELJÁRÁSOK

## Meta-predikátum deklaráció, modulnév-kiterjesztés

- Meta-predikátum deklaráció

- Formája:
  - `:- meta_predicate (eljárásnév)((módspec1), ..., (módspecn)), ...`
  - `(módspeci)` lehet `':`, `'+`, `'-`, vagy `'?`.
  - A `':` mód azt jelzi, hogy az adott argumentumot **betöltéskor** ún. modulnév-kiterjesztésnek kell alávetni. (A többi mód hatása azonos, be/kimenő irányt jelezhetünk segítségükkel.)

- Egy *KiF* kifejezés modulnév-kiterjesztése a következő átalakítást jelenti:

- ha *KiF*  $M:X$  alakú, vagy egy olyan változó, amely az adott eljárás fejében meta-argumentum pozíción szerepelt, akkor változatlanul hagyjuk;
- egyébként helyettesítjük *CurMod:KiF*-fel, ahol *CurMod* a kurrens modul.

- Példa folyt. (tfh. a modul1-beli *kétszer* meta-predikátumnak deklarált!)

<pre>:- module(modul2, [négyszer/1,q/0]). :- use_module(modul1). q :- kétszer(p).</pre>	<pre>% tárolt alak: =&gt; q :- kétszer(modul2:p).</pre>
<pre>:- meta_predicate négyszer(:). négyszer(X) :- kétszer(X), kétszer(X).</pre>	<pre>=&gt; változatlan</pre>

## Magasabbrendű eljárások — listakezelés

- Magasabbrendű (vagy meta-eljárás) egy eljárás,

- ha eljárásként értelmezi egy vagy több argumentumát
- pl. `call/1`, `findall/3`, `\+ /1` stb.

- Listafeldolgozás `findall` segítségével — példák

- Páros elemek kiválasztása (vö. Erlang filter)

% Az *L* egész-lista páros elemeinek listája *Pk*.

```
páros_elemei(L, Pk) :- findall(X, (member(X, L), X mod 2 == 0), Pk).
```

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

- A listaelemek négyzetre emelése (vö. Erlang map)

% Az *L* számlista elemei négyzeteinek listája *Nk*.

```
négyzetei(L, Nk) :- findall(Y, (member(X, L), négyzete(X, Y)), Nk).
```

```
négyzete(X, Y) :- Y is X*X.
```

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

## Részlegesen paraméterezett eljáráshívások — segédeszközök

- A `négyzete/0` kifejezés a `négyzete/2` **részlegesen paraméterezett** hívásának tekinthető.
- Ilyen hívások kiegészítésére és meghívására szolgálnak a `call/N` eljárások.
- `call(RPred, A1, A2, ...)` végrehajtása: az `RPred` hívást kiegészíti az `A1, A2, ...` argumentumokkal, és meghívja.
- A `call/N` eljárások SICStus 4-ben már beépítettek, SICStus 3-ban még definiálni kellett ezeket, pl. így:

```
:- meta_predicate call(:, ?), call(:, ?, ?), ....
```

```
% Pred az A utolsó argumentummal meghívva igaz.
call(M:Pred, A) :-
```

```
    Pred =.. FAs0, append(FAs0, [A], FAs1),
    Pred1 =.. FAs1, call(M:Pred1).
```

```
% Pred az A és B utolsó argumentumokkal meghívva igaz.
```

```
call(M:Pred, A, B) :-
    Pred =.. FAs0, append(FAs0, [A,B], FAs2),
    Pred2 =.. FAs2, call(M:Pred2).
```

```
...
```

## Részlegesen paraméterezett eljárások — rekurzív map/3

- A részleges paraméterezés segítségével a `map/3` meta-eljárás rekurzívan definiálható:

```
% map(Xs, Pred, Ys): Az Xs lista elemeire a Pred transzformációt
% alkalmazva kapjuk az Ys listát.
```

```
map([X|Xs], Pred, [Y|Ys]) :-
    call(Pred, X, Y), map(Xs, Pred, Ys).
map([], _, []).
```

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

- Példák:

```
| ?- map([1,2,3,4], négyzete, L).           => L = [1,4,9,16]
| ?- map([1,2,3,4], másodfokú_képe(2,1), L). => L = [4,9,16,25]
```

- A `call/N`-re épülő megoldás előnyei:

- általánosabb és hatékonyabb lehet, mint a `findall`-ra épülő;
- alkalmazható akkor is, ha az elemekre elvégzendő műveletek nem függetlenek, pl. `foldl`.

Rekurzív meta-eljárások — `foldl` és `foldr`

- `foldl(+Xs, :Pred, +Y0, -Y)`: `Y0`-ból indulva, az `Xs` elemeire balról jobbra  
% sorra alkalmazva a `Pred` által leírt kétargumentumú függvényt kapjuk `Y`-t.  
`foldl([X|Xs], Pred, Y0, Y) :-`  
    `call(Pred, X, Y0, Y1), foldl(Xs, Pred, Y1, Y).`  
`foldl([], _, Y, Y).`

```
jegyhozzá(Alap, Jegy, Szam0, Szam) :- Szam is Szam0*Alap+Jegy.
```

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

- `foldr(+Xs, :Pred, +Y0, -Y)`: `Y0`-ból indulva, az `Xs` elemeire jobbról balra  
% sorra alkalmazva a `Pred` kétargumentumú függvényt kapjuk `Y`-t.  
`foldr([X|Xs], Pred, Y0, Y) :-`  
    `foldr(Xs, Pred, Y0, Y1), call(Pred, X, Y1, Y).`  
`foldr([], _, Y, Y).`

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

## DINAMIKUS ADATBÁZISKEZELÉS

## Dinamikus predikátumok

- A dinamikus predikátum jellemzői:
  - a program szövegében lehet 0 vagy több klóza;
  - futási időben hozzáadhatunk és elvehetünk klózokat belőle;
  - végrehajtása mindenképpen interpretált.
- Létrehozása
  - programszövegbeli deklarációval:
 

```
:- dynamic(Eljárásnév/Argumentumszám).
```

 (ha van klóza a programban, akkor az első előtt — ilyenkor kötelező);
  - futási időben, adatbáziskezelő beépített eljárással
- Adatbáziskezelő eljárások („adatbázis” = a program klózainak összessége):
  - klóz felvétele első, utolsó helyre: `asserta/1`, `assertz/1`
  - klóz törlése (illesztéssel, többszörösen sikerülhet): `retract/1`
  - klóz lekérdezése (illesztéssel, többszörösen sikerülhet): `clause/2`
- A klózfelvétel ill. törlés **tartós** mellékhatás, visszalépéskor **nem** áll vissza a korábbi állapot.

## Klóz felvétele: `asserta/1`, `assertz/1`

- `asserta(:@Klóz)`
  - A Klóz kifejezést klózként értelmezve felveszi a programba az adott predikátum *első* klózáként. A Klózban levő változók szisztematikusan újakra cserélődnek.
  - A '@' mód jelentése: tisztán bemenő paraméter, az eljárás a paraméterbeli változókat nem helyettesíti be (a '+' mód speciális esete).
  - A ':' mód modul-kvalifikált paramétert jelez.
- `assertz(:@Klóz)`
  - Ugyanaz mint `asserta`, csak a Klóz kifejezést az adott predikátum *utolsó* klózáként veszi fel.
- Példa:
 

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

```
| ?- assert(s(X,X)), s(U,V), U == V, X \== U.
      V = U ? ; no
```

## Klóz törlése: `retract/1`

- `retract(:@Klóz)`
  - A Klóz klóz-kifejezésből megállapítja a predikátum funktorát.
  - Az adott predikátum klózeit sorra megpróbálja illeszteni Klóz-zal.
  - Ha az illesztés sikerült, akkor kitörli a klózt és sikeresen lefut.
  - Visszalépés esetén folytatja a keresést (illeszt, töröl, sikerül stb.)
- Példa (folytatás):
 

```
| ?- listing(p), retract((p(2,_):-_)), listing(p), fail. => no
```
- A futás kimenete:
 

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

## Alkalmazási példa — egyszerűsített `findall`

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

```
:- dynamic(megoldás/1).

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

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

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

## Klóz lekérdezése: clause/2

- `clause(:@Fej, ?Törzs)`
  - A `Fej` alapján megállapítja a predikátum funktorát.
  - Az adott predikátum klózeit sorra megpróbálja illeszteni a `Fej :- Törzs` kifejezéssel (tényállítás esetén `Törzs = true`).
  - Ha az illesztés sikerült, akkor sikeresen lefut.
  - Visszalépés esetén folytatja a keresést (illeszt, sikerül stb.)

### • Példa:

```
:- listing(p), clause(p(2, 0), T).
```

```
p(2, 0).
p(1, A) :-
    q(A).
p(2, A) :-
    r(A).
T = true ? ;
T = r(0) ? ;
no
```

## A clause eljárás alkalmazása: egyszerű nyomkövető interpreter

- Az alábbi interpreter csak „tisztá”, beépített eljárást nem alkalmazó Prolog programok futtatására alkalmas.

```
% interp(G, D): A G cél futását D bekezdésű nyomkövetéssel mutatja.
interp(true, _) :- !.
interp((G1, G2), D) :- !,
    interp(G1, D), interp(G2, D).
interp(G, D) :-
    ( trace(G, D, call)
    ; trace(G, D, fail), fail % követi a fail kaput, tovább-hiúsul
    ),
    D2 is D+2,
    clause(G, B), interp(B, D2),
    ( trace(G, D, exit)
    ; trace(G, D, redo), fail % követi a redo kaput, tovább-hiúsul
    ).

% A G cél áthaladását a Port kapun D bekezdésű nyomkövetéssel mutatja.
trace(G, D, Port) :-
    /*D szókózt ír ki:*/ tab(D),
    write(Port), write(' '), write(G), nl.
```

## Nyomkövető interpreter - példafutás

```
:- dynamic app/3, app/4. % (*)
app([], L, L).
app([X|L1], L2, [X|L3]) :-
    app(L1, L2, L3).

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

% A (*) sor elhagyható, ha a fenti
% (mondjuk app34) állományt az
% alábbi (SICStus-specifikus)
% beépített eljárással töltjük be:
?- load_files(app34,
    compilation_mode(
        assert_all)).

?- interp(app(_, [b,c], L, [c,b,c,b]), 0).
call: app(_203, [b,c], _253, [c,b,c,b])
call: app(_203, _666, [c,b,c,b])
exit: app([], [c,b,c,b], [c,b,c,b])
call: app([b,c], _253, [c,b,c,b])
fail: app([b,c], _253, [c,b,c,b])
redo: app([], [c,b,c,b], [c,b,c,b])
call: app(_873, _666, [b,c,b])
exit: app([], [b,c,b], [b,c,b])
exit: app([c], [b,c,b], [c,b,c,b])
call: app([b,c], _253, [b,c,b])
call: app([c], _253, [c,b])
call: app([], _253, [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] ?
```

## NYELVTANI ELEMZÉS PROLOGBAN

## Egy egyszerű nyelvtani elemzési példa

- Bináris számok nyelvtana

```

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

```

- Ugyanez DCG (Definite Clause Grammar) jelöléssel:

```

szám --> számjegy, számmaradék.
számmaradék --> számjegy, számmaradék | "".
számjegy --> "0" | "1".

```

- A definit klóz nyelvtan (DCG):

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

## Nyelvtani elemzés „bevetítése” Prologba

- Nyelvtani elemzés: annak eldöntése, hogy egy (Prolog listában tárolt) jelsorozat megfelel-e egy adott nem-terminális nyelvtani fogalomnak.

- A lista tetszőleges elemekből állhat, pl. karakterkódok listája, lexikai elemek (token-ek) listája.

- A nem-terminálisoknak kétargumentumú Prolog szabályok felelnek meg, pl.

```

szám --> számjegy, számmaradék.
szám(L0, L) :- számjegy(L0, L1), számmaradék(L1, L).
% Az L0 kódlistáról "leelemezhető" egy <szám>, marad L ha
% L0-ról leelemezhető egy <számjegy>, marad L1, és
% L1-ről leelemezhető egy <számmaradék>, marad L.

```

- Általánosan: az adott nem-terminálisnak megfelelő jelsorozatot „leelemezve” (lehagyva) egy L0 lista elejéről marad egy L lista.

- Terminális szimbólumok esetén egyetlen elemet kell leahagyni a listáról, erre szolgál a 'C' / 3 beépített eljárás. Definíciója: 'C'(L0, X, L) :- L0 = [X|L]. (A SICStus fordító a 'C' / 3 hívást ténylegesen a fenti egyenlőséggel helyettesíti.)

- A „leelemezés” tulajdonképpen akkumulációs folyamat, ahol az elemi akkumulációs lépés: egy terminális leahagyása a lista elejéről ('C' / 3).

## A DCG szabályok lefordított alakja

- A korábbi DCG példa:

```

szám --> számjegy, számmaradék. % A | B ≡ A ; B
számmaradék --> számjegy, számmaradék | "". % "" ≡ []
számjegy --> "0" | "1". % "0" ≡ [48]

```

- A fenti DCG szabályok betöltésekor a következő Prolog kód keletkezik:

```

szám(L0, L) :-
    számjegy(L0, L1), számmaradék(L1, L).

számmaradék(L0, L) :-
    ( számjegy(L0, L1), számmaradék(L1, L)
    ; L = L0
    ).

számjegy(L0, L) :-
    ( 'C'(L0, 48, L)
    ; 'C'(L0, 49, L)
    ).

```

- A DCG elemző futtatása:

```

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

```

## Vezérlési szerkezetek DCG szabályokban

- DCG szabályokban használható: vágó, diszjunkció, negáció és feltételes diszjunktív szerkezet.

- Ezek változtatás nélkül átkerülnek a Prolog alakba. Példák:

```

% Leelemezhető számjegyek egy MAXIMÁLIS (esetleg üres) listája.
számmaradék -->
    ( számjegy -> számmaradék
    ; [] % Vigyázat: [] helyett true nem jó!
    ).

% Ugyanez vágóval
számmaradék --> számjegy, !, számmaradék.
számmaradék --> []. % Figyelem: nincsenek DCG tényállítások!

% Az utóbbi Prolog alakja:
számmaradék(L0, L) :-
    számjegy(L0, L1), !, számmaradék(L1, L).
számmaradék(L0, L) :-
    L = L0.

```

```

| ?- számmaradék("102", L). => L = "2" ; no

```

## Prolog hívás beillesztése DCG szabályba

- Általánosabb példa: decimális számjegyek elemzése

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

% Ugyanez általánosabban és egyszerűbben:
számjegy -->
  [K],                % K a következő terminális
  {decimális_jegy_kódja(K)}. % Prolog hívás

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

- A fenti DCG szabály Prolog megfelelője:

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

## Az elemző kiegészítése argumentumokkal

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

- Példa: szám elemzése és értékének kiszámítása:

```
% leelemezhető egy Sz értékű decimálisszámjegy-sorozat
szám(Sz) --> számjegy(J), számmaradék(J, Sz).

% leelemezhető számjegyek egy esetleg üres listája, amelynek
% az eddig leelemezett Sz0-val együtt vett értéke Sz.
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_jegy_kódja(K), J is K-0'0}.

| ?- szám(Sz, "102 56", L). => L = " 56", Sz = 102; no
```

- A számmaradék DCG szabály Prolog alakja:

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

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

## A DCG nyelvtani szabályok szerkezete — összefoglalás

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

```
p(A,...) -->
  q0(B,...), ..., [X],          qi(C,...), ...,
  {Cél}, ..., qn(D,...).

p(A,...,L0,L):-
  q0(B,...,L0,L1), ..., 'C'(Li-1, X, Li), qi(C,...,Li,Li+1), ...,
  Cél, ..., qn(D,...,Ln,L).
```

## DCG példa: kifejezés kiértékelése

- Egyszerű aritmetikai kifejezés elemzése és kiértékelése.

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

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

| ?- kif(Z, "10*10-6*6", ""). => Z = 64 ; no
| ?- kif(Z, "10*10-6*6", L). => L = [], Z = 64 ; L = "*6", Z = 94 ; ...
| ?- kif(Z, "4-2+1", []). => Z = 1 Probléma: jobbról balra eleméz!
```

- Egy lehetséges javítás

```
kif(Z) --> tag(X), kifmaradék(X, Z).

kifmaradék(X, Z) --> "+", tag(Y), W is X + Y, kifmaradék(W, Z).
kifmaradék(X, Z) --> "-", tag(Y), W is X - Y, kifmaradék(W, Z).
kifmaradék(X, X) --> [].

...
```



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

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

% mondat(Alany, Áll, L0, L): L0-L kielemezhető egy Alany alanyból és Áll
% állítmányból álló mondat. Alany lehet első vagy második személyű
% névmás, vagy egyetlen szóból álló (harmadik személyű) alany.
mondat(Alany, Áll) -->
    {én_te(Alany, Ige)}, én_te_perm(Alany, Ige, Áll).
mondat(Alany, Áll) -->
    szó(Alany), szavak(Áll).

% én_te(Alany, Ige):
% Az Alany első/második személyű névmásnak megfelelő létige az Ige.
én_te("én", "vagyok").
én_te("te", "vagy").

% én_te_perm(Ki, Ige, Áll, L0, L): L0-L kielemezhető egy Ki
% névmásból, Ige igealakból és Áll állítmányból álló mondat.
én_te_perm(Alany, Ige, Áll) -->
    ( szó(Alany), szó(Ige), szavak(Áll)
    ; szó(Alany), szavak(Áll), szó(Ige)
    ; szavak(Áll), szó(Ige), szó(Alany)
    ; szavak(Áll), szó(Ige)
    ).
```

Deklaratív programozás. BME VIK, 2010. őszi félév

(Logikai Programozás)

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

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

% számaradék(Sz, L0, L): L0-L egy Sz kódlistából álló (esetleg üres) szó.
számaradék([B|Sz]) -->
    betű(B, !), számaradék(Sz).
számaradék([]) --> [].

% illik(Szó0, Szó): Szó0 = Szó, vagy a kezdő kis-nagy betűben különböznek.
illik([B0|L], [B|L]) :-
    ( B = B0 -> true
    ; abs(B-B0) =:= 32
    ).

% köz(L0, L): L0-L nulla, egy vagy több szóköz.
köz --> ( " " -> köz ; " " ).

% betű(K, L0, L): L0-L egy K kódú "betű" (különbözik a " ." jelektől)
betű(K) --> [K], {\+ member(K, " .?")}.

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

Deklaratív programozás. BME VIK, 2010. őszi félév

(Logikai Programozás)

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

```
% :- type mondás ---> kérdez(szó) ; kijelent(szó,list(szó)) ; un.

% Megvalósít egy párbeszédet.
párbeszéd :-
    repeat,
        read_line(L, % beolvas egy sort, L a karakterkódok listája
        ( menet(Mondás, L, [])
        -> feldolgoz(Mondás)
        ; write('Nem értem\n'), fail
        ),
    Mondás = un, !.

% menet(Mondás, L0, L): Az L0-L kielemezett alakja Mondás.
menet(kérdez(Alany)) -->
    {kérdő(Szó)}, mondat(Alany, [Szó]), "?".
menet(kijelent(Alany,Áll)) -->
    mondat(Alany, Áll), ".".
menet(un) -->
    szó("unlak"), ".".

% kérdő(Szó): Szó egy kérdőszó.
kérdő("mi").
kérdő("ki").
kérdő("kicsoda").
```

Deklaratív programozás. BME VIK, 2010. őszi félév

(Logikai Programozás)

## Példa: „természetes” nyelvű beszélgetés — válaszok előállítás

```
:- dynamic tudom/2.

% feldolgoz(Mondás): feldolgozza a felhasználtól érkező Mondás üzenetet.
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').

% Felsorolja az Alany ismert tulajdonságait.
válasz(Alany) :-
    tudom(Alany, Áll),
    ( member(Szó, Áll), format('~s ', [Szó]), fail
    ; nl
    ),
    fail.
válasz(_).
```

Deklaratív programozás. BME VIK, 2010. őszi félév

(Logikai Programozás)

## 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.
|: Ki vagy te?
egy Prolog program
Okos
|: Valóban?
Nem értem
|: Unlak.
Én is.

```

## „HAGYOMÁNYOS” BEÉPÍTETT ELJÁRÁSOK

## A DCG formalizmus felhasználása elemzésen kívül

### • A DCG szabályok kényelmesen használhatók általános akkumulálásra

#### • Listák akkumulálása — az elemi akkumulálási lépést a 'C'/3 adja

```

% anbn(+N, ?L): Az L lista N db a-ból és azt követő N db b-ből áll.
% Nem csak elemzésre, hanem L felépítésére is használható!
anbn(N, L) :- anbn(N, L, []).

```

```

% anbn(N, L0, L): L0-L N db a-ból és azt követő N db b-ből áll.
anbn(0) --> !.
anbn(N) --> {N > 0, N1 is N-1}, [a], anbn(N1), [b].

```

% a fenti DCG szabály kifejtve:

```

anbn(N, L0, L) :-
    N > 0, N1 is N-1, L0=[a|L1], anbn(N1, L1, L2), L2=[b|L].

```

#### • Egyébként az elemi akkumulálási lépést DCG-n kívül kell megírni:

```

% sum(L, S0, S): L összege S-S0.
sum([]) --> [].
sum([X|L]) -->
    plus(X), sum(L).

```

```

% L számlista összege S.
sum(L, S) :- sum(L, 0, S).
plus(X, S0, S) :- S is S0+X.

```

## Aritmetikai beépített eljárások

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

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

Függvény jelölésűek			
abs/1	exp/1	floor/1	sign/1
atan/1	float/1	log/1	sin/1
ceiling/1	float_fractional_part/1	max/2,min/2	sqrt/1
cos/1	float_integer_part/1	round/1	truncate/1

## Listakezelő beépített eljárások

- Lista hossza: `length(?L, ?N)`
  - Jelentése: az L lista hossza N.
  - `length(-L, +N)` módban adott hosszúságú, csupa különböző változóból álló listát hoz létre.
  - `length(-L, -N)` módban rendre felsorolja a 0, 1, ... hosszú listákat.
  - Megvalósítását lásd korábban.
- Lista rendezése: `sort(@L, ?S)`
  - Jelentése: az L lista @< szerinti rendezése S, (= = / 2 szerint azonos elemek ismétlődését kiszűrve).
- Lista kulcs szerinti rendezése: `keysort(@L, ?S)`
  - Az L argumentum Kulcs-Érték alakú kifejezések listája.
  - Az eljárás jelentése: az S lista az L lista Kulcs értékei szerinti szabványos (@< általi) rendezése, ismétlődéseket nem szűr.

## Kifejezések kiírása

- `write(@X)`: Kiírja X-et, ha szükséges operátorokat, zárójeleket használva.
- `writeln(@X)`: Mint `write(X)`, csak gondoskodik, hogy szükség esetén az névkonstansok idézőjelek közé legyenek téve.
- `write_canonical(@X)`: Mint `writeln(X)`, csak operátorok nélkül, minden struktúra szabványos alakban jelenik meg.
- `write_term(@X, +Opciók)`: Az Opciók opciólista szerint kiírja X-et.
- `format(@Formátum, @AdatLista)`: A Formátum-nak megfelelő módon kiírja AdatLista-t. A formázójelek alakja: ~{szám esetleg}<formázójel>.

```
| ?- write('Helló világ').           => Helló világ
| ?- writeln('Helló világ').         => 'Helló világ'
| ?- write_canonical('*' - '%').    => -(*, '%')
| ?- write_canonical([1,2]).        => '.(1,).(2,[])
| ?- write_term([1,2,3], [max_depth(2)]). => [1,2|...]
| ?- format('X==s --- ~3d s', [[0'j,0'ó],3245]). => X=jó --- 3.245 s
```

## Kifejezések kiírása — felhasználó vezérelte formázás

- `print(@X)`: Alapértelmezésben azonos `writeln`-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, meghíúsulás esetén maga írja ki a részkifejezést.  
A rendszer a `print` eljárást használja a változó-behelyettesítések és a nyomkövetés kiírására!
- `portray(@Kif)` (felhasználó által definiálandó ún. *kampó eljárás*): Igaz, ha Kif kifejezést a Prolog rendszernek *nem* kell kiírnia (és ekkor maga a `portray` kell, hogy elvégezze a kiírást).
- Példa:

```
portray(Matrix) :-
  Matrix = [[_]|_]|_,
  ( member(Row, Matrix),
    nl, print(Row), fail
  ; true
  ).
| ?- X = [[1,2],[3,4],[5,6]].
X =
[1,2]
[3,4]
[5,6] ?
```

## Karakterek kiírása és beolvasása

- `put_code(+Kód)`: Kiírja az adott kódú karaktert.
- `tab(+N)`: Kiír N szóközt feltéve, hogy  $N > 0$ .
- `nl`: Kiír egy soremelést.
- `get_code(?Kód)`: Beolvas egy karaktert és (karakterkódját) egyesíti Kód-dal. (File végénél  $Kód = -1$ .)
- `peek_code(?Kód)`: A soronkövetkező karakter kódját egyesíti Kód-dal. A karaktert nem távolítja el a bemenetről. (File végénél  $Kód = -1$ .)
- Példa:

```
% rd_line(L): L a következő sor karakterkódjainak listája.
% read_line néven beépített eljárás SICStus 3.9.0-től.
rd_line(L) :-
  peek_code(0'\n), !, get_code(_), L = [].
rd_line([C|L]) :-
  get_code(C), rd_line(L).
| ?- rd_line(L), tab(20), member(X, L), put_code(X), tab(1), fail ; nl.
|: Hello world!
      H e l l o   w o r l d !
```

## Példa: számbeolvasás

```
% számbe(Szám): a Szám szám következik az input-folyamban.
```

```
számbe(Szám) :-
    számjegy(Érték),
    számbe(Érték, Szám).
```

```
% Az eddig beolvasott Szám0-val együtt az input-folyamban következő
```

```
% szám értéke Szám.
```

```
számbe(Szám0, Szám) :-
    számjegy(E), !,
    Szám1 is Szám0*10+E,
    számbe(Szám1, Szám).
```

```
számbe(Szám, Szám).
```

```
% Érték értékű számjegy következik.
```

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

```
| ?- számbe(X), get_code(_), számbe(Y).
```

```
|: 123 456
```

```
⇒ X = 123, Y = 456
```

## Kifejezések beolvasása

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

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

| ?- listing([p/1]).
p(A) :-
    q(A),
    r(A).
yes

| ?- consult_body.
|: p(X) :- q(X), r(X).
|: ^D
yes
```

## Be- és kiviteli csatornák

- Csatornák megnyitása és kezelése:

- `open(@Filenév, @Mód, -Csatorna)`: Megnyitja a `Filenév` nevű állományt `Mód` módban (`read`, `write` vagy `append`). A `Csatorna` argumentumban visszaadja a megnyitott csatorna „nyelét”.
- `set_input(@Csatorna), set_output(@Csatorna)`: Az ezt követő beviteli/kiviteli eljárások `Csatorna`-t használják majd (jelenlegi csatorna).
- `current_input(?Csatorna), current_output(?Csatorna)`: A jelenlegi beviteli/kiviteli csatornát egyesíti `Csatorna`-val.
- `close(@Csatorna)`: Lezárja a `Csatorna` csatornát.

- Explicit csatornamegadás be- és kiviteli eljárásokban

- Az eddig ismertett összes be- és kiviteli eljárásnak van egy eggyel több argumentumú változata, amelynek első argumentuma a csatorna. Ezek: `write/2`, `writeq/2`, `write_canonical/2`, `write_term/3`, `print/2`, `read/2`, `read_term/3`, `format/3`, `put_code/2`, `tab/2`, `nl/1`, `get_code/2`, `peek_code/2`.

## Egy egyszerűbb be- és kiviteli szervezés: DEC10 I/O

- `see(@Filenév), tell(@Filenév)`: Megnyitja a `Filenév` file-t olvasásra/írásra és a jelenlegi csatornává teszi. Újabb híváskor csak a jelenlegi csatornává teszi.
- `seeing(?Filenév), telling(?Filenév)`: A jelenlegi beviteli/kiviteli csatorna állománynevét egyesíti `Filenév`-vel.
- `seen, told`: Lezárja a jelenlegi beviteli/kiviteli csatornát.
- Példák — nagyon egyszerű `consult` variánsok:

```
consult_dec10_style(File) :-
    seeing(Old), see(File),
    repeat,
        read(Term),
        ( Term = end_of_file
        -> seen
        ; assertz(Term), fail
        ),
    !,
    see(Old).

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

## Hibakezelési beépített eljárások

- Hibahelyzetet beépített eljárás rossz argumentumokkal való meghívása, vagy a `throw/1` (`raise_exception/1`) eljárás válthat ki.
- Minden hibahelyzetet egy Prolog kifejezés (ún. hiba-kifejezés) jellemez.
- Hiba „dobása”, azaz a `HibaKif` hibahelyzet kiváltása:
 

```
throw(@HibaKif),
raise_exception(@HibaKif)
```
- Hiba „elkapása”:
 

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

  - Hatása: Futtatja a `Cél` hívást.
    - Ha `Cél` végrehajtása során hibahelyzet nem fordul elő, futása azonos `Cél`-lal.
    - Ha `Cél`-ban hiba van, a hiba-kifejezést egyesíti `Mintá`-val.
    - Ha ez sikeres, meghívja a `Hibaág`-at.
    - Ellenkező esetben továbbdobja a hiba-kifejezést, hogy a további körülvevő `catch` eljárások esetleg elkaphassák azt.

## Programfejlesztési beépített eljárások (SICStus specifikusak)

- `set_prolog_flag(+Jelző, @Érték)`: Jelző értékét `Érték`-re állítja.
- `current_prolog_flag(?Jelző, ?Érték)`: Jelző pillanatnyi értéke `Érték`.
- Néhány fontos Prolog jelző:
  - `argv`: csak olvasható, a parancssorbeli argumentumok listája.
  - `unknown`: viselkedés definiálatlan eljárás hívásakor (`trace`, `fail`, `error`).
  - `source_info`: forrásszintű nyomkövetés (`on`, `off`, `emacs`).
- `consult(:@Files), [:@File, ...]`: Betölti a `File`(ok)at, interpretált alakban.
- `compile(:@File)`: Betölti a `File`(ok)at, lefordított alakot hozva létre.
- `listing`: Kírja az összes interpretált eljárást az aktuális kimenetre.
- `listing(:@EljárásSpec)`: Kírja a megnevezett interpretált eljárásokat.
- Itt és később: `EljárásSpec` — név vagy funktor, eseteg modul-kvalifikációval ellátva, ill. ezek listája, pl. `listing(p)`, `listing([m:q,p/1])`.

## Programfejlesztési eljárások (folytatás)

- `statistics`: Különféle statisztikákat ír ki az aktuális kimenetre.
- `statistics(?Fajta, ?Érték)`: `Érték` a `Fajta` fajtájú mennyiség értéke.
  - Példa: `statistics(runtime, E) => E=[Tdiff, T]`, `Tdiff` az előző lekérdezés óta, `T` a rendszerindítás óta eltelt idő, ezredmásodpercben.
- `break`: Egy új interakciós szintet hoz létre.
- `abort`, `halt`: Kilép a legkülső interakciós szintre ill. a Prolog rendszerből.
- `trace`: Elindítja az interaktív nyomkövetést.
- `debug`, `zip`: Elindítja a szelektív nyomkövetést, csak spion-pontoknál áll meg. (A `zip` mód gyorsabb, de nem gyűjt annyi információt mint a `debug` mód.)
- `nodebug`, `notrace`, `nozip`: Leállítja a nyomkövetést.
- `spy(:@EljárásSpec)`: Spion-pontot tesz a megadott eljárásokra.
- `nospy(:@EljárásSpec)`: Megszünteti a megadott spion-pontokat.
- `nospyall`: Az összes spion-pontot megszünteti.

## FEJLETTEBB NYELVI ÉS RENDSZERELEMEK

## Külső nyelvi interfész

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

## Külső nyelvi interfész — példa

- A példa a `library(bdb)` megvalósításából származik.
- A C nyelven megírandó eljárás Prolog hívási alakja:  
`index_keys(+Spec, +Kif, -Kulcs, -Szám)`
- A megírandó eljárás jelentése:
  - Ha *Spec* és *Kif* különböző funktorú kifejezések, akkor *Szám* = -1 és *Kulcs* = [].
  - Egyébként, ha *Spec* valamelyik argumentuma + és *Kif* megfelelő argumentuma változó, akkor *Szám* = -2 és *Kulcs* = [].
  - Egyébként *Szám* a *Spec* argumentumaként előforduló + névkonstansok száma, *Kulcs* pedig *Kif* megfelelő argumentumainak *kivonatából* képzett lista. A kivonat lényegében az argumentum funktora, azzal az eltéréssel, hogy a konstansok kivonata maga a konstans, struktúrák esetén pedig a struktúra neve és az aritása külön elemként kerül a kivonat-listába.

## Külső nyelvi interfész — példa

- A példaeljárás használata
 

```
| ?- [ixtest].
| ?- index_keys(f(+, -, +, +),
               f(12.3, _, s(1, _, z(2)), t),
               Kulcs, Szam).
Kulcs = [12.3,s,3,t], Szam = 3 ?
```
- Az `ixtest.pl` Prolog file tartalmazza az interfész specifikációját:
 

```
foreign(ixkeys, index_keys(+term, +term, -term, [-integer])).
      % 1. arg: bemenő, általános kifejezés
      % 2. arg: bemenő, általános kifejezés
      % 3. arg: kimenő, általános kifejezés
      % 4. arg: a C függvény értéke, egész (long)

foreign_resource(ixkeys, [ixkeys]).

:- load_foreign_resource(ixkeys).
```
- A C programot elő kell készíteni a Prolog számára az `splfr` (link foreign resource) eszköz segítségével:
 

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

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

```
#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 check */
        if (t != plus) continue;

        SP_get_arg(i, term, arg);
        switch (SP_term_type(arg)) {
            case SP_TYPE_VARIABLE:
                return NI;
            case SP_TYPE_COMPOUND:
                SP_get_functor(arg, &tname, &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(40,F).
```

```
F = 815915283247897734345611269596115894272000000000 ?
```

- 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 névkonstans 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 visszacsinalódik. `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 behelyettesíthető változó (blokkolási feltétel), akkor a `p` hívás felfüggesztődik.

Ugyanarra az eljárásra több vagylagos feltétel is szerepelhet, pl.

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

- Végtelen választási pontok kiküszöbölése blokk-deklarációval

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

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

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

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

## Blokk-deklarációk (folytatás)

- Generál-és-ellenőriz típusú programok gyorsítása

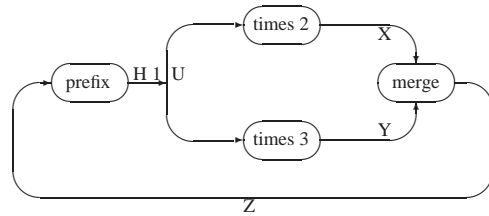
- általában nem hatékonyak (pl `megrajzolja_1`), mert túl sok visszalépést használnak
- korutinszervezéssel a generáló és ellenőrző rész „automatikusan” összefűsülhet
- ehhez az ellenőrző részt kell előre tenni és megfelelően blokkolni

- 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)

Hivast felfüggeszti mindaddig, amíg X behelyettesíthető 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, amíg 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)))]
| ?- call_residue((dif(X, f(Y)), X=f(Z)), Maradek).
    => X = f(Z), Maradek = [[Y,Z]-(prolog:dif(f(Z),f(Y)))]
```

## SICStus könyvtárak

- Könyvtár betöltése

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

- A legfontosabb könyvtárak

- arrays Logaritmikus elérési idejű kiterjeszhető tömbök megvalósítását tartalmazza.
- assoc AVL fák segítségével valósítja meg az „asszociációs listák”, azaz véges Prolog kifejezeshalmazokon definiált kiterjeszhető leképezések fogalmát.
- atts tetszőleges attributumokat enged a Prolog változókhoz rendelni, ezeket tárolórekeszként és a Prolog egyesítési mechanizmusának módosítására is 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 (halmaz  $\equiv$  @< szerint rendezett lista).
- queues Sorokra (queue, FIFO store) vonatkozó műveleteket definiál.
- random Egy véletelenszám-generátort tartalmaz.



- `system` Különbféle operációsrendszer-szolgáltatások elérését biztosítja.
- `trees` Az `arrays` könyvtárhoz hasonló, de nem-kiterjeszthető logaritmikus elérési idejű tömbfogalmat valósít meg, bináris fákkal (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.
- `bdb` Felhasználó által definiált többszörös indexelést lehetővé tevő, Prolog kifejezések állományokban 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).
- `tcltk` A *Tcl/Tk* nyelv és eszközkészlet elérését biztosítja.
- `gauge` Prolog programok a profilozására szolgáló, a `tcltk` -n alapuló grafikus eszköz.
- `charsio` Karakter sorozatból olvasó ill. abba író be- és kiviteli eljárások gyűjteménye.
- `timeout` Lehetőséget ad arra, hogy célok futási idejét korlátozzuk.

## ÚJ IRÁNYZATOK A LOGIKAI PROGRAMOZÁSBAN

### Új irányzatok a logikai programozásban — kitekintés

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

## KORLÁT-LOGIKAI PROGRAMOZÁS – RÖVID ÁTTEKINTÉS

## A korlát-logikai programozás (CLP, Constraint Logic Programming) alapfogalata

### • 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

- $\mathcal{X} = \mathbb{Q}$  vagy  $\mathbb{R}$  (a racionális vagy valós számok)  
korlátok = lineáris egyenlőségek és egyenlőtlenségek  
következtetési mechanizmus = Gauß elimináció és szimplex módszer
- $\mathcal{X} = \text{FD}$  (egész számok Véges Tartománya, angolul FD — Finite Domain)  
korlátok = különféle aritmetikai és kombinatorikus relációk  
következtetési mechanizmus = MI CSP-módszerek (CSP = Korlát-Kielégítési Probléma)
- $\mathcal{X} = \text{B}$  (0 és 1 Boole értékek)  
korlátok = ítéletkalkulusbeli relációk  
következtetési mechanizmus = MI SAT-módszerek (SAT — Boole kielégíthetőség)

## A CLP következtetés alapelvei

### • A CLP következtetés

- közege az ún. korlát-tár, amelyben a korlátok gyűlnek, egyre pontosabban közelítve a megoldást;
- elemei az ún. primitív korlátok (a megengedett korlátok egy részhalmaza)
- a korlát-tár mindig konzisztens, ellentmondás esetén visszalépés;
- visszalépés esetén a korlát-tár is visszaáll a korábbi állapotba
- a következtetés fajtái:
  - **teljes**, pl. CLP( $\mathbb{R}$ ) lineáris esetben, CLP( $\text{B}$ ) — minden korlát bekerül a tárba;
  - **részleges**, pl. CLP( $\text{FD}$ ) — csak bizonyos egyszerű korlátok mennek a tárba, a többi, nem-primitív korlátok ágensként (démonként) várakoznak arra, hogy:
    - a primitív korláttá váljanak
    - a tárat egy primitív korláttal bővíthessék (az ún. erősítés)

## A SICStus clp(Q,R) könyvtárak

### • Alapelemek

- Tartomány:  
clpr: lebegőpontos számok, clpq: racionális számok
- Függvények:  
+ - \* / min max pow exp (kétargumentumúak, pow  $\equiv$  exp),  
+ - abs sin cos tan (egyargumentumúak).
- Korlát-relációk: = ::= < > =< >= \= (=  $\equiv$  ::=)
- Primitív korlátok (a korlát-tár elemei): lineáris kifejezéseket tartalmazó relációk
- Megoldó algoritmus: lineáris programozási módszerek (Gauss elimináció, szimplex módszer)

### • A könyvtár betöltése:

use\_module(library(clpq)), vagy use\_module(library(clpr))

### • A fő beépített eljárás

- { *Constraint* }, ahol *Constraint* változókból és (egész vagy lebegőpontos) számokból a fenti műveletekkel felépített reláció, vagy ilyen relációknak a (, operátorral képzett) konjunkciója.

## Példák a SICStus clpq könyvtárának használatára

```
| ?- use_module(library(clpq)).
| ?- {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<29/5}, {Y= -2+2*X}, {Z=9-1/2*X} ?
| ?- {(Y+X)*(X+Y)/X = Y*Y/X+100}.      % lineárisra 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} ?
| ?- {exp(X+Y+1,2) = 3*X*X+Y*Y}.       % nem lineáris...
      clpq:{1+2*X+2*(Y*X)-2*X^2+2*Y=0} ?
| ?- {exp(X+Y+1,2) = 3*X*X+Y*Y}, X=Y.  % így már igen...
      X = -1/4, Y = -1/4 ?
| ?- {2 = exp(8, X)}.                   % nem-lineárisak is megoldhatók
      X = 1/3 ?
```

## A SICStus clpb könyvtár

## ● Alapelemek:

- **Tartomány:** logikai értékek (1 és 0, igaz és hamis)
- **Függvények** (egyben korlát-relációk):
  - $\sim P$   $P$  hamis (*negáció*).
  - $P * Q$   $P$  és  $Q$  mindegyike igaz (*konjunkció*).
  - $P + Q$   $P$  és  $Q$  legalább egyike igaz (*diszjunkció*).
  - $P \# Q$   $P$  és  $Q$  pontosan egyike igaz (*kizáró vagy*).
  - $P ::= Q$  Ugyanaz mint  $\sim(P \# Q)$ .
- **Constraint-megoldó algoritmus:** Boole-egyesítés.

## ● A library(clpb) könyvtár eljárásai

- `sat` (*Kifejezés*), ahol *Kifejezés* változókból, a 0 1 számkonstansokból és névkonstansokból (ún. szimbolikus konstansok) a fenti műveletekkel felépített logikai kifejezés. Hozzáveszi *Kifejezést* a korlát-tárhoz.
- `labeling` (*Változók*). Behelyettesíti a *Változókat* 0 1 értékekre, úgy, hogy a tár teljesüljön. Visszalépéskor felsorolja az összes lehetséges értéket.

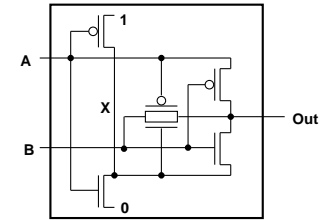
## Példa a clpb könyvtár használatára: tranzisztoros áramkör verifikálása

```
n(D, G, S) :- % Gate => Drain = Source
    sat( G*D ::= G*S ).

p(D, G, S) :- % ~ Gate => Drain = Source
    sat( ~G*D ::= ~G*S ).

xor(A, B, Out) :-
    p(1, A, X),          n(B, X, Out),
    n(0, A, X),          p(A, B, Out),
    p(B, A, Out),        n(X, B, Out).

| ?- n(D, 1, S).          S = D ?
| ?- n(D, 0, S).         true ?
| ?- p(D, 0, S).         S = D ?
| ?- p(D, 1, S).         true ?
| ?- xor(a, b, X).       sat(X::=a#b) ?
```



## A SICStus clpfd könyvtár

## ● A clpfd könyvtár alapelemei

- Tartomány: egészek (negatívak is!)
- Függvények (aritmetika): + - \* / ...
- Constraint-relációk
  - aritmetikaiak:** #<, #>, #=<, #>=, #= #\=
  - halmazműveletek:** X in Halmaz, pl. X in 1..5
  - logikai műveletek:** #/\, #\/, #\ (negáció), #<=> (ekvivalencia), ...
- egyszerű korlátok (korlát tár elemei): X in Halmaz
- Constraint-megoldó algoritmus:
  - aritmetikaiak:** ún. intervallum-konzisztencia (csak a határokat szűkítik)
  - halmazműveletek:** teljes konzisztencia (ún. tartomány-konzisztencia)

## ● A tipikus CLP(FD) megoldási folyamat (forrás: CSP = Constraint Satisfaction Problems)

- a változók tartományának megadása
- korlátok felvétele
- címkézés (visszalépéses keresés) — pl. a labeling(Opciók, Változók) könyvtári eljárás segítségével.

## Példa a clpfd könyvtár használatára: N királynő a sakkasztalán

```
% A Qs lista N királynő biztonságos elhelyezését mutatja egy N*N-es sakkasztalán:
% a lista i. eleme j ==> az i. királynőt az i. sor j. oszlopába kell helyezni.
queens(N, Qs):-
    length(Qs, N), domain(Qs, 1, N), safe(Qs).

% safe(Qs): A Qs királynő-lista biztonságos.
safe([]).
safe([Q|Qs]):-
    no_attack(Qs, Q, 1), safe(Qs).

% no_attack(Qs, Q, I): A Qs lista által leírt királynők egyike sem támadja a
% Q oszlopban levő királynőt, feltéve hogy Q és Qs távolsága I.
no_attack([],_,_).
no_attack([X|Xs], Y, I):-
    no_threat(X, Y, I), J is I+1, no_attack(Xs, Y, J).

% Az X és Y oszlopokban I sortávolságra levő királynők nem támadják egymást.
no_threat(X, Y, I) :-
    Y #\= X, Y #\= X-I, Y #\= X+I.

| ?- queens(4, Qs).
    Qs = [_A,_B,_C,_D], _A in 1..4, _B in 1..4, _C in 1..4, _D in 1..4 ?
| ?- queens(4, Qs), Qs = [1|_].
    Qs = [1,_A,_B,_C], _A in 3..4, _B in {2}\{4}, _C in 2..3 ?
| ?- queens(4, Qs), Qs = [1|_], labeling([], Qs).
    no
| ?- queens(4, Qs), Qs = [2|_], labeling([], Qs).
    Qs = [2,4,1,3] ?
```

## Egy példasor: Lovagok és lóköttők

### • A feladat

- Egy szigeten minden bennszülött lovag vagy lóköttő.
- A lovagok mindig igazat mondanak.
- A lóköttők mindig hazudnak.
- Egy vagy több bennszülöttnak saját magukra vonatkozó kijelentése alapján meg kell határozni a bennszülött típusát.
- Példa: Találkozunk két bennszülöttel Alfréd-dal és Bélával. Alfréd azt mondja: van köztünk lóköttő. 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 lóköttők: 1. változat (Prolog)

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

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

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

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

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

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

## Lovagok és lóköttők – A megoldás elvei

- Készítünk egy egyszerű formális nyelvet a bennszülöttek kijelentéseire, pl. Alfréd mondja Alfréd = lóköttő vagy Béla = lóköttő
- A bennszülöttek nevei (pl. Alfréd) Prolog változók, amelyek a lovag vagy lóköttő értéket vesznek 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 lóköttők: 2., CLP(B) változat

(A bennszülöttek típusát numerikusan jelöljük: lovag  $\rightarrow$  1, lóköttő  $\rightarrow$  0.)

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

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

A mondja Áll :- értéke(A mondja Áll, 1).

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

| ?- Alfréd mondja Alfréd = 0 vagy Béla = 0.
    Béla = 0, Alfréd = 1 ? ; no

| ?- A mondja B mondja C mondja A = C.
    B = 1 ? ; no

| ?- A mondja B = C.
    sat(B =\= C # A) ? ; no

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

## Lovagok és lóköltők: 3., CLP(FD) változat

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

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

A mondja Áll :- értéke(A mondja Áll, 1).

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

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

Deklaratív programozás. BME VIK, 2010. őszi félév

(Logikai Programozás)

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

(A bennszülöttek típusa: normális  $\rightarrow 2$ , lovag  $\rightarrow 1$ , lóköltő  $\rightarrow 0$ .)

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

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

A mondja Áll :- értéke(A mondja Áll, 1).

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

% http://www.math.wayne.edu/~boehm/Probweek2w99sol.htm: We are given three
% people, A, B, C, one of whom is a knight, one a knave, and one a normal
% (but not necessarily in that order). They make the following statements.
% A: I am normal, B: A is telling the truth, C: I am not normal
% What are A, B, and C?

| ?- A mondja A = 2, B mondja A = 2, C mondja nem C =2, all_different([A,B,C]),
    labeling([], [A,B,C]).
    A = 0, B = 2, C = 1 ? ; no
```

Deklaratív programozás. BME VIK, 2010. őszi félév

(Logikai Programozás)

## CLP rendszerek a nagyvilágban

- Néhány implementáció
  - clp(R) — az első CLP(X) rendszer (Monash Univ, Australia, IBM és CMU)
  - CHIP — FD, Q és B (ECRC, Németho., Cosytec, Francia.); CHARME (Bull); Decision Power (ICL)
  - Prolog III, Prolog IV (PrologIA, Marseille), Q (nem-lineáris is), B, FD, listák, intervallumok
  - ILOG solver (ILOG, Francia.) — C++ könyvtár: R (nem-lineáris is), FD, halmazok
  - SICStus Prolog (SICS, Svéd.) — R/Q, FD, B
  - GNU Prolog (INRIA, Francia.) — FD (C-re fordít)
  - Oz (DFKI, Németho.) — korlát alapú elosztott funkcionális nyelv.
- 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

Deklaratív programozás. BME VIK, 2010. őszi félév

(Logikai Programozás)

## Mire használják a CLP rendszereket — néhány példa

- Ipari erőforrás optimalizálás
  - termék- és gépkonfiguráció
  - gyártásütemezés
  - emberi erőforrások ütemezése
  - logisztikai tervezés
- Közlekedés, szállítás
  - repülőtéri allokációs feladatok (beszállókapu, poggyász-szalag stb.)
  - repülő-személyzet járatokhoz rendelése
  - menetrendkészítés
  - forgalomtervezés
- Távközlés, elektronika
  - GSM átjátszók frekvencia-kiosztása
  - lokális mobiltelefon-hálózat tervezése
  - áramkörtervezés és verifikálás

Deklaratív programozás. BME VIK, 2010. őszi félév

(Logikai Programozás)