EGYENLŐSÉGFAJTÁK — ÖSSZEFOGLALÁS

A Prolog egyenlőség-szerű beépített eljárásai

- *U* `=` *V*: *U* egyesítendő *V*-vel.
  Soha sem jelez hibát.

```
| ?- X = 1+2.      ⟹ X = 1+2
| ?- 3 = 1+2.      ⟹ no
```

- *U* `==` *V*: *U* azonos *V*-vel.
  Soha sem jelez hibát és soha sem helyettesít be.

```
| ?- X == 1+2.     ⟹ no
| ?- 3 == 1+2.     ⟹ no
| ?- +(1,2)==1+2   ⟹ yes
```

- *U* `=:=` *V*: Az *U* és *V* aritmetikai kifejezések értéke megegyezik.
  Hibát jelez, ha *U* vagy *V* nem (tömör) aritmetikai kifejezés.

```
| ?- X =:= 1+2.    ⟹ hiba
| ?- 1+2 =:= X.    ⟹ hiba
| ?- 2+1 =:= 1+2.  ⟹ yes
| ?- 2.0 =:= 1+1.  ⟹ yes
| ?- 2.0 is 1+1.   ⟹ no
```

- *U* `is` *V*: *U* egyesítendő a *V* aritmetikai kifejezés értékével.
  Hiba, ha *V* nem (tömör) aritmetikai kifejezés.

```
| ?- X is 1+2.     ⟹ X = 3
| ?- 1+2 is X.     ⟹ hiba
| ?- 3 is 1+2.     ⟹ yes
| ?- 1+2 is 1+2.   ⟹ no
```

- (*U* `=..` *V*: *U* „szétszedettje” a *V* lista)

```
| ?- 1+2 =.. X.    ⟹ X = [+,1,2]
| ?- X =.. [f,1].  ⟹ X = f(1)
```

A Prolog nem-egyenlőség jellegű beépített eljárásai

- **A nem-egyenlőség jellegű eljárások sohasem helyettesítenek be változót!**
  - `U \= V`: `U` nem egyesíthető `V`-vel. Soha sem jelez hibát.
    ```
    | ?- X \= 1+2.          ⟹ no
    | ?- +(1,2) \= 1+2.     ⟹ no
    ```
  - `U \== V`: `U` nem azonos `V`-vel. Soha sem jelez hibát.
    ```
    | ?- X \== 1+2.         ⟹ yes
    | ?- 3 \== 1+2.         ⟹ yes
    | ?- +(1,2)\==1+2       ⟹ no
    ```
  - `U =\= 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

		Egyesítés		***Azonosság***		***Aritmetika***		
`U`	`V`	`U = V`	`U \= V`	`U == V`	`U \== V`	`U =:= V`	`U =\= V`	`U 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.**

MODULARITÁS

Modulok definiálása SICStus Prolog nyelven

- A SICStus Prolog modulfogalmának jellemzői:
  - Minden modul külön állományba kell kerüljön.
  - Az állomány első programeleme egy modul-parancs kell legyen:

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

    ExpFunktor = az exportálandó eljárás funktora (név/argumentumszám)
  - Pl. `:- 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 funktora
    - *ÁllományNév* lehet atom, vagy pl. `library(`*KönyvtárNév*`)`:

      ```
      :- use_module(plato).                          % a fenti modul betöltése
      :- use_module(library(lists), [last/2]).  % csak last/2 importált
      ```
  - A modulfogalom nem szigorú: `platók:első_fennsík(...)` meghívható!
  - Modulkvalifikált hívási forma: *ModulNév*`:`*EljárásNév*`(`*Argumentumok* `...)`.

Meta-eljárások modularizált programban

- **Eljárások átadása paraméterként modulközi hívásban gondot okozhat:**
  - `m1.pl` **állomány:**

    ```
    :- module(m1, [kétszer/1]).
    kétszer(X) :- X, X.                         p :- write(bu).
    ```

  - `m2.pl` **állomány:**

    ```
    :- module(m2, [q/0,r/0]).
    :- use_module(m1).
    q :- kétszer(p).  r :- kétszer(m2:p).       p :- write(ba).
    ```

  - **Futtatás:**

    ```
    | ?- [m1,m2].
    | ?- q.  ⟹ bubu
    | ?- r.  ⟹ baba
    ```

  - **Automatikus modul-kvalifikáció meta-predikátum deklarációval:**
    Ha `m1.pl`**-be beszúrjuk:** `:- meta_predicate kétszer(:).`**, akkor**
    `| ?- q.` $\Longrightarrow$ `baba`**!**

Meta-predikátum deklaráció, modulnév-kiterjesztés

- Meta-predikátum deklaráció
  - Formája:
    `:- meta_predicate ⟨eljárásnév⟩(⟨módspec`$_1$`⟩, ..., ⟨módspec`$_n$`⟩), ....`
  - ⟨`módspec`$_i$⟩ lehet ‘`:`’, ‘`+`’, ‘`-`’, vagy ‘`?`’.
  - A ‘`:`’ mód azt jelzi, hogy az adott argumentumot betöltéskor ún. modulnév-kiterjesztésnek kell alávetni.
- Egy *Kif* kifejezés modulnév-kiterjesztése:
  - ha *M*:*X* alakú (vagy egy olyan változó, amely az adott eljárás fejében meta-argumentum pozíción szerepelt) akkor változatlanul hagyjuk;
  - egyébként helyettesítjük *CurMod*:*Kif*-fel (*CurMod* a kurrens modul).
- Példa folyt. (az `m1`-beli `kétszer` meta-predikátumnak deklarált!)

```
:- module(m2, [négyszer/1,q/0]). :- use_module(m1).
q :- kétszer(p).                          ⟹ q :- kétszer(m2:p).
:- meta_predicate négyszer(:).
négyszer(X) :- kétszer(X), kétszer(X). ⟹ változatlan
```

MAGASABBRENDŰ ELJÁRÁSOK

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:

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

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

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

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

Listakezelő meta-eljárások megoldásgyűjtő eszközökkel

- **Lista szűrése (vö. a `filter` SML függvénnyel!)**

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

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

- **Lista leképezése (vö. a `map` SML függvénnyel!)**

```
% Az L lista X elemeit Pred-del Y-ba képezve kapjuk az ML listát.
:- meta_predicate map(+, ?, :, ?, -).
map(L, X, Pred, Y, ML) :-
        findall(Y, (member(X, L), Pred), ML).

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

- **A példákban a szűrést az ⟨`X, Pred`⟩ argumentumpár, a leképezést az ⟨`X, Pred, Y`⟩ hármas határozza meg. Ezek egy egy- ill. kétargumentumú predikátumot adnak meg (vö. a funkcionális nyelvek λ-kifejezéseivel).**

Részlegesen paraméterezett eljáráshívások

- A listát elemenként négyzetreemelő eljárás egy másik változata:

```
négyzete(X, Y) :- Y is X*X.

négyzeteik(Xk, Yk) :- map(Xk, X, négyzete(X,Y), Y, Yk).
```

- A lista elemeire az $x \to x^2 + \mathtt{P}x + \mathtt{Q}$ hozzárendelést alkalmazó eljárás:

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

másodfokú_képeik(P, Q, Xk, Yk) :-
        map(Xk, X, másodfokú_képe(P,Q,X,Y), Y, Yk).
```

- Konvenció: a meta-alkalmazásban változó paramétereket az eljárás végére tesszük — így egyszerűsíthető a meta-eljárás hívása. Példa: `map/3`:

```
map(Xk, RészlPred, Yk) :-
        RészlPred =.. L0, append(L0, [X,Y], L), Pred =.. L,
        findall(Y, (member(X, Xk), Pred), Yk).

másodfokú_képeik(P, Q, Xk, Yk) :- map(Xk, másodfokú_képe(P,Q), Yk).
```

Részlegesen paraméterezett eljáráshívások — segédeszközök

- **A `call/1` eljárás általánosítása: a `call/2`, `call/3`, ... eljárások.**
- **`call(RPred, A1, A2, ...)` végrehajtása: az `RPred` hívást kiegészíti az `A1, A2, ...` argumentumokkal, és meghívja.**
- **A `call/N` eljárások sok Prologban beépítettek, SICStusban definiálandók:**

```
:- 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`

- `map/3` rekurzív definíciója:

```
% 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([], _, []).
```

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

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

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

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

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

- `asserta(:@Klóz)`
  - A `Klóz` kifejezést klózként értelmezve felveszi a programba az adott predikátum *első* klózaként.
  - 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)`
  - A `Klóz` kifejezést az adott predikátum *utolsó* klózaként veszi fel
- Példa:

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

p(2, 0).
p(1, A) :-   q(A).
p(2, A) :-   r(A).
```

Klóz törlése: `retract/1`

- `retract(:@Klóz)`
  - A `Klóz` klóz-kifejezésből megállapítja a predikátum funktorát.
  - Az adott predikátum klózait 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:

`p(2, 0).`
`p(1, A) :-`
`        q(A).`
`p(2, A) :-`
`        r(A).`
	`p(1, A) :-`
	`        q(A).`
	`p(2, A) :-`
	`        r(A).`
		`p(1, A) :-`
		`        q(A).`

Alkalmazási példa — egyszerűsített `findall`

- **A `findall1/3` eljárás hatása megegyezik a beépített `findall`-lal, de**
- **nem működik helyesen, ha a `Cél`-ban újabb `findall1` hívás van.**

```
:- dynamic(megoldás/1).

% findall1(Minta, Cél, L): Cél összes megoldására Minták listája L.
findall1(Minta, Cél, _MegoldL) :-
    call(Cél), asserta(megoldás(Minta)),  fail.     % fordított sorrend!
findall1(_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).

| ?- findall1(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 klózait 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).
```

<pre>p(2, 0).
p(1, A) :-
        q(A).
p(2, A) :-
        r(A).</pre>
	<pre>T = true ? ;
	T = r(0) ? ;
	no</pre>

A `clause` eljárás alkalmazása: egyszerű nyomkövető interpreter

```
% 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 szóközt 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).
```

```
| ?-  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] ?
```

„HAGYOMÁNYOS” BEÉPÍTETT ELJÁRÁSOK

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.
- `writeq(@X)`: Mint `write(X)`, csak gondoskodik, hogy szükség esetén az atomok idézőjelek közé legyenek téve.
- `write_canonical(@X)`: Mint `writeq(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
| ?- writeq('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 `write`-tal. Ha a felhasználó definiál egy `portray/1` eljárást, akkor a rendszer minden a `print`-tel kinyomtatandó részkifejezésre meghívja `portray`-t. Ennek sikere esetén feltételezi, hogy a kiírás megtörtént, meghiúsulás esetén maga írja ki a részkifejezést.

  A rendszer a `print` eljárást használja a változó-behelyettesítések és a nyomkövetés kiírására!
- `portray(@Kif)` (felhasználó által definiálandó ún. *kampó eljárás*): Igaz, ha `Kif` kifejezést a Prolog rendszernek *nem* kell kiírnia (és ekkor maga a `portray` kell, hogy elvégzze 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:**

```
% L a következő sor karakterkódjainak listája.
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
        ), !.

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

```
| ?- listing([p/1]).
p(A) :-
        q(A),
        r(A).
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 ismertetett 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”: `throw(@HibaKif)`, `raise_exception(@HibaKif)`
  - Hatása: Kiváltja a `HibaKif` hibahelyzetet.
- 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` **értkékét** `Érték`**-re állítja.**
- `current_prolog_flag(?Jelző, ?Érték)`: `Jelző` **pillanatnyi értéke** `Érték`.
- **Néhány fontos Prolog jelző:**
  - `language`: **végrehajtási mód** (`sicstus`, `iso`).
  - `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, leforditott alakot hozva létre.**
- `listing`: **Kiírja az összes interpretált eljárást az aktuális kimenetre.**
- `listing(:@EljárásSpec)`: **Kiírja 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)` $\Longrightarrow$`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.

NYELVTANI ELEMZÉS PROLOGBAN

Egy egyszerű nyelvtani elemzési példa

- Bináris számok nyelvtana

$$\begin{array}{lll}\langle\,\textbf{szám}\,\rangle & ::= & \langle\,\textbf{számjegy}\,\rangle\ \langle\,\textbf{számmaradék}\,\rangle \\ \langle\,\textbf{számmaradék}\,\rangle & ::= & \langle\,\textbf{számjegy}\,\rangle\ \langle\,\textbf{számmaradék}\,\rangle \mid \epsilon \\ \langle\,\textbf{számjegy}\,\rangle & ::= & \texttt{0} \mid \texttt{1}\end{array}$$

- 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 lehet karakterkódok listája, lexikai elemek (token-ek) listája stb.
- Egy nem-terminálisnak egy kétargumentumú Prolog szabály felel meg:

```
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 lehagyni a listáról, erre szolgál a `'C'/3` beépített eljárás. Definíciója: `'C'(L0, X, L)` $\equiv$ `L0 = [X|L]` (A SICStus fordító a két hívást pontosan ugyanúgy fordítja).

A DCG szabályok lefordított alakja

- **A DCG mint szintaktikai édesítőszer — 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 = [0'2] ; L = [0'0,0'2] ; no
```

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.
                  ;   []
                  ).

% Ugyanez vágóval
számmaradék -->   számjegy, !, számmaradék.
számmaradék -->   [].

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

| ?- számmaradék("102", L).  ⟹ L = [0'2] ; 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)}.

% 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 kód
        decimális_jegy_kódja(K). % megfelelő-e a K?
```

Az elemző kiegészítése argumentumokkal

- Egy DCG szabály argumentumaiban egy „belső” alakot építhet:

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

% leelemezhető számjegyek egy esetleg üres listája, amelynek
% az eddig leelemzett 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.
```

A DCG nyelvtani szabályok szerkezete — összefoglalás

- A DCG szabály alakja: ⟨ ***Baloldal*** ⟩ `->` ⟨ ***Jobboldal*** ⟩ .
- ⟨ ***Baloldal*** ⟩: egy nem-terminális(, amit esetleg terminálisok listája követ).
- ⟨ ***Jobboldal*** ⟩: konjunkció (,), diszjunkció (;), ha-akkor (->) és negáció (\+) segítségével épül terminálisokból, nem-terminálisokból és Prolog hívásokból.
- Nem-terminális: tetszőleges ***hívható*** kifejezés (atom 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).
- DCG = egy „ingyen” akkumulátor (akkum. 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

```
% kif(Z, L0, L): L0 elején egy Z értékű aritm. 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 = [42,54], Z = 94 ; ...
| ?- kif(Z, "4-2+1", []).       ==> Z = 1   Jobbról balra elemez!

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

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

DCG példa: „természetes” nyelvű beszélgetés

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

% mondat(Alany, Áll, L0, L): L0-L kielemezhető egy Alany alanyból és Áll
% állítmányból álló mondattá. 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 mondattá.
én_te_perm(Alany, Ige, Áll)  --> szó(Alany), szó(Ige), szavak(Áll).
én_te_perm(Alany, Ige, Áll)  --> szó(Alany), szavak(Áll), szó(Ige).
én_te_perm(Alany, Ige, Áll)  --> szavak(Áll), szó(Ige), szó(Alany).
én_te_perm(_Alany, Ige, Áll) --> szavak(Áll), szó(Ige).
```

Példa: "természetes" nyelvű beszélgetés — szavak elemzése

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

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

% 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], {non_member(K, " .?")}.

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

Példa: “természetes” nyelvű beszélgetés — párbeszéd-szervezés

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

% Megvalósít egy párbeszédet.
párbeszéd :-
    repeat, rd_line(L),                        % lásd korábban
    (   menet(Mondás, L, []) -> feldolgoz(Mondás)
    ;   write('Nem értem\n'), fail
    ),
    Mondás = un, !.

% menet(Mondás, L0, L): Az L0-L kielemzett 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").
```

Példa: “természetes” nyelvű beszélgetés — válaszok előállítása

```
:- dynamic tudom/2.

% feldolgoz(Mondás): feldolgozza a felhasználótól 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(_).
```

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.
```

A DCG formalizmus felhasználása elemzésen kívül

- DCG szabályok kényelmesen használhatók általános akkumulálásra
  - Listák akkumulálása (nemcsak elemzés, építés is:)

```
% anbn(+N, ?L): Az L lista N db a-ból és azt követő N db b-ből áll.
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.       | % L számlista összege S.
sum([]) --> [].                        | sum(L, S) :- sum(L, 0, S).
sum([X|L]) -->                         |
        plus(X), sum(L).               | plus(X, S0, S) :- S is S0+X.
```