MEGOLDÁSGYŰJTŐ BEÉPÍTETT ELJÁRÁSOK

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

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

- **Az eljárás végrehajtása:**
  - **a `Cél` kifejezést eljáráshívásként értelmezi, meghívja**
    (A `:+` annotáció meta- (azaz eljárás) argumentumot jelez);
  - **minden egyes megoldásához előállítja `Gyűjtő` egy *másolatát***
    (a megoldásbeli változók, ha vannak, szisztematikusan újakkal helyettesítődnek);
  - **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:** `Lista` $=$ { `Gyűjtő` másolat | $(\exists$`X`$\ldots$`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).

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

- Az eljárás végrehajtása:
  - a `Cél` kifejezést eljáráshívásként értelmezi;
  - összegyűjti a megoldásait (a `Gyűjtő`-t és a szabad változókat);
  - a `bagof`-beli 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
```

- Az eljárás jelentése: $\texttt{Lista} = \{\ \texttt{Gyűjtő} \mid \texttt{Cél}\ \text{igaz}\ \}$, $\texttt{Lista} \neq \texttt{[]}$.

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` | ($\exists$`V1`, ..., `Vn`)`Cél` 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` **nemdet. 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 előállítása**
  - **a vezérlési szerkezeteket célszerű elkerülni a meta-argumentumokban**
  - **segédeljárás bevezetésével a kvantor is szükségtelenné válik:**

```
 % Az A pont foka a G irányított gráfban 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 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 sorrendfüggőek):
  - 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 ⟺ név és argumentumok (2)

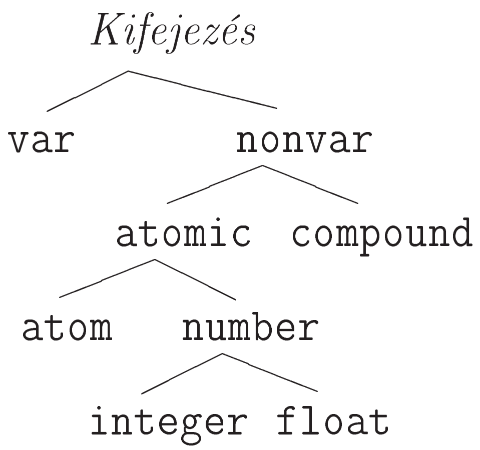
    ```
    | ?- X = f(alma,körte), X =.. L  ⟹ L = [f,alma,körte]
    ```

  - atomok és számok ⟺ karaktereik (3)

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

Kifejezések osztályozása

- Kifejezés-osztályok fastruktúrája — osztályozó beépített eljárások



`var(X)`	`X` változó
`nonvar(X)`	`X` nem változó
`atomic(X)`	`X` konstans
`compound(X)`	`X` struktúra
`atom(X)`	`X` atom
`number(X)`	`X` szám
`integer(X)`	`X` egész szám
`float(X)`	`X` lebegőpontos szám

- 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

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 ?
```

Osztályozó eljárások: nem-megkülönböztetett úniók kezelése

- **Példa: egy formula természetes módon ábrázolható Prologban, pl.** `x*y+y+1`**.**
  A formula típusának leírásához nem-megkülönböztetett úniót kell használni:

```
% :- type form == atom \/ int \/ { form+form } \/ { form-form } ...
```

- **Példa formulakezelésre: szimbolikus derivált előállítása**

```
% deriv(+Kif, +X, ?D):  Kif-nek az X atom szerinti deriváltja D.
deriv(X, X, D) :-   !, D = 1.
deriv(C, _X, D) :-  atomic(C), !, D = 0.
deriv(U+V, X, DU+DV) :-                deriv(U, X, DU), deriv(V, X, DV).
deriv(U-V, X, DU-DV) :-                deriv(U, X, DU), deriv(V, X, DV).
deriv(U*V, X, DU*V + U*DV) :-          deriv(U, X, DU), deriv(V, X, DV).
deriv(U/V, X, (DU*V - U*DV)/(V*V)) :- deriv(U, X, DU), deriv(V, X, DV).

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

Struktúrák szétszedése és összerakása: az *univ* eljárás

- **Az *univ* eljárás hívási mintái:**
  - `+Kif =.. ?Lista`
  - `-Kif =.. +Lista`
- **Az eljárás jelentése: Igaz, ha**
  - `Kif = ` $Fun(A_1, \ldots, A_n)$ és `Lista = ` $[Fun, A_1, \ldots\ A_n]$, **ahol** Fun **egy atom és** $A_1, \ldots\ A_n$ **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, +Nev, +Argszám)`
    `functor(+Kif, ?Nev, ?Argszám)`
  - Jelentése: igaz, ha `Kif` egy `Nev/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*** **(kétirányúság!).**
- **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 itt biztosan nem változó: */
    Kif =.. [Func, A1|ArgL],
    ( current_op(_, Kind, Func), (Kind=xfy;Kind=yfx;Kind=xfx),
      ArgL = [A2] ->  % kétargumentumú operátor
      write('('), ki(A1), format(' ~w ', [Fun]), ki(A2), write(')')
    ; write(Func), write('('), ki(A1), arglistaki(ArgL), write(')')
    ).
ki(Kif) :- write(Kif).

% 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 `'$VAR'(I)` alakú kifejezéssel helyettesíti, `I = N0, ..., N-1` (azaz `Kif`-ben `N-N0` különböző változó van).
- A `'$VAR'(0), '$VAR'(1), ...` kifejezések `write`-tal való kiíráskor változónévként (`A, B` ...) jelennek meg.
- Ezek speciális opciókkal `write_term`-mel „eredetiben” is megjeleníthetők:

```
| ?- _Kif = [f(_X),g(_),_X], numbervars(_Kif, 0, N), write(_Kif), nl,
     write_term(_Kif, [quoted(true),numbervars(false)]).
⟹       [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 : saját változómentesítés

```
% 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)]
```

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 $[i_1, i_2, \ldots, i_k]$ lista egy `Kif`-ből az i_1-edik argumentum i_2-edik argumentumának, $\ldots i_k$-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/3`: **atom 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` **bemenő, és a** $c_1 c_2 \ldots c_n$ **karakterekből áll, akkor** `KódLista =` $[k_1, k_2, \ldots, k_n]$**, ahol** k_i **a** c_i **karakter kódja.**
    - **Ha** `Atom` **kimenő, akkor a** `KódLista` **karakterkód-listából összerak egy atomot, 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 atomokban**

```
% 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 atomok 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/3`: **szám és karakterkód-lista közötti átalakítás**
  - **Hívási minták:** `number_codes(+Szám, ?KódLista)`
    `number_codes(-Szám, +KódLista)`
  - **Jelentése: Igaz, ha** `Szám` **tizes számrendszerbeli alakja a** `KódLista` **karakterkód-listának felel meg.**
  - **Végrehajtása:**
    - **Ha** `Szám` **bemenő, és tizes számrendszerben a** $c_1c_2...c_n$ **karakterekből áll, akkor** `KódLista = `$[k_1, k_2, ..., k_n]$**, ahol** k_i **a** c_i **karakter kódja.**
    - **Ha** `Szám` **kimenő, akkor a** `KódLista` **karakterkód-listából összerak egy számot, 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)
```

Kifejezések rendezése: szabványos sorrend

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

1. **Ha X és Y azonos, akkor sem $X \prec Y$ sem $Y \prec X$ nem igaz és fordítva.**
2. **Ha X és Y 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 az abc sorrenddel.**
6. **Ha X és Y struktúrák:**
   - 6.1. **Ha X és Y aritása 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 balról az első nem azonos argumentum dönt.**

Végtelen (ciklikus) kifejezésekre a fenti rendezés nem érvényes.

Kifejezések összehasonlítása — beépített eljárások

- Két tetszőleges kifejezés összehasonlítását végző eljárások:

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

- Az összehasonlító eljárások logikailag nem tiszták:

`| ?- X @< 3, X = 4.` $\Longrightarrow$ `X = 4`
`| ?- X = 4, X @< 3.` $\Longrightarrow$ `no`

- Az összehasonlítás mindig a belső ábrázolás szerint történik:

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

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

```
% T1 megelőzi T2-t a szabványos sorrendben (lényegében T1 @< T2)
precedes(T1, T2) :- \+ \+ (numbervars1(T1-T2, 0, _), prec(T1, T2)).

% T1 megelőzi T2-t, a változók már '$myvar'(n) konstansokra cseréltek.
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
        ).

% class(+T, -C): A T kifejezés a C-edik kifejezésosztályba tartozik.
class(T, C) :- ( /*vált*/ T='$myvar'(_) -> C=0
               ; /*szám*/ float(T) ->   C=1 ; integer(T) -> C=2
               ; /*atom*/ atom(T) ->    C=3 ; /*struktúra*/ C=4
               ).
```

A $\prec$ reláció megvalósítása (folyt.)

```
% S1 megelőzi S2-t (struktúra-kifejezésekre és atomokra).
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)
    ).

% A1 atom megelőzi A2 atomot (előfeltétel: A1 \= A2).
atom_prec(A1, A2) :-
    atom_codes(A1, C1), atom_codes(A2, C2), struct_prec(C1, C2).
```