Nyomkövetés – legfontosabb parancsok (SICStus + SWI)

- Beépített eljárások
  - `trace`, `debug` – a `c`, `l` paranccsal indítja a nyomkövetést
  - `notrace`, `nodebug` – kikapcsolja a nyomkövetést
  - `spy(P)`, `nospy(P)`, `nospyall` – töréspont be/ki a `P` eljárásra, $\forall$ ki.
- Alapvető nyomkövetési parancsok (SICStus: `<RET>`-tel kell lezárni)
  - `h` (help) – parancsok listázása
  - `c` (creep) vagy csak `<RET>` – lassú futás (minden kapunál megáll)
  - `l` (leap) – csak töréspontnál áll meg
  - `+` ill. `-` – töréspont be/ki a kurrens eljárásra
  - `s` (skip) – eljárástörzs átlépése (Call/Redo $\Rightarrow$ Exit/Fail)
  - `w` (write) – teljes mélységű kiíratás
  - `o` (out) SICStus, `u` (up) SWI – kilépés az eljárástörzsből
  - `r` (retry) – újrakezdi a kurrens hívás végrehajtását
- Információ-megjelenítő és egyéb parancsok
  - `g` (goals) – a kurrens hívást tartalmazó célok kiíratása
  - `b` (break) – új, beágyazott Prolog interakciós szint létrehozása
  - `n` (notrace) – nyomkövető kikapcsolása
  - `a` (abort) – a kurrens futás abbahagyása

Eljárás-doboz modell – OO szemléletben (kieg. anyag)

- Minden eljáráshoz tartozik egy osztály, amelynek van egy konstruktor függvénye (amely megkapja a hívási paramétereket) és egy `next` „adj egy (következő) megoldást” metódusa.
- Az osztály nyilvántartja, hogy hányadik klózban jár a vezérlés
- A metódus első meghívásakor az első klóz első Hívás kapujára adja a vezérlést
- Amikor egy részeljárás Hívás kapujához érkezünk, létrehozunk egy példányt a meghívandó eljárásból, majd
- meghívjuk az eljáráspéldány „következő megoldás” metódusát (*)
  - Ha ez sikerül, akkor a vezérlés átkerül a következő hívás Hívás kapujára, vagy a szülő Kilépési kapujára
  - Ha ez meghiúsul, megszüntetjük az eljáráspéldányt majd ugrunk az előző hívás Újra kapujára, vagy a következő klóz elejére, stb.
- Amikor egy Újra kapuhoz érkezünk, a (*) lépésnél folytatjuk.
- A szülő Újra kapuja (a „következő megoldás” nem első hívása) a tárolt klózsorszámnak megfelelő klózban az utolsó Újra kapura adja a vezérlést.

OO szemléletű dobozok: p/2 C++ kódrészlet (kieg. anyag)

Az ős/2 Prolog eljárásnak (298. dia) megfelelő C++ objektum „köv. megoldás” metódusa:

```
boolean os::next(           { // Return next solution for os/2
  switch(clno)           {
  case 0:                     // first call of the method
    clno = 1;                 // enter clause 1:                os(X,Y) :- sz(X,Z), os(Z,Y).
    szaptr = new sz(x, &z);   // create a new instance of subgoal sz(X,Z)
  redo11:
    if(!szaptr->next()) {     // if sz(X,Z) fails
      delete szaptr;          // destroy it,
      goto cl2;        }      // and continue with clause 2 of os/2
    pptr = new os(z, py);     // otherwise, create a new instance of subgoal os(Z,Y)
  case 1:                     // (enter here for Redo port if clno==1)
    /* redo12: */
    if(!pptr->next())  {      // if os(Z,Y) fails
      delete pptr;            // destroy it,
      goto redo11;     }      // and continue at redo port of sz(X,Z)
    return TRUE;              // otherwise, exit via the Exit port
  cl2:
    clno = 2;                 // enter clause 2:                os(X,Y) :- sz(X,Y).
    szbptr = new sz(x, py);   // create a new instance of subgoal sz(X,Y)
  case 2:                     // (enter here for Redo port if clno==2)
    /* redo21: */
    if(!szbptr->next()) {     // if sz(X,Y) fails
      delete szbptr;          // destroy it,
      return FALSE;    }      // and exit via the Fail port
    return TRUE;              // otherwise, exit via the Exit port
                        } }
```

Tartalom

17 Programozás Prologban
- A funkcionális és logikai megközelítés összevetése
- Prolog bevezetés – példák
- A Prolog nyelv alapszintaxisa
- Listakezelő eljárások Prologban
- Nyomkövetés: 4-kapus doboz modell
- További vezérlési szerkezetek
- Magasabbrendű eljárások
- Megoldásgyűjtő beépített eljárások
- Operátorok
- Meta-logikai eljárások

Diszjunkció

- Ismétlés: klóztörzsben a vessző (‘,’) jelentése „és”, azaz konjunkció
- A ‘;’ operátor jelentése „vagy”, azaz diszjunkció

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

```
fakt(N, F) :-
    (   N = 0, F = 1

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

- A diszjunkciót nyitó zárójel elérésekor választási pont jön létre
  - először a diszjunkciót az első ágára redukáljuk
  - visszalépés esetén a diszjunkciót a második ágára redukáljuk
- Tehát az első ág sikeres lefutása után kilépünk a diszjunkcióból, és az utána jövő célokkal folytatjuk a redukálást
  - azaz a ‘;’ elérésekor a ‘)’-nél folytatjuk a futást
- A ‘;’ skatulyázható (jobbról-balra) és gyengébben köt mint a ‘,’
- Konvenció: a diszjunkciót *mindig* zárójelbe tesszük,
  a skatulyázott diszjunkciót és az ágakat feleslegesen nem zárójelezzük.
  Pl. (a felesleges zárójelek aláhúzva, kiemelve): `(p;(q;r)), (a;(b,c);d)`

A diszjunkció mint szintaktikus édesítőszer

- A diszjunkció egy segéd-predikátummal mindig kiküszöbölhető, pl.:

```
a(X, Y, Z) :-
        p(X, U), q(Y, V),
        (   r(U, T), s(T, Z)
        ;   t(V, Z)
        ;   t(U, Z)
        ),
        u(X, Z).
```

- Kigyűjtjük azokat a változókat, amelyek a diszjunkcióban és azon kívül is előfordulnak(`U`, `V`, `Z`)
- A segéd-predikátumnak ezek a változók lesznek az argumentumai
- A segéd-predikátum minden klóza megfelel a diszjunkció egy ágának

```
seged(U, V, Z) :- r(U, T), s(T, Z).
seged(U, V, Z) :- t(V, Z).
seged(U, V, Z) :- t(U, Z).
```

```
a(X, Y, Z) :-
    p(X, U), q(Y, V),
    seged(U, V, Z),
    u(X, Z).
```

Diszjunkció – megjegyzések (kieg. anyag)

- Az egyes klózok ‘ÉS’ vagy ‘VAGY’ kapcsolatban vannak?
  - A program klózai ÉS kapcsolatban vannak, pl.

    ```
    szuloje('Imre', 'István').     szuloje('Imre', 'Gizella').        % (1)
    ```

    azt állítja: Imre szülője István ÉS Imre szülője Gizella.
  - Az *(1)* klózok alternatív (VAGY kapcsolatú) válaszokhoz vezetnek:

    ```
    :- szuloje('Imre' Ki). ⟹   Ki = 'István' ? ; Ki = 'Gizella' ? ; no
    ```

    „`Imre` szülője `Sz`” ha `( Sz = István` vagy `Sz = Gizella ).`
- Az *(1)* predikátum átalakítható egyetlen, diszjunkciós klózzá:

  ```
  szuloje('Imre', Sz) :-          (   Sz = 'István'
                                  ;   Sz = 'Gizella'
                                  ).                                 % (2)
  ```

  Vö. De Morgan azonosságok: $(A \leftarrow B) \wedge (A \leftarrow C) \equiv (A \leftarrow (B \vee C))$
- Általánosan: tetszőleges predikátum egyklózossá alakítható:
  - a klózokat azonos fejűvé alakítjuk, új változók és =-ek bevezetésével:

    ```
    szuloje('Imre', Sz) :- Sz = 'István'.
    szuloje('Imre', Sz) :- Sz = 'Gizella'.
    ```
  - a klóztörzseket egy diszjunkcióvá fogjuk össze, lásd *(2)*.

A meghiúsulásos negáció (NF – Negation by Failure)

- A `\+ Hívás` vezérlési szerkezet (vö. $\nvdash$ – nem bizonyítható) procedurális szemantikája
  - végrehajtja a `Hívás` hívást,
  - ha `Hívás` sikeresen lefutott, akkor meghiúsul,
  - egyébként (azaz ha `Hívás` meghiúsult) sikerül.
- `\+ Hívás` futása során `Hívás` legfeljebb egyszer sikerül
- `\+ Hívás` sohasem helyettesít be változót
- Példa: Keressünk (adatbázisunkban) olyan gyermeket, aki nem férfi!

```
| ?- sz(X, _Sz), \+ ffi(X). % negált cél ≡ ¬ffi(X)
⟹   X = 'Gizella' ? ; no
```

- Mi történik ha a két hívást megcseréljük?

```
| ?- \+ ffi(X), sz(X, _Sz). % negált cél ≡ ¬(∃X.ffi(X))
⟹   no
```

- $\backslash + H$ logikai megfelelője: $\neg\exists\vec{X}(H)$, ahol $\vec{X}$ a H-ban a hívás pillanatában behelyettesítetlen változókat jelöli. Emiatt $\backslash +$ **érzékeny a sorrendre!!!**

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

Gondok a meghiúsulásos negációval

- A negált cél jelentése függ attól, hogy mely változók bírnak értékkel
- Mikor nincs gond?
  - Ha a negált cél tömör (nincs benne behelyettesítetlen változó)
  - Ha nyilvánvaló, hogy mely változók behelyettesítetlenek (pl. mert „semmis” változók: _), és a többi változó tömör értékkel bír.

```
% nem_szulo(+Sz): adott Sz nem szulo
nem_szulo(Sz) :- \+ szuloje(_, Sz).
```

- A `\+` művelet a „Zárt Világ” feltételezésen alapul (Closed World Assumption – CWA): ami nem bizonyítható, az nem igaz.

```
| ?- \+ szuloje('Imre', X).        ⟹   no
| ?- \+ szuloje('Géza', X).        ⟹   true ?          (*)
```

  - A klasszikus matematikai logika következményfogalma monoton: ha a premisszák halmaza bővül, a következmények halmaza nem szűkülhet.
  - A CWA alapú logika nem monoton, példa: bővítsük a programot egy `szuloje('Géza', xxx).` alakú állítással $\Rightarrow$`(*)` meghiúsul.

Példa: együttható meghatározása lineáris kifejezésben

- Formula: számokból és az ‘x’ atomból ‘+’ és ‘*’ operátorokkal épül fel.
- Lineáris formula: a ‘*’ operátor (legalább) egyik oldalán szám áll.

```
% egyhat(Kif, E): A Kif lineáris formulában az x együtthatója E.
egyhat(x, 1).
egyhat(Kif, E) :-
        number(Kif), E = 0.
egyhat(K1+K2, E) :-
        egyhat(K1, E1),
        egyhat(K2, E2),
        E is E1+E2.
egyhat(K1*K2, E) :-          % (4)
        number(K1),
        egyhat(K2, E0),
        E is K1*E0.
egyhat(K1*K2, E) :-          % (5)
        number(K2),
        egyhat(K1, E0),
        E is K2*E0.
```

- A fenti megoldás hibás – többszörös megoldást kaphatunk:

```
| ?- egyhat(((x+1)*3)+x+2*(x+x+3), E).   ⟹   E = 8 ?; no
| ?- egyhat(2*3+x, E).                   ⟹   E = 1 ?; E = 1 ?; no
```

- A többszörös megoldás oka:
  az `egyhat(2*3, E)` hívás esetén a (4) és (5) klóz egyaránt sikeres!

Többszörös megoldások kiküszöbölése

- El kell érnünk, hogy ha a (4) sikeres, akkor (5) már ne sikerüljön
- A többszörös megoldás kiküszöbölhető:
  - Negációval – írjuk be (4) előfeltételének negáltját (5) törzsébe:

```
(...)
egyhat(K1*K2, E) :-                                          % (4)
        number(K1), egyhat(K2, E0), E is K1*E0.
egyhat(K1*K2, E) :-                                          % (5)
        \+ number(K1),
        number(K2), egyhat(K1, E0), E is K2*E0.
```

  - hatékonyabban, feltételes kifejezéssel:

```
(...)
egyhat(K1*K2, E) :-
        (   number(K1) -> egyhat(K2, E0), E is K1*E0
        ;   number(K2), egyhat(K1, E0), E is K2*E0
        ).
```

- A feltételes kifejezés hatékonyabban fut, mert:
  - nem kell kétszer futtatni a `number(K1)` feltételt
  - nem hagy választási pontot

Feltételes kifejezés Prologban

- Szintaxis (`felt`, `akkor`, `egyébként` tetszőleges célsorozatok):

```
(...) :-
        ...,
        (   felt -> akkor
        ;   egyébként
        ),
        ....
```

- Deklaratív szemantika: a fenti alak jelentése megegyezik az alábbival, ha a `felt` egy egyszerű feltétel (azaz nem oldható meg többféleképpen):

```
(...) :-
        ...,
        (   felt, akkor
        ;   \+ felt, egyébként
        ),
        ....
```

Feltételes kifejezések (folyt.)

- Procedurális szemantika

  A `(felt->akkor;egyébként),folytatás` célsorozat végrehajtása:

  - Végrehajtjuk a `felt` hívást.
  - Ha `felt` sikeres, akkor az `(akkor,folytatás)` célsorozatra redukáljuk a fenti célsorozatot, a `felt` *első* megoldása által adott behelyettesítésekkel. A `felt` cél többi megoldását nem keressük meg!
  - Ha `felt` sikertelen, akkor az `(egyébként,folytatás)` célsorozatra redukáljuk, behelyettesítés nélkül.

- Többszörös elágaztatás skatulyázott feltételes kifejezésekkel:

```
(   felt1 -> akkor1
;   felt2 -> akkor2
;   ...
)
```

```
(   felt1 -> akkor1
;  (felt2 -> akkor2
;   ...
))
```

  A kiemelt zárójelek feleslegesek!

- Az `egyébként` rész elhagyható, alapértelmezése: `fail`.
- `\+ felt` átírható feltételes kifejezéssé: `( felt -> fail ; true )`

Feltételes kifejezés – példák

- Faktoriális

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

- Jelentése itt azonos a diszjunkciós alakkal (-> helyett , – lásd komment)
- A diszjunkciós alak választási pontot hagy, a feltételes szerkezet nem.
- Szám előjele

```
% Sign = sign(Num)
sign(Num, Sign) :-
        (   Num > 0 -> Sign = 1     %   if Num > 0 then Sign = 1
        ;   Num < 0 -> Sign = -1    % elif Num < 0 then Sign = -1
        ;              Sign = 0     % else              Sign = 0
        ).
```

A vágó eljárás – a feltételes szerkezet megvalósítási alapja

- A vágó beépített eljárás (`!`) mindig sikerül; de mellékhatásként
  1. letiltja az adott predikátum további klózainak választását,

```
első_poz_elem([X|_], X) :- X > 0, !.      % "zöld vágó"
első_poz_elem([X|L], EP) :- X =< 0, első_poz_elem(L, EP).
```

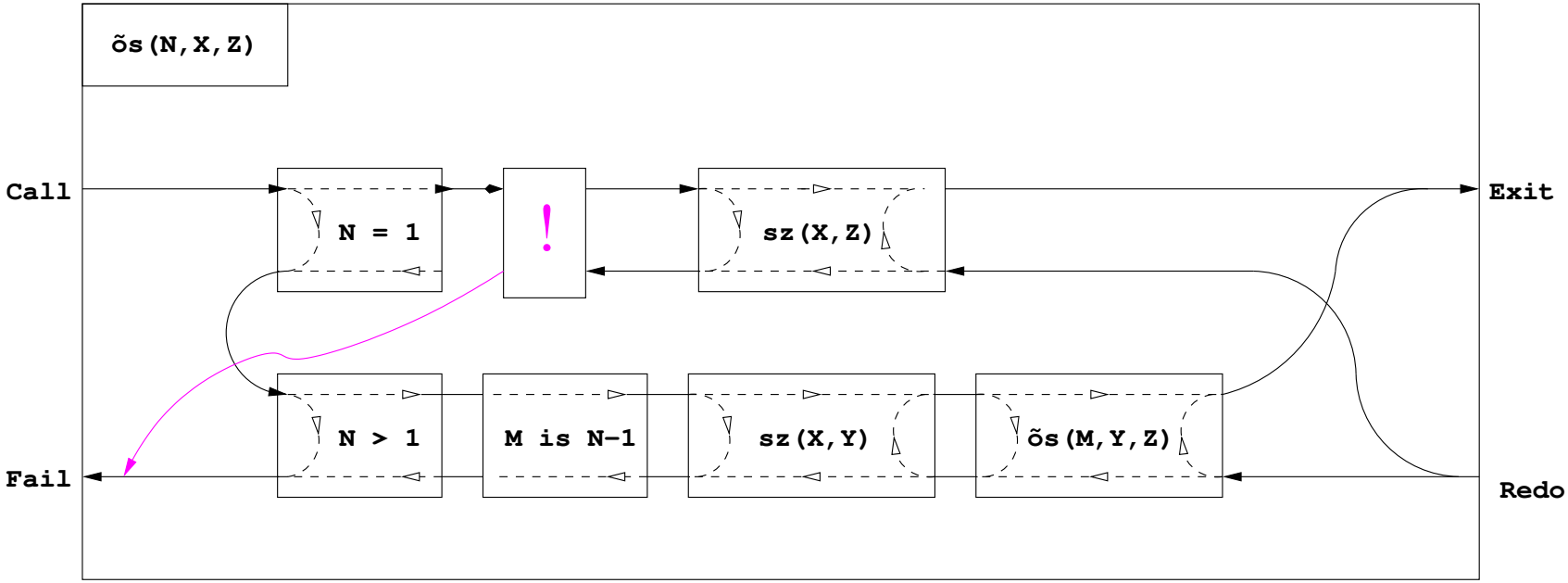
  2. megszünteti a választási pontokat az előtte levő eljáráshívásokban.

```
első_poz_elem(L, EP) :-  member(X, L), X>0, !, EP = X. % "vörös vágó"
```

- A zöld vágó nem változtatja meg az eredmény(eke)t, de gyorsítja a futást
- A vörös vágó megváltoztatja az eredményhalmazt
- Segédfogalom: egy cél szülőjének az őt tartalmazó klóz fejével illesztett hívást nevezzük
  - A 4-kapus modellben a szülő a körülvevő dobozhoz rendelt cél.
  - A fenti vágók szülője lehet pl. az `első_poz_elem([-1,0,3,0,2], P)` cél
- Átfogalmazás: a vágó a keresési térben vág le ágakat:
  - a vágó meghívásától visszafelé egészen a szülő célig – azt is beleértve – megszünteti a választási pontokat.

A vágó megvalósítása a 4-kapus doboz modellben

```
% ős(+N, ?X, ?Z): X-nek N-edik generációs őse Z (N>0 adott egész szám)
ős(1, X, Z) :-  !, sz(X, Z).                          % sz(X, Z): X-nek szülője Z
ős(N, X, Z) :-  N > 1, M is N-1, sz(X, Y), ős(M, Y, Z).
```



- A vágó `Fail` kapujából a körülvevő (szülő) doboz `Fail` kapujára megyünk.
- Ugyanilyen doboz keletkezik feltételes szerkezet használatakor:

```
ős2(N, X, Z) :- (   N = 1 -> sz(X, Z)
                ;   N > 1, M is N-1, sz(X, Y), ős2(M, Y, Z).
                ).
```

Tartalom

17 Programozás Prologban
- A funkcionális és logikai megközelítés összevetése
- Prolog bevezetés – példák
- A Prolog nyelv alapszintaxisa
- Listakezelő eljárások Prologban
- Nyomkövetés: 4-kapus doboz modell
- További vezérlési szerkezetek
- Magasabbrendű eljárások
- Megoldásgyűjtő beépített eljárások
- Operátorok
- Meta-logikai eljárások

Magasabbrendű eljárások – listakezelés

- Magasabbrendű (vagy meta-eljárás) egy eljárás,
  - ha eljárásként értelmezi egy vagy több argumentumát
  - pl. `findall/3`, `call/1`: `call(P)` a `P` kifejezést hívásként végrehajtja.
- Listafeldolgozás `findall` segítségével – példák
  - Páros elemek kiválasztása (vö. Elixir filter)

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

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

  - A listaelemek négyzetre emelése (vö. Elixir map)

```
% Az L számlista  elemei négyzeteinek listája Nk.
négyzetei0(L, Nk) :-
    findall(Y, (member(X, L), négyzete(X, Y)), Nk).
négyzete(X, Y) :- Y is X*X.
```

```
| ?- négyzetei0([1,2,3,4], Nk).    ⟹   Nk = [1,4,9,16]
```

- A `findall` futása során a megoldásokat le kell másolja – ez nagyobb adatstruktúrák esetén komoly hátrány lehet

Részlegesen paraméterezett eljáráshívások – segédeszközök

- A `négyzetei/2`-nek megfelelő feladatot Elixirben a `map/2` magasabbrendű függvénnyel oldhatjuk meg.
- Prologban ennek a `maplist/3` eljárás felel meg, amely a `library(lists)`-ben található:

```
:- use_module(library(lists)).          %
négyzetei(Xs, Ys) :-
        % Xs minden minden X elemére hívjuk meg a négyzete(X,Y)
        % eljárást, és a kapott Y értékeket gyűjtsük az Ys listába
        maplist(négyzete, Xs, Ys).
```

- A `maplist/3` az esetek nagy többségében visszavezethető a `findall/3`-ra:

```
maplist0(Fun, Xs, Ys) :-
    findall(Y, (member(X, Xs), call(Fun, X, Y)), Ys).
```

- A `négyzete` argumentum a `négyzete/2` részlegesen paraméterezett hívásának tekinthető: `call(négyzete, X, Y)` ≡ `négyzete(X, Y)`
- Általánosan: `call(RPred, A1, A2, ...)` végrehajtása: az `RPred` részleges hívást kiegészíti az `A1, A2, ...` argumentumokkal, és meghívja.
- A `call/N` predikátumok SICStus 4-ben beépített eljárásként állnak rendelkezésre

Részlegesen paraméterezett eljárások – rekurzív `maplist/3`

- Részleges paraméterezéssel a `maplist/3` meta-eljárás rekurzívan is definiálható:

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

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

- Példák:

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

- A `call/N`-re épülő megoldás előnyei:
  - általánosabb és hatékonyabb lehet, mint a `findall`-ra épülő;
  - alkalmazható akkor is, ha az elemekre elvégzendő műveletek nem függetlenek, pl. `foldl`.

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

- ```
  % foldl(+Xs, :Pred, +Y0, -Y): Y0-ból indulva, az Xs elemeire
  % balról jobbra sorra alkalmazva a Pred által leírt
  % kétargumentumú függvényt kapjuk Y-t.
  % SICStus library(lists)-ben scanlist/4 néven érhető el.
  foldl([X|Xs], Pred, Y0, Y) :-
          call(Pred, X, Y0, Y1), foldl(Xs, Pred, Y1, Y).
  foldl([], _, Y, Y).

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

  | ?- foldl([1,2,3], jegyhozzá(10), 0, E).  ⟹  E = 123
  ```
- ```
  % foldr(+Xs, :Pred, +Y0, -Y): Y0-ból indulva, az Xs elemeire jobbról
  % balra sorra alkalmazva a Pred kétargumentumú függvényt kapjuk Y-t.
  foldr([X|Xs], Pred, Y0, Y) :-
          foldr(Xs, Pred, Y0, Y1), call(Pred, X, Y1, Y).
  foldr([], _, Y, Y).

  | ?- foldr([1,2,3], jegyhozzá(10), 0, E).  ⟹  E = 321
  ```
- A `foldr` eljárás nem jobbrekurzív, ezért ritkábban használjuk

Tartalom

17 Programozás Prologban
- A funkcionális és logikai megközelítés összevetése
- Prolog bevezetés – példák
- A Prolog nyelv alapszintaxisa
- Listakezelő eljárások Prologban
- Nyomkövetés: 4-kapus doboz modell
- További vezérlési szerkezetek
- Magasabbrendű eljárások
- Megoldásgyűjtő beépített eljárások
- Operátorok
- Meta-logikai eljárások

Keresési feladat Prologban – felsorolás vagy gyűjtés?

- Keresési feladat: adott 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 visszalépéssel sorra előáll minden megoldás.
- Egyszerű példa: egy egészlista 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 ->
        Pk = [X|Pk1],
        páros_elemei(L, Pk1)
    ;   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
    % X akár páros, akár páratlan
    % folytatjuk a felsorolást:
    ;   páros_eleme(L, P)
    ).
% egyszerűbb, deklaratív megoldás:
páros_eleme2(L, P) :-
    member(P, L), P mod 2 =:= 0.
```

Gyűjtés és felsorolás kapcsolata

- Ha adott `páros_elemei`, hogyan definiálható `páros_eleme`?
  - A `member/2` beépített eljárás segítségével, pl.

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

  - Természetesen ez így nem hatékony!
- Ha adott `páros_eleme`, hogyan definiálható `páros_elemei`?
  - Megoldásgyűjtő beépített eljárás segítségével, pl.

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

  - a `findall/3` beépített eljárás – és társai – az Elixir listajelölőhöz (komprehenzióhoz) hasonlóak, pl.:

```
% my_numlist(+A, +B, ?L): L = [A,...,B], A és B egészek.
my_numlist(A, B, L) :-
    B >= A-1,
    findall(X, between(A, B, X), L).
```

    vö. $L = \{X | A \leq X \leq B, integer(X)\}$, ahol $B \geq A - 1$

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

- Ezt az eljárást már korábban bemutattuk, most részletesen ismertetjük
- Az eljárás végrehajtása (procedurális szemantikája):
  - a `Cél` kifejezést eljáráshívásként értelmezi, meghívja (A `:Cél` annotáció meta- (azaz eljárás) argumentumot jelez);
  - minden egyes megoldásához előállítja `Gyűjtő` egy *másolatát*, azaz a változókat, ha vannak, szisztematikusan újakkal helyettesíti;
  - Az összes `Gyűjtő` másolat listáját 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(Y, member(X-Y, [a-c,a-b,b-c,c-e,b-d]), L).
                ⟹   L = [c,b,c,e,d] ? ; no
```

- Az eljárás jelentése (deklaratív szemantikája):
  `Lista` = { `Gyűjtő` másolat | (∃ `X` ... `Z`)`Cél` igaz }
  ahol `X`, ..., `Z` a `findall` hívásban levő *szabad változók*.

  Szabad változó (definíció): olyan, a hívás pillanatában behelyettesítetlen változó, amely a `Cél`-ban előfordul de a `Gyűjtő`-ben nem.

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

- Példa az eljárás használatára:

```
gráf([a-c,a-b,b-c,c-e,b-d]).
| ?- gráf(_G), findall(B, member(A-B, _G), VegP).      % ld. előző dia
                              ⟹   VegP = [c,b,c,e,d] ? ; no
| ?- gráf(_G), bagof(B, member(A-B, _G), VegPk).
                              ⟹   A = a, VegPk = [c,b] ? ;
                              ⟹   A = b, VegPk = [c,d] ? ;
                              ⟹   A = c, VegPk = [e] ? ; no
```

- Az eljárás végrehajtása (procedurális szemantikája):
  - a `Cél` kifejezést eljáráshívásként értelmezi, meghívja;
  - összegyűjti a megoldásait (a `Gyűjtő`-t és a szabad változók behelyettesítéseit);
  - a szabad változók összes behelyettesítését *felsorolja* és mindegyik esetén a `Lista`-ban megadja az összes hozzá tartozó `Gyűjtő` értéket.
- A `bagof` eljárás jelentése (deklaratív szemantikája):
  `Lista` $=\{$ `Gyűjtő` $\mid$ `Cél` igaz $\}$, `Lista` $\neq$ `[]`, a szabad változók minden lehetséges behelyettesítésére.

A `bagof` megoldásgyűjtő eljárás – folyt. (kieg. anyag)

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

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

- Egymásba ágyazott gyűjtések
  - szabad változók esetén a `bagof` nemdeterminisztikus lehet, így érdemes lehet skatulyázni:

```
% A G irányított gráf fokszámlistája FL:
% FL = { A-N | N = |{ V | A-V ∈ G }|, N > 0 }
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. (kieg. anyag)

- Fokszámlista kicsit hatékonyabb előállítása
  - Az előző példában a meta-argumentumban célsorozat szerepelt, ez mindenképpen interpretáltan fut – nevezzük el segédeljárásként
  - A segédeljárás bevezetésével a kvantor is szükségtelenné válik:

```
% pont_foka(?A, +G, ?N): 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 költségesebb!

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

- Az eljárás végrehajtása:
  - ugyanaz mint: `bagof(Gyűjtő, Cél, L0), sort(L0, Lista),`
  - `sort(L, RL)` egy univerzális rendező eljárás, amely az `L` listát rendezi az az azonos elemek kiszűrésével (a `@<` általános rendezés szerint, lásd később), és az eredményt `RL`-ban adja vissza.
- Példa a `setof/3` eljárás használatára:

```
gráf([a-c,a-b,b-c,c-e,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,e] ? ; no
| ?- gráf(_G), bagof(P, pontja(P, _G), Pk).
⟹    Pk = [a,c,a,b,b,c,c,e,b,d] ? ; no
```

Tartalom

17 Programozás Prologban
- A funkcionális és logikai megközelítés összevetése
- Prolog bevezetés – példák
- A Prolog nyelv alapszintaxisa
- Listakezelő eljárások Prologban
- Nyomkövetés: 4-kapus doboz modell
- További vezérlési szerkezetek
- Magasabbrendű eljárások
- Megoldásgyűjtő beépített eljárások
- Operátorok
- Meta-logikai eljárások

Operátoros kifejezések

- Példa: `S is -S1+S2` ekvivalens az `is(S, +(-(S1),S2))` kifejezéssel
- Szintaxis:

  ⟨ összetett kif. ⟩ ::=

	⟨ struktúranév ⟩ (⟨ argumentum ⟩, . . .)	{eddig csak ez volt}
|	⟨ argumentum ⟩ ⟨ operátornév ⟩ ⟨ argumentum ⟩	{infix kifejezés}
|	⟨ operátornév ⟩ ⟨ argumentum ⟩	{prefix kifejezés}
|	⟨ argumentum ⟩ ⟨ operátornév ⟩	{posztfix kifejezés}
|	(⟨ kifejezés ⟩)	{zárójeles kif.}

  ⟨ operátornév ⟩ ::= ⟨ struktúranév ⟩ {ha operátorként lett definiálva}
- Operátor(ok) definiálása

  `op(Prio, Fajta, OpNév)` vagy `op(Prio, Fajta, [OpNév`$_1$`,...OpNév`$_n$`])`, ahol
  - `Prio` (prioritás): 1–1200 közötti egész
  - `Fajta`: az `yfx`, `xfy`, `xfx`, `fy`, `fx`, `yf`, `xf` névkonstansok egyike
  - `OpNév`$_i$ (az operátor neve): tetszőleges névkonstans
- Az `op/3` beépített predikátum meghívását általában a programot tartalmazó fájl elején, *direktívában* helyezzük el:

  ```
  :- op(800, xfx, [szuloje,nagyszuloje]).  'Imre' szuloje 'István'.
  ```
- A direktívák a programfájl *betöltésekor* azonnal végrehajtódnak.

Operátorok jellemzői

- Egy operátort jellemez a fajtája és prioritása
- A fajta az asszociatívitás irányát és az irásmódot határozza meg:

Fajta			Írásmód	Értelmezés
bal-assz.	jobb-assz.	nem-assz.		
`yfx`	`xfy`	`xfx`	infix	`A f B` ≡ `f(A, B)`
	`fy`	`fx`	prefix	`f A` ≡ `f(A)`
`yf`		`xf`	posztfix	`A f` ≡ `f(A)`

- A zárójelezést a prioritás és az asszociatívitás együtt határozza meg, pl.
  - `a/b+c*d` ≡ `(a/b)+(c*d)` mert `/` és `*` prioritása 400 < 500 (`+` prioritása) (kisebb prioritás = erősebb kötés)
  - `a-b-c` ≡ `(a-b)-c` mert a `-` operátor fajtája `yfx`, azaz bal-asszociatív – balra köt, balról jobbra zárójelez (a fajtanévben az `y` betű mutatja az asszociatívitás irányát)
  - `a^b^c` ≡ `a^(b^c)` mert a `^` operátor fajtája `xfy`, azaz jobb-asszociatív (jobbra köt, jobbról balra zárójelez)
  - `a=b=c` szintaktikusan hibás, mert az `=` operátor fajtája `xfx`, azaz nem-asszociatív

Szabványos, beépített operátorok

Szabványos operátorok

Színkód: már ismert, új aritmetikai

1200	xfx	`:- -->`	
1200	fx	`:- ?-`	
1100	xfy	`;`	diszjunkció
1050	xfy	`->`	if-then
1000	xfy	`','`	
900	fy	`\+`	negáció
700	xfx	`= \=`	
		`< =< > >= =:= =\= is`	
		`@< @=< @> @>= == \== =..`	
500	yfx	`+ - \/ /\`	bitműveletek
400	yfx	`* / // rem`	
		`mod`	modulus
		`<< >>`	léptetések
200	xfx	`**`	hatványozás
200	xfy	`^`	
200	fy	`- \`	bitenkénti negáció

További beépített operátorok SICStus Prologban

1150	fx	`mode public`
		`dynamic block`
		`volatile`
		`discontiguous`
		`initialization`
		`multifile`
		`meta_predicate`
1100	xfy	`do`
900	fy	`spy nospy`
550	xfy	`:`
500	yfx	`\`
200	fy	`+`

Operátorok zárójelezése (kieg. anyag)

- Egy `X` $\texttt{op}_1$ `Y` $\texttt{op}_2$ `Z` zárójelezése, ahol $\texttt{op}_1$ és $\texttt{op}_2$ prioritása n_1 és n_2:
  - ha $n_1 > n_2$ akkor `X` $\texttt{op}_1$ `(Y` $\texttt{op}_2$ `Z)`;
  - ha $n_1 < n_2$ akkor `(X` $\texttt{op}_1$ `Y)` $\texttt{op}_2$ `Z`; (kisebb prio. $\Rightarrow$ erősebb kötés)
  - ha $n_1 = n_2$ és $\texttt{op}_1$ jobb-asszociatív (`xfy`), akkor `X` $\texttt{op}_1$ `(Y` $\texttt{op}_2$ `Z)`;
  - egyébként, ha $n_1 = n_2$ és $\texttt{op}_2$ bal-assz. (`yfx`), akkor `(X` $\texttt{op}_1$ `Y)` $\texttt{op}_2$ `Z`;
  - egyébként szintaktikus hiba
- Érdekes példa: `:- op(500, xfy, +^).  % :- op(500, yfx, +).`

  `| ?- :- write((1 +^ 2) + 3), nl.` $\Rightarrow$ `(1+^2)+3`

  `| ?- :- write(1 +^ (2 + 3)), nl.` $\Rightarrow$ `1+^2+3`

  tehát: konfliktus esetén az első operátor asszociativitása „győz”.
- Alapszabály: egy n prioritású operátor zárójelezetlen operandusaként
  - legfeljebb $n-1$ prioritású operátort fogadunk el az `x` oldalon
  - legfeljebb n prioritású operátort fogadunk el az `y` oldalon
- A zárójelezett kifejezéseket és az alapstruktúra-alakú kifejezéseket feltétel nélkül elfogadjuk operandusként
- Az alapszabály a prefix és posztfix operátorokra is alkalmazandó

Operátorok – további megjegyzések

- Ugyanaz a névkonstans használható többféle fajtájú operátorként is, pl. a ‘-’ és ‘+’ atomok prefix és infix beépített operátorként is definiálva vannak a Prolog ISO szabványában
- A „vessző” jel három szintaktikus helyzetben is használható:
  - összetett kifejezés (struktúra) argumentumait határoló jel pl. `szuloje('István', 'Gizella')`
  - listaelemeket határoló jel, pl. `[1,2,3|T]`
  - 1000 prioritású `xfy` op. pl.: `(p:-a,b,c)`$\equiv$`:-(p,','(a,','(b,c)))`
- A vessző atomként csak a `','`, határolóként csak a `,`, operátorként mindkét formában – `','` vagy `,` – használható.
- `:-(p, a,b,c)` többértelmű: $\stackrel{?}{=}$ `:-(p, (a,b,c))`, ... $\stackrel{?}{=}$ `:-(p,a,b,c)`...
- Egyértelműsítés: argumentumban vagy listaelemben az 1000-nél $\geq$ prioritású operátort tartalmazó kifejezést *zárójelezni kell*:

```
| ?- write_canonical((a,b,c)).  ⟹  ','(a,','(b,c))
| ?- write_canonical(a,b,c).    ⟹  ! write_canonical/3 does not exist
```

Operátorok törlése, lekérdezése (kieg. anyag)

- Egy vagy több operátor törlésére az `op/3` beépített eljárást használhatjuk, ha első argumentumként (prioritásként) 0-t adunk meg.

```
| ?- X = a+b, op(0, yfx, +). ⟹   X = +(a,b) ? ; no
| ?- X = a+b.               ⟹   ! Syntax error
                                 ! op. expected after expression
                                 ! X = a <<here>> + b .
| ?- op(500, yfx, +).       ⟹   yes
| ?- X = +(a,b).            ⟹   X = a+b ? ; no
```

- Az adott pillanatban érvényes operátorok lekérdezése:

```
current_op(Prioritás, Fajta, OpNév)
```

```
| ?- current_op(P, F, +).
⟹   F = fy, P = 200 ? ;
     F = yfx, P = 500 ? ;
no
| ?- current_op(_, xfy, Op), write_canonical(Op), write(' '), fail.
; do -> ',' : ^
no
```

Operátorok felhasználása

- Mire jók az operátorok?
  - aritmetikai eljárások kényelmes irására, pl. `X is (Y+3) mod 4`
  - szimbolikus kifejezések kezelésére (pl. szimbolikus deriválás)
  - klózok leírására (`:-` és `','` is operátor), és meta-eljárásoknak való átadására, pl `asserta( (p(X):-q(X),r(X)) )`
  - eljárásfejek, eljáráshívások olvashatóbbá tételére:

    ```
    :- op(800, xfx, [nagyszülője, szülője]).

    Gy nagyszülője N :-  Gy szülője Sz, Sz szülője N.
    ```

  - adatstruktúrák olvashatóbbá tételére, pl.

    ```
    sav(kén, h*2-s-o*4).
    ```

Operátoros példa: polinom behelyettesítési értéke

- Polinom: az ‘x’ atomból és számokból a ‘+’ és ‘*’ op.-okkal felépülő kif.
- A feladat: egy polinom értékének kiszámolása egy adott x érték esetén.

```
% value_of0(P, X, V): A P polinom x=X helyen vett értéke V.
value_of0(x, X, V) :-
    V = X.
value_of0(N, _, V) :-
    number(N), V = N.
value_of0(P1+P2, X, V) :-
    value_of0(P1, X, V1),
    value_of0(P2, X, V2),
    V is V1+V2.
value_of0(P1*P2, X, V) :-
    value_of0(P1, X, V1),
    value_of0(P2, X, V2),
    V is V1*V2.
```

```
| ?- value_of0((x+1)*x+x+2*(x+x+3), 2, V).
V = 22 ? ; no
```

Klasszikus szimbolikuskifejezés-feldolgozás: deriválás

- Írjunk olyan Prolog predikátumot, amely az `x` névkonstansból és számokból a `+`, `-`, `*` műveletekkel képzett kifejezések deriválását elvégzi!

```
% deriv(Kif, D):  Kif-nek az x szerinti deriváltja D.
deriv(x, D) :-                      D = 1.
deriv(C, D) :-                      number(C), D = 0.
deriv(U+V, DU+DV) :-                deriv(U, DU), deriv(V, DV).
deriv(U-V, DU-DV) :-                deriv(U, DU), deriv(V, DV).
deriv(U*V, DU*V + U*DV) :-          deriv(U, DU), deriv(V, DV).
```

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

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

```
| ?- deriv(I, 1*x+x*1+1).
  ⟹     I = x*x+x ? ; no
```

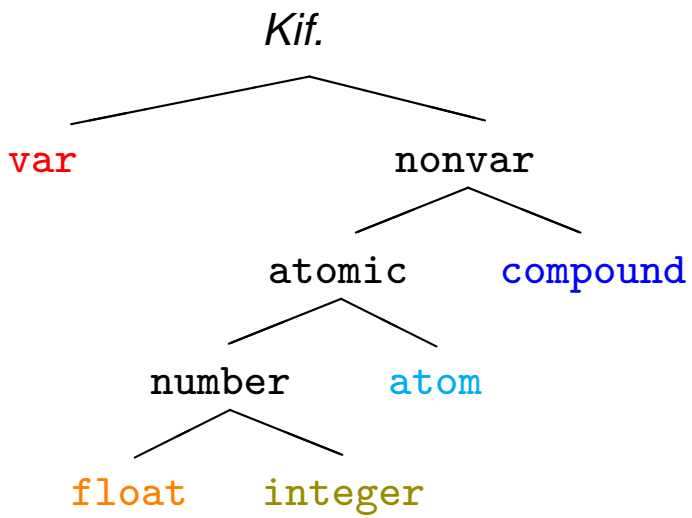
```
| ?- deriv(I, 0).
  ⟹     no
```

Tartalom

17 Programozás Prologban
- A funkcionális és logikai megközelítés összevetése
- Prolog bevezetés – példák
- A Prolog nyelv alapszintaxisa
- Listakezelő eljárások Prologban
- Nyomkövetés: 4-kapus doboz modell
- További vezérlési szerkezetek
- Magasabbrendű eljárások
- Megoldásgyűjtő beépített eljárások
- Operátorok
- Meta-logikai eljárások

Kifejezések osztályozása

- Kifejezésfajták – osztályozó beépített eljárások (ismétlés)



Szabványos eljárások:

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

- További osztályozó eljárások:
  - `simple(X)`: `X` nem összetett (konstans vagy változó);
  - `callable(X)`: `X` atom vagy struktúra (nem szám és nem változó);
  - `ground(X)`: `X` tömör, azaz nem tartalmaz behelyettesítetlen változót.

Osztályozó eljárások: a `length/2` példája (kieg. anyag)

- Példa: a `length/2` beépített eljárás egy lehetséges megvalósítása

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

Kifejezések szétszedése és összerakása – motiváló példa

- `Polinom ::= x` | szám | `Polinom + Polinom` | `Polinom * Polinom`
- Egy `P` polinom kiértékelése adott `x` behelyettesítés mellett (ismétlés):

```
% value_of(+P, +XV, ?V): az x = XV helyettesítéssel P értéke V.
value_of0(x, X, V) :- V = X.
value_of0(N, _, V) :-
    number(N), V = N.

value_of0(P1+P2, X, V) :-
    value_of0(P1, X, V1),
    value_of0(P2, X, V2),
    V is V1+V2.
value_of0(Polinom, X, V) :-
    Polinom = *(P1,P2),
    value_of0(P1, X, V1),
    value_of0(P2, X, V2),
    PolinomV = *(V1,V2),
    V is PolinomV.
```

```
value_of(x, X, V) :- V = X.
value_of(N, _, V) :-
    number(N), V = N.

value_of(Polinom, X, V) :-
    Polinom =.. [Func,P1,P2],
    value_of(P1, X, V1),
    value_of(P2, X, V2),
    PolinomV =.. [Func,V1,V2],
    V is PolinomV.
```

- `value_of/3` minden az `is/2` által elfogadott bináris függvényre működik!

```
| ?- value_of(exp(100,min(x,1/x)), 2, V).   ⟹   V = 10.0 ? ; no
```

Az *univ* beépített eljárás

- Kiindulás: `| ?- K=F(A,B).` $\Rightarrow$ szintaxis-hiba, helyette: `K=..[F,A,B]`, pl.:

```
| ?- el(a,b,10) =.. L.        ⟹   L = [el,a,b,10]
| ?- Kif =.. [el,a,b,10].     ⟹   Kif = el(a,b,10)
| ?- alma =.. L.              ⟹   L = [alma]
```

- Az *univ* eljárás hívási mintái: `+Kif =..  ?Lista`
  `-Kif =..  +Lista` (`Lista` zárt végű!)
- Az eljárás jelentése:
  - `Kif` = $Fun(A_1, \ldots, A_n)$ és `Lista` = $[Fun, A_1, \ldots, A_n]$, ahol Fun egy névkonstans és $A_1, \ldots, A_n$ tetszőleges kifejezések; vagy
  - `Kif` = C és `Lista` = $[C]$, ahol C egy (szám- vagy név)konstans.
- További példák:

```
| ?- 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]]

          (SWI Prologban:)  ⟹   L=['[|]',a,[b,c]]
```

Indexelés (áttekintés)

- Mi az indexelés?
  - egy hívásra alkalmazható (illeszthető fejű) klózok gyors kiválasztása,
  - egy eljárás klózainak fordítási idejű csoportosításával.
- A legtöbb Prolog rendszer, így a SICStus Prolog is, az első fej-argumentum alapján indexel (first argument indexing).
- Az indexelés alapja az első fejargumentum külső funktora:
  - `C` szám vagy névkonstans esetén `C/0`;
  - `R` nevű és `N` argumentumú struktúra esetén `R/N`;
  - változó esetén nem értelmezett (minden funktorhoz besoroljuk).
- Az indexelés megvalósítása:
  - Fordításkor minden funktor $\Rightarrow$az alkalmazható klózok listája
  - Futáskor konstans idő alatt elő tudjuk vennie a megfelelő klózlistát
  - *Fontos:* ha egyelemű a lista, nem hozunk létre választási pontot!
- Például `szuloje('István', X)` kételemű klózlistára szűkít, de `szuloje(X, 'István')` mind a 6 klózt megtartja (mert a SICStus Prolog csak az első argumentum szerint indexel)

Struktúrák kezelése: a `functor/3` eljárás (kieg. anyag)

- `functor/3`: kifejezés funktorának, adott funktorú kifejezésnek az előállítása
  - Hívási minták: `functor(-Kif, +Név, +Argszám)`
    `functor(+Kif, ?Név, ?Argszám)`
  - Jelentése: `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 kezelése: az `arg/3` eljárás (kieg. anyag)

- `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)
```

Alkalmazás: részkifejezések keresése (kieg. anyag)

- A feladat: adott egy tetszőleges kifejezés, soroljuk fel a benne levő számokat, és minden szám esetén adjuk meg az ún. *kiválasztóját*!
- Egy részkifejezés kiválasztója egy olyan lista, amely megadja, hogy sorra mely argumentumpozíciók mentén juthatunk el hozzá.
- Az `[`$i_1, i_2, \ldots, i_k$`]` $k \geq 0$ lista egy `Kif`-ből az i_1-edik argumentum i_2-edik argumentumának, ... i_k-adik argumentumát választja ki.
  (Az `[]` kiválasztó `Kif`-ből `Kif`-et választja ki.)
- Pl. `a*b+f(5,8,7)/c`-ben `b` kiválasztója `[1,2]`, `7` kiválasztója `[2,1,3]`.

```
% kif_szám(?Kif, ?N, ?Kiv): Kif Kiv kiválasztójú része az N szám.
kif_szám(X, X, []) :- number(X).
kif_szám(X, N, [I|Kiv]) :-  compound(X),
        X =.. [_F|Args], nth1(I, Args, X1),                    % (*)
        kif_szám(X1, N, Kiv).
```

```
| ?- kif_szám(f(5,[8,b]), Sz, K).⟹   Sz = 5, K = [1]  ? ;
                                     Sz = 8, K = [2,1] ? ; no
```

- A `(*)` sor helyett ez is állhat:

```
        functor(X, _F, ArgNo), between(1, ArgNo, I), arg(I, X, X1),
```

Atomok szétszedése és összerakása

- `atom_codes/2`: névkonstans és karakterkód-lista közötti átalakítás
  - Hívási minták: `atom_codes(+Atom, ?KódLista)`
    `atom_codes(-Atom, +KódLista)`
  - Jelentése: `Atom` karakterkódjainak a listája `KódLista`.
- 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
```

- Az `atom_codes(Atom, KódLista)` beépített eljárás végrehajtása:
  - Ha `Atom` adott (bemenő), és a $c_1 c_2 \ldots c_n$ karakterekből áll, akkor `KódLista`-t egyesíti a $[k_1, k_2, \ldots, k_n]$ 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.

Atomok kezelése: példák (kieg. anyag)

- Keresés névkonstansokban

```
% Atom-ban a Rész nem üres részatom kétszer ismétlődik.
dadogó_rész(Atom, Rész) :-
    atom_codes(Atom, Cs),
    Ds = [_|_],
    append([_,Ds,Ds,_], Cs), % append/2, lásd library(lists)
    atom_codes(Rész, Ds).
```

```
| ?- 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 ?
```

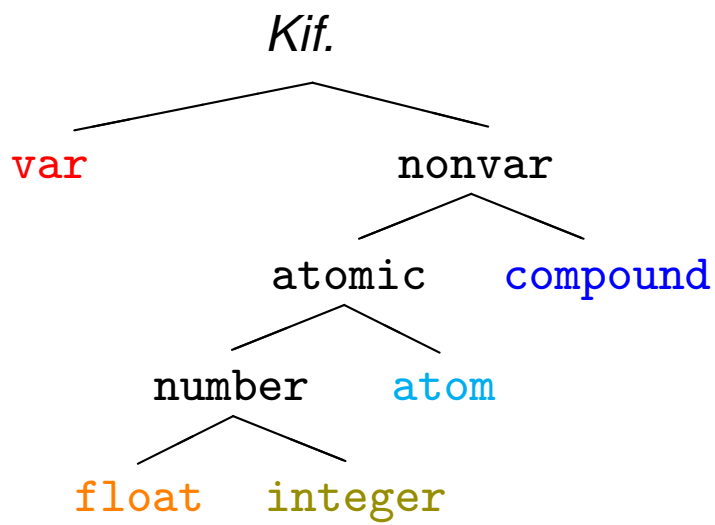
Számok szétszedése és összerakása

- `number_codes/2`: szám és karakterkód-lista közötti átalakítás
  - Hívási minták: `number_codes(+Szám, ?KódLista)`
    `number_codes(-Szám, +KódLista)`
  - Jelentése: Igaz, ha `Szám` tizes számrendszerbeli alakja a `KódLista` karakterkód-listának felel meg.
- 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 (mert a szám adott! :-)
```

- A `number_codes(Szám, KódLista)` beépített eljárás végrehajtása:
  - Ha `Szám` adott (bemenő), és a $c_1 c_2 \ldots c_n$ karakterekből áll, akkor `KódLista`-t egyesíti a $[k_1, k_2, \ldots, k_n]$ 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.

Prolog kifejezések általános rendezése: a $\prec$ reláció



A különböző kif.-fajták sorrendje:

$$\texttt{var} \prec \texttt{float} \prec \texttt{integer} \prec \texttt{atom} \prec \texttt{compound}$$

Egy kifejezésfajtán belüli sorrendezés szabályai:

- Változók: rendszerfüggő (pl. memóriacím alapján)
- Egész és lebegőpontos számok: szokásosan ($x \prec y \Leftrightarrow x < y$)
- Atomok: lexikografikus sorrend ($\texttt{abc} \prec \texttt{abcd}$, $\texttt{abcv} \prec \texttt{abcz}$)
- Összetett kif.-ek: $\texttt{név}_a(a_1, \ldots, a_n) \prec \texttt{név}_b(b_1, \ldots, b_m) \Leftrightarrow$
  1. $n < m$, pl. $\texttt{p(x,s(u,v,w))} \prec \texttt{a(b,c,d)}$, vagy
  2. $n = m$, és $\texttt{név}_a \prec \texttt{név}_b$ (lexikografikusan), pl. $\texttt{a(x,y)} \prec \texttt{p(b,c)}$, vagy
  3. $n = m$, $\texttt{név}_a = \texttt{név}_b$, és az első olyan i-re melyre $a_i \neq b_i$, $a_i \prec b_i$, pl. $\texttt{r(1,u+v,3,x)} \prec \texttt{r(1,u+v,5,a)}$

Kifejezések összehasonlítása – beépített eljárások

- Beépített eljárások tetszőleges kifejezések összehasonlítására:

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

- Az összehasonlítás mindig a belső (kanonikus) alak szerint történik:

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

- Beépített elj. tetszőleges lista rendezésére: `sort(+L, ?S)`
  Jelentése: az `L` lista `@<` szerinti rendezése `S`,
  `==/2` szerint azonos elemek ismétlődését kiszűrve.

```
        | ?- sort([a,c,a,b,b,c,c,b,d,a(2,3),c(1),2.0,1,X], S).
        S = [X,2.0,1,a,b,c,d,c(1),a(2,3)] ? ; no
(SWI):S = [X,1,2.0,a,b,c,d,c(1),a(2,3)]. :-(
```

Összefoglalás: a Prolog egyenlőség-szerű beépített eljárásai

- `U = V`: `U` egyesítendő `V`-vel.
  Soha sem jelez hibát.
- `U == V`: `U` azonos `V`-vel.
  Soha sem jelez hibát és soha sem helyettesít be.
- `U =:= V`: Az `U` és `V` aritmetikai kifejezések értéke megegyezik.
  Hibát jelez, ha `U` vagy `V` nem (tömör) aritmetikai kifejezés.
- `U is V`: `U` egyesítendő a `V` aritmetikai kifejezés értékével.
  Hiba, ha `V` nem (tömör) aritmetikai kifejezés.
- (`U =.. V`: `U` „szétszedettje” a `V` lista)

```
| ?- X = 1+2.       ⟹   X = 1+2
| ?- 3 = 1+2.       ⟹   no
| ?- X == 1+2.      ⟹   no
| ?- 3 == 1+2.      ⟹   no
| ?- +(1,2)==1+2    ⟹   yes
| ?- X =:= 1+2.     ⟹   hiba
| ?- 1+2 =:= X.     ⟹   hiba
| ?- 2+1 =:= 1+2.   ⟹   yes
| ?- 2.0 =:= 1+1.   ⟹   yes
| ?- 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
| ?- 1+2 =.. X.     ⟹   X = [+,1,2]
| ?- X =.. [f,1].   ⟹   X = f(1)
```

Összefoglalás: a Prolog nem-egyenlő jellegű beépített eljárásai

A nem-egyenlőség jellegű eljárások soha sem 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.

Az egyesítés kiegészítése: előfordulás-ellenőrzés, *occurs check*

- Kérdés: `X` és `s(X)` egyesíthető-e?
  - A matematikai válasz: *nem*, egy változó nem egyesíthető egy olyan struktúrával, amelyben előfordul (ez az előfordulás-ellenőrzés).
  - Az ellenőrzés költséges, ezért alaphelyzetben nem alkalmazzák (emiatt ún. ciklikus kifejezések keletkezhetnek)
  - Szabványos eljárásként rendelkezésre áll: `unify_with_occurs_check/2`
  - Kiterjesztés (pl. SICStus): az előfordulás-ellenőrzés elhagyása miatt keletkező ciklikus kifejezések tisztességes kezelése.
- Példák:

```
| ?- X = s(1,X).
        X = s(1,s(1,s(1,s(1,s(...))))) ?
| ?- unify_with_occurs_check(X, s(1,X)).
        no
| ?- X = s(X), Y = s(s(Y)), X = Y.
        X = s(s(s(s(s(...))))), Y = s(s(s(s(s(...))))) ?
```