

A Prolog megvalósítás néhány mérföldköve

- 1973: Marseille Prolog (A. Colmerauer et al.)
 - értelmező (interpreter), Fortran nyelven
 - kifejezések ábrázolása: struktúra-osztásos (structure-sharing), konstans idejű kifejezés-építés
 - veremszervezés: egyetlen verem (csak visszalépéskor szabadul fel)
- 1977: DEC-10 Prolog (D. H. D. Warren)
 - fordítóprogram Prolog és assembly nyelven (+ értelmező Prologban)
 - kifejezések ábrázolása: struktúra-osztásos (mérettel arányos idejű kifejezés-építés)
 - veremszervezés: három verem (visszalépéskor mindhárom felszabadul)
 - globális verem (global stack): globális (struktúra-beli) változók, szemétyűjtött
 - fő verem (local stack): eljárások, választási pontok, változók, det. lefutáskor felszabadul
 - nyom verem (trail): változó-behelyettesítések tárolása (vágónál felszabadítható)
- 1983: WAM — Warren Abstract Machine (D. H. D. Warren)
 - absztrakt gép Prolog programok végrehajtására
 - kifejezések ábrázolása: struktúra-másolásos (structure-copying)
 - három verem, mint DEC-10 Prologban, a globális verem tárolja a struktúrákat
 - A legtöbb mai Prolog WAM alapú (SICStus, SWI, GNU Prolog, ...)

Struktúrák ábrázolása

- A kétféle kifejezés-ábrázolás összehasonlítása:

	struktúra-osztásos	struktúra-másolásos
tárigény:	a változók számával arányos	a struktúra méretével arányos
struktúra-építés	konstans idejű	a struktúra méretével arányos idejű
struktúra-szétszedés	költségesebb	kevésbé költséges

- **Struktúra építése:** egy változónak és egy **programszövegbeli** struktúrának az egyesítése
- **FONTOS:** egy változó értékeként megjelenő struktúra egyesítése egy behelyettesítetlen változóval mindenképpen konstans költségű!
- Példa:


```
hosszabbít(L, [1,2,3,...,n|L]).
sokszoroz(0, L) :- !, L = [].
sokszoroz(N, L) :-
    hosszabbít(L0, L), N1 is N-1, sokszoroz(N1, L0).
```
- `sokszoroz(n, L)` költsége és tárigénye struktúra-osztásnál $O(n)$, struktúra-másolásnál $O(n^2)$
- A gyakorlatban mégis a struktúra-másolásos megoldás bizonyult hatékonyabbnak.

WAM: Prolog kifejezések tárolása

- A WAM-ban javasolt kifejezés-ábrázolás (LBT: low bit tagging scheme)

	<i>globális/lokális</i>	<i>globális verem</i>
● Behelyettesíthetetlen változó:	saját cím	REF
● Másik változóra/kifejezésre való utalás:	másik kif. címe	REF
● Névkonstans	atom tábla index	A CON
● Egész szám	egész érték	I CON
● Lista	cím	LIST
	cím:	fej-kifejezés
		farok-kifejezés
● Struktúra	cím	STRU
	cím:	funktor tábla index
		argumentum-kif.
		...

- A SICStus 3.x rendszer a 4 legmagasabb helyiértékű biten tárolja jelzőket (tag) — ezért a veremterületek mérete 256 Mbyte-ban korlátozott. (SICStus 4-ben már LBT séma lesz.)

WAM: néhány további részlet

- Változók kezelése
 - Két változó egyesítése: a fiatalabbik az öregebbikre utaló **REF** értéket kap
 - **Utalástalanítás**: az (esetleg többtagú) REF-lánc követése
 - Behelyettesíthetetlen változó \equiv önmagára mutató utalás \Rightarrow egyszerűbb utalástalanítás
- Visszalépés
 - **Feltételes változó**: behelyettesíthetetlen változó, öregebb mint a legfrissebb választási pont
 - Feltételes változó behelyettesítése esetén a változó címét beírjuk a nyom-verembe
 - Visszalépéskor a nyom alapján „visszacsináljuk” a változó-behelyettesítéseket, majd a vermeket visszahúzzuk
- SICStus programok WAM utasítás-sorozatra fordíthatók (*File.pl* \Rightarrow *File.wam*):


```
| ?- prolog:fshell_files(File, wam, []).
```
- A WAM bemutatása (tutorial): <http://www.vanx.org/archive/wam/wam.html>

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. szemétyűjtésre 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 ( $S = 0 + L_n + L_{n-1} + \dots + L_1$ ).
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 ( $S - L_1 - L_2 - \dots - L_n = 0$ ).
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 ( $0 + L_1 + L_2 + \dots + L_n$ ) 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. ( $\equiv \sigma L = S - S0$ )
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: *Vált0*; közbülső értékek: *Vált1*, ..., *Váltn*; végérték: *Vált*.
- 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) \implies 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 \implies 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 leahagyva 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 akkumulált 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

● SML megoldás

```

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

```

● 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(integer, 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, integer::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(integer)::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(integer)::in,
%               list(integer)::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(integer)::in, list(integer)::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

- A függvény eredménye a reláció utolsó argumentuma: $\text{hatv}(+A, +H, ?E): A^H = E$.
- A ciklusnak segéd eljárás 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 > 0, !, (H0 /\ 1 == 1 % /\ ≡ bitenkénti ``és`` -> E1 is E0*A0 ; E1 = E0), H1 is H0 >> 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 > 0) { if (h & 1) e *= a; h >>= 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 fejkomentje, a példában:


```
% hatv(+A0, +H0, +E0, ?E): A0H0 * 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: $b^2 - 4ac \geq 0, a \neq 0$
 - utófeltétel: $a^2x^2 + bx + c = 0$
 - 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 következik,
 - 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 (h > 0)
  { /* ciklus-invariáns: a0h0 == e*ah */
    /* 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 > 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 > 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:

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

MEGOLDÁSOK GYŰJTÉSE ÉS FELSOROLÁSA

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.
```

Mi a közös a felsoroló és gyűjtő megoldásokban?

- 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([_|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:

```
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

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|_] ->
        length(Ek, H)
    ).
```

Fennsíkok felsorolása — 2., hatékony megoldás

- Használjuk a megoldás-felsoroló sémát: `megoldás(V0, Param, E)!`
 - `V0`: »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)
    ).
```

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

- Az első fennsík előállítás:

```
% 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).
```

Gyűjtés és felsorolás kapcsolata

- Korábban láttuk, hogyan lehet egy keresési feladat gyűjtő és felsoroló eljárásait egy közös magból előállítani.
- Most 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.
```

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 újakkal 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):
 $Lista = \{ Gyűjtő\ másolat \mid (\exists X \dots Z) Cél\ igaz \}$
 ahol X, \dots, Z a `findall` hívásban levő szabad változók (azaz olyan, a hívás pillanatában behelyettesítetlen változók, amelyek a `Cél`-ban előfordulnak de a `Gyűjtő`-ben nem).

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

- Az eljárás végrehajtása (procedurális szemantikája):
 - a Cél kifejezést eljárásnévként 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):

$Lista = \{ Gyűjtő \mid Cél \text{ igaz} \}, Lista \neq []$.

A bagof megoldásgyűjtő eljárás (folyt.)

- Explicit kvantorok

- $bagof(Gyűjtő, V_1 \wedge \dots \wedge V_n \wedge Cél, Lista)$ alakú hívása a V_1, \dots, V_n változókat egzisztenciálisan kötöttnek tekinti, nem sorolja fel.

- jelentése: $Lista = \{ Gyűjtő \mid (\exists V_1, \dots, V_n) Cél \text{ igaz} \} \neq []$.

```
| ?- 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
- segéd eljárás 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), Vks), length(Vks, 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 rendezi (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
```

META-LOGIKAI ELJÁRÁSOK

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

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

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

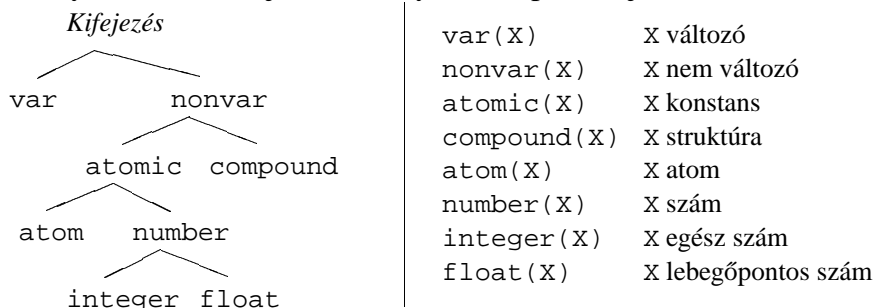
```
| ?- X @< 3 /* X megelőzi 3-t? */, X = 4. => X = 4  
    % a változók megelőzik a nem változó kifejezéseket  
| ?- X = 4, X @< 3. => no
```
- Prolog kifejezéseket szétszedő vagy összerakó eljárások:
 - (struktúra) kifejezés \iff név és argumentumok (2)

```
| ?- 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 behelyettesítetlen 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 behelyettesítettség 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
 - $Kif = Fun(A_1, \dots, A_n)$ és $Lista = [Fun, A_1, \dots, A_n]$, ahol Fun egy névkonstans és A_1, \dots, A_n 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 előállítása
 - 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 változó-argumentumának 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:

```
egysz1(Kif, EKif) :-
    Kif =.. [M|ArgL], egysz1_lista(ArgL, EArgL), EKif0 =.. [M|EArgL],
    % catch(:Cél,?Kiv,:KCél): ha Cél kivételt dob, KCél-t futtatja:
    catch(EKif is EKif0, _, EKif = EKif0).

egysz1_lista([], []).
egysz1_lista([K|Kk], [E|Ek]) :-
    egysz1(K, E), egysz1_lista(Kk, Ek).

| ?- egysz1(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: kiiratás

- A feladat: egy tetszőleges kifejezés kiiratása úgy, hogy
 - a kétargumentumú operátorok zárójelezett 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* egyesíthetőek;
- azaz, ha az egyik változót tartalmaz, akkor a másik ugyanott ugyanazt a változót tartalmazza.
- azonos/2 == néven, nem_azonos/2 \== néven szabványos beépített eljárás és operátor.

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

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

% 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, N), between(1, N, 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 névkonstanst, 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 struktúrák:
 - 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
$Kif1 == Kif2$	$Kif1 \not\prec Kif2 \wedge Kif2 \not\prec Kif1$
$Kif1 \backslash == Kif2$	$Kif1 \prec Kif2 \vee Kif2 \prec Kif1$
$Kif1 @< Kif2$	$Kif1 \prec Kif2$
$Kif1 @=< Kif2$	$Kif2 \not\prec Kif1$
$Kif1 @> Kif2$	$Kif2 \prec Kif1$
$Kif1 @>= Kif2$	$Kif1 \not\prec Kif2$

- 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 kifejezőosztályba 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).

```


A Prolog egyenlőség-szerű beépített eljárásai

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

A Prolog nem-egyenlőség jellegű beépített eljárásai

<ul style="list-style-type: none"> ● A nem-egyenlőség jellegű eljárások soha sem helyettesítenek be változót! 	
<ul style="list-style-type: none"> ● $U \backslash= V$: U nem egyesíthető V-vel. Soha sem jelez hibát. 	<pre> ?- X \= 1+2. => no ?- +(1,2) \= 1+2. => no</pre>
<ul style="list-style-type: none"> ● $U \backslash== V$: U nem azonos V-vel. Soha sem jelez hibát. 	<pre> ?- X \== 1+2. => yes ?- 3 \== 1+2. => yes ?- +(1,2)\==1+2 => no</pre>
<ul style="list-style-type: none"> ● $U \backslash= V$: Az U és V aritmetikai kifejezések értéke különbözik. Hibát jelez, ha U vagy V nem (tömör) aritmetikai kifejezés. 	<pre> ?- X \= 1+2. => hiba ?- 1+2 \= X. => hiba ?- 2+1 \= 1+2. => no ?- 2.0 \= 1+1. => no</pre>

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

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

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