4.6. Megoldásgyűjtő beépített eljárások

Az egyik gyakran használt megoldásgyűjtő eljárás a `findall(?Gyűjtő, +Cél, ?Lista)`. A `findall` a `Cél` kifejezést eljáráshívásként értelmezi, meghívja és minden egyes megoldásához előállítja `Gyűjtő` egy másolatát (vagyis a megoldásban levő változókat új változókra cseréli). Végül ezeket a `Gyűjtő` másolatokat egy listába összegyűjti és egyesíti `Lista`-val:

```
| ?- findall(X, (member(X, [1,7,8,3,2,4]), X>3), L).
L = [7,8,4] ?
| ?- 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] ?
```

A `findall/3` használatával egy a megoldásokat felsoroló eljárásból könnyen előállítható az összes megoldást összegyűjtő program. Az előző alfejezetbeli példa esetében:

```
khatvanyok(Max, Hk) :-
        findall(H, khatvany(Max, H), Hk).
```

A `bagof(?Gyűjtő, +Cél, ?Lista)` eljárás hasonlít a `findall`-hoz, a `Cél` kifejezést eljáráshívásként értelmezi, és összegyűjti a megoldásait. Azonban ha a `Cél`-ban vannak olyan üres változók, amelyek a `Gyűjtő`-ben nem szerepelnek, akkor ezek minden egyes behelyettesítését felsorolja és külön-külön mindegyikhez összegyűjti a `Gyűjtő` összes megoldását `Lista`-ba. Például:

```
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
```

Ha a `bagof` eljárás második argumentuma $\texttt{V}_1$`^ ...`$\texttt{V}_n$`^Cél` alakú (egzisztenciális kvantifikálás: léteznek olyan $\texttt{V}_1,\ldots,\texttt{V}_n$ értékek, hogy `Cél` igaz), akkor a $\texttt{V}_1,\ldots,\texttt{V}_n$ változók behelyettesítéseit nem sorolja fel. Ha a `Cél`-beli összes szabad változót így felsoroljuk, akkor a `findall` eljáráshoz hasonló viselkedést kapunk:

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

Jó példa egzisztenciális kvantor használatára a következő `fokszámai` eljárás, ahol egy irányított gráf minden pontjának ki-fokát határozzuk meg, majd gyűjtjük egy listába.

```
% G gráf fokszámlistája FL. A fokszámlista olyan A-F
% párokból áll, ahol A a gráf egy pontja,
% és F>0 az A pont fokszáma.
fokszámai(G, FL) :-
        bagof(A-F, Vk^(bagof(V, member(A-V, G), Vk), length(Vk, F)), FL).

| ?- gráf(_G), fokszámai(_G, FL).
FL = [a-2,b-2,c-1] ? ;
no
```

A kvantort kiküszöbölhetjük egy segédeljárás bevezetésével:

```
% Az A pont foka a G gráfban F>0.
pont_foka(A, G, F) :-
        bagof(V, member(A-V, G), Vk),
        length(Vk, F).

fokszámai(G, FL) :-
        bagof(A-F, pont_foka(A, G, F), FL).
```

Bár a `bagof` és `findall` hasonló eljárások, fontos megemlíteni néhány további különbséget közöttük:

- Ha `Célnak` nincs megoldása, `findall` üres listát ad, `bagof` meghiúsul.
- Ha `Gyűjtő` nem tömör (van benne üres változó), akkor
  - `findall` ezeket megoldásonként szisztematikusan új változókra cseréli,
  - `bagof` megőrzi a változókat.
- A `bagof` végrehajtása időigényesebb.

Ezeket a különbségeket mutatják be az alábbi példák:

```
| ?- findall(X, (between(1, 5, X), X<0), L).
L = [] ?
yes
| ?- 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)] ?
yes
| ?- bagof(S, member(S, [f(X,X),g(X,Y)]), L).
L = [f(X,X),g(X,Y)] ?
yes
```

A megoldásgyűjtő eljárások csoportjába tartozik még a `setof(?Gyűjtő, :+Cél, ?Lista)`, amely tulajdonképpen ugyanaz mint `bagof`, de az eredménylistát rendezi (az ismétlődések kiszűrésével). A rendezéshez a minden Prolog kifejezésre alkalmazható `@<` összehasonlító beépített eljárást használja (amiről részletesebben a 4.7.4. pontban lesz szó).

Az előző példát tekintve, a gráf pontjainak rendezett listáját nyerhetjük a `setof` eljárás segítségével.

```
gráf([a-b,a-c,b-c,c-d,b-d]).

% Gráf egy pontja P.
pontja(Gráf, P) :-
        member(P-_, Gráf).
pontja(Gráf, P) :-
        member(_-P, Gráf).

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

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

4.7. Beépített meta-logikai eljárások

Meta-logikai eljárásoknak az olyan, a „tisztán logikai” módszereken túlmutató beépített eljárásokat hívjuk, amelyek

a. a Prolog kifejezések pillanatnyi behelyettesítettségi állapotát tekintik:

- általános kifejezések osztályozása
- általános kifejezések rendezése

b. kifejezéseket szétszednek vagy összeraknak

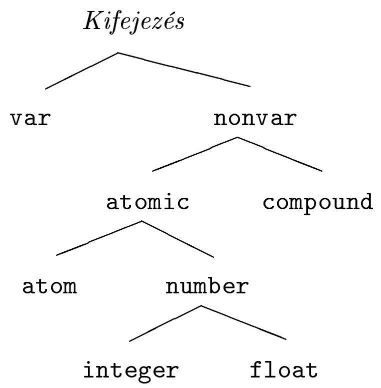
- (struktúra) kifejezés $\Longleftrightarrow$ név és argumentumok
- atomok és számok $\Longleftrightarrow$ karaktereik

Azt, hogy az **a.** típusú eljárások nem tisztán logikaiak, az is jól mutatja, hogy ezek eredménye általában sorrend-függő. Erre mutatunk két példát (a `var` osztályozó és a `@<` összehasonlító eljárások esetére):

```
| ?- var(X) /* X változó? */, X = 1.
X = 1 ?
yes
| ?- X = 1, var(X).
no
| ?- X @< 3 /* X megelőzi 3-t? */, X = 4.
   % a változók megelőzik a nem változó kifejezéseket
X = 4 ?
yes
| ?- X = 4, X @< 3.
no
```

4.7.1. Kifejezések osztályozása

Az osztályozásra szolgáló eljárások megértéséhez először vizsgáljuk meg a kifejezés-osztályok fastruktúráját.



Az osztályok feltüntetett nevei egyben egyargumentumú ellenőrző eljárások is. Ezen eljárásokkal dönthetjük el egy adott kifejezésről, hogy az beletartozik-e az adott osztályba vagy sem. Hogy mire használhatók az osztályozó eljárások?

A teljesség igénye nélkül csak pár gyakori példát említünk. A `var`, `nonvar` — többirányú eljárások esetén — a különböző irányok elágaztatásánál használható. A `compound`, a `number`, és az `atom` eljárás pedig olyan esetekben alkalmazható, amikor nem-megkülönböztetett únió típusú adatokkal dolgozunk.

P12 Példa: A `length/2` beépített eljárás megvalósítása

Az alábbi példa a `length/2` beépített eljárás megvalósítását mutatja be. A `var` osztályozó eljárás segítségével ellenőrizzük, hogy melyek a be- ill. kimenő argumentumok, és ennek megfelelően más-más kódrészt futtatunk.

```
% 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).
Len = 2 ? ;
no
| ?- length([1,2], 3).
no
| ?- length(L, 3).
L = [_A,_B,_C] ? ;
no
| ?- length(L, Len).
L = [], Len = 0 ? ;
L = [_A], Len = 1 ? ;
L = [_A,_B], Len = 2 ? ;
L = [_A,_B,_C], Len = 3 ?
L = [_A,_B,_C,_D], Len = 4 ?
```

P13 Példa: Szimbolikus kifejezés-feldolgozásra

Az alábbi `deriv` eljárás egy egyszerű deriváló program, mellyel a `+, -, *, /` műveletekkel atomokból és számokból felépített kifejezések deriválását lehet elvégezni, de apró módosítás segítségével ki lehet terjeszteni az eljárást például a szinusz, koszinusz, exponenciális függvényekre is. Itt az `atomic` eljárást használjuk annak eldöntésére, hogy a deriválandó konstans-e.

```
% deriv(Kif, X, D):  Kif-nek az X atom szerinti
%   deriváltja D.  Kif a +, -, *, /  műveletekkel
%   atomokból és számokból felépített kifejezés.
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
| ?-
```

4.7.2. Struktúrák szétszedése és összerakása

Az univ eljárás

Az univ eljárást két különböző feladatra használhatjuk, amelyek lényegében egymás ellentettjei:

- struktúrák szétszedése
- struktúrák összerakása, létrehozása egyszerűbb struktúrákból

Tehát az univ tulajdonképpen egy olyan kétargumentumú reláció A és B között, ahol A és B pontosan akkor állnak relációban, ha A „szétbontottja” B. Az univ eljárás szintaxisa:

```
+Kif =.. ?Lista
-Kif =.. +Lista
```

Az univ eljárás a `Kif` kifejezést egy olyan `Listá`ra bontja (ill. egy olyan `Listá`ból építi fel), amelyiknek az első eleme a kifejezés neve, a többi eleme pedig a kifejezés argumentumai, a megfelelő sorrendben.

```
| ?- el(a,b,10) =.. L.
L = [el,a,b,10] ?
| ?- el(a,b,10) =.. [F|As].
F = el, As = [a,b,10] ?
| ?- Kif =.. [/,1,2+3].
Kif = 1/(2+3) ?
| ?- [a,b,c] =.. L.
L = ['.',a,[b,c]] ?
```

Emlékeztetjük az olvasót arra, hogy a Prolog nyelv nem engedi meg struktúranév-pozícióban a változót, tehát `Kif = S(X,Y)` nem megengedett! Ehelyett használható viszont a `Kif =..  [S,X,Y]` hívás. Az univ persze ennél is „többet tud”, hiszen megengedi a kifejezés szétbontását akkor is, ha nem ismert az argumentumok száma (ez emlékeztet a `C` nyelv *vararg* nyelvi szerkezetére).

Az univ eljárás építőelemei

Az univ eljárás két egyszerűbb, beépített eljárásra épül, melyek külön-külön is használhatók. Az egyik eljárás a `functor`. Ez az eljárás a `Kif` kifejezés, és funktora, `Név/Argszám` közötti kapcsolatot írja le. Így tehát használható egy adott `Kif` kifejezés nevének (`Név`) és argumentumszámának (`Argszám`) meghatározására. A `functor` az univhoz hasonlóan egy kétirányú eljárás, fordítva is működik, tehát segítségével létre is tudunk hozni egy adott nevű és argumentumszámú kifejezést. A kifejezés az adott funktorúak közül a *legáltalánosabb*, tehát argumentumai különböző változók lesznek. A `funktor` eljárás használata:

```
functor(-Kif, +Nev, +ArgSzam)
functor(+Kif, ?Nev, ?ArgSzam)
```

Megjegyzés: A számok és atomok 0-argumentumúnak számítanak.

```
| ?- functor(szemely(kiss, pal, 1990), Nev, Aszam).
Nev = szemely, Aszam = 3 ?
yes
| ?- functor([a,b,c], Nev, Aszam).
Nev = '.', Aszam = 2 ?
yes
| ?- functor(Str, szemely, 3).
Str = szemely(_A,_B,_C) ?
yes
| ?- functor(25, Nev, Aszam).
Nev = 25, Aszam = 0 ?
yes
```

Az argumentumok vizsgálatára a `arg(+Sorszám, +Kif, ?Arg)` eljárás áll rendelkezésünkre. Ez akkor sikerül, ha a `Kif` kifejezés `Sorszám`-adik argumentuma `Arg`-gal egyesíthető:

```
| ?- arg(2, szemely(kiss, pal, 1990), Arg).
Arg = pal ?
yes
| ?- arg(1, [a,b,c], A_1), arg(2, [a,b,c], A_2).
A_1 = a, A_2 = [b,c] ?
yes
```

Az univ visszavezethető a `functor` és `arg` eljárásokra. Például a

```
Kif =.. [F,A1,A2]
```

hívás helyettesíthető a következő hívássorozattal:

```
functor(Kif, F, 2), arg(1, Kif, A1), arg(2, Kif, A2)
```

Példa:

```
| ?- functor(Str, szemely, 3), arg(1, Str, kiss), arg(2, Str, pal).
Str = szemely(kiss,pal,_A) ?
yes
| ?- Str =.. [szemely,kiss,pal,_].
Str = szemely(kiss,pal,_A) ?
yes
```

A P19 példa bemutatja, hogyan lehet ezt a visszavezetést általánosan elvégezni.

Az univ eljárás alkalmazásai

Legyen feladatunk az, hogy egy `+` és `*` operátorokat tartalmazó kifejezésben a csak számokból álló részkifejezéseket helyettesítsük a részkifejezés számértékével. Az első példa az univ nélküli, a második pedig az univ segítségével történő megvalósítást mutatja be:

P14 Példa: Kifejezések egyszerűsítése, 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).

| ?- eriv((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 ?
```

P15 Példa: Kifejezések egyszerűsítése, 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).
```

Vegyük észre, hogy a második megoldás sokkal általánosabb: nem csak a `+` és `*` operátorok esetén, hanem az `is/2` által elfogadott összes kétargumentumú műveletre működik.

Az alábbi példák az univ általános kifejezés-bejáró képességét demonstrálják. Az első egy kifejezés kiirató, a másik egy változómentesítő program.

P16 Példa: Speciális kifejezés kiiratás

A feladat az, hogy egy tetszőleges kifejezést kiírjunk úgy, hogy

- az összetett kifejezések alap-struktúra alakban jelennek meg, de
- a kétargumentumú operátorok infix (zárójeles) formában íródnak ki.

Annak eldöntésére, hogy egy `A` atom `F` fajtájú `P` prioritású operátor-e a `current_op(P, F, A)` beépített eljárást használjuk.

```
% Kif-et kiírja a fenti speciális alakban
alapki(Kif) :-
        compound(Kif), !, Kif =.. [Func,A1|ArgL],
        (   current_op(_, Kind, Func),     % Func operátor?
            (Kind = xfy ; Kind = yfx ; Kind = xfx),
            ArgL = [A2] ->      % kétargumentumú használat?
            write('('), alapki(A1), write(' '),
            write(Func), write(' '), alapki(A2), write(')')
        ;   write(Func), write('('), alapki(A1),
            arglistaki(ArgL), write(')')
        ).
alapki(Kif) :- write(Kif).

% Az [A1,...,An] listát ",A1,...,An" alakban kiírja.
arglistaki([]).
```

```
arglistaki([A|AL]) :-
        write(','), alapki(A), arglistaki(AL).

| ?- alapki(1+2+3).
((1 + 2) + 3)

| ?- alapki([1,2]).
.(1,.(2,[]))

| ?- alapki(f(X,2,g(X))).
f(_117,2,g(_117))

| ?- alapki(f(+a, b*c*d, e)).
f(+(a),((b * c) * d),e)
```

P17 Példa: Kifejezés változómentesítése

Tekintsük a következő beépített eljárást:

`numbervars(?Kif, +N0, ?N)`: A `Kif` kifejezést tömörré (`ground`) teszi úgy, hogy a benne szereplő különböző változókat sorra a `'$VAR'`(n_0), `'$VAR'`(n_0+1) ... `'$VAR'`(n) struktúrákkal helyettesíti, ahol $n0$ = `N0`, és `N`-t az n+1 értékkel egyesíti.

Ezt az eljárást például akkor használhatjuk, ha kiiratáskor a változók belső nevei helyett (_⟨szám⟩) egy rövidebb alfanumerikus változónevet szeretnénk megjeleníteni. A `write/1` beépített eljárás ugyanis a `'$VAR'(0)`, `'$VAR(1)`, ... struktúrákat rendre az `A`, `B`, ... változónevekként jeleníti meg. Ha a `numbervars` eredményeként előálló kifejezés tényleges formáját szeretnénk látni, akkor a `write_term/2` beépített eljárást kell használnunk, a `numbervars(false)` opcióval (a `quoted(true)` opciót pedig azért használjuk az alábbi példában, hogy a `'$VAR'` név aposztrofjelek között jelenjen meg).

```
| ?- Kif = [f(_X),g(_),_X], numbervars(Kif, 0, N),
     write_term(Kif, [quoted(true),numbervars(false)]).
     [f('$VAR'(0)),g('$VAR'(1)),'$VAR'(0)]
N = 2, Kif = [f(A),g(B),A] ?
```

Az univ eljárás segítségével most megírjuk a `numbervars` beépített eljárás egy változatát:

```
% Term változóit sorra a '$myvar'(N0), '$myvar'(N0+1),  ..., '$myvar'(n)
% struktúrákra cseréli, és N = n+1.
numbervars1(Term, N0, N) :- var(Term), !,
        Term = '$myvar'(N0), N is N0+1.
numbervars1(Term, N0, N) :-
        Term =.. [_|Args],
        number_list(Args, N0, N).

% number_list(List, N0, N): a List listában szereplő változókat sorra
% a '$myvar'(N0) ... '$myvar'(n) struktúrákra cseréli, és N = n+1.
number_list([], N, N).
number_list([A|As], N0, N) :-
        numbervars1(A, N0, N1),
        number_list(As, N1, N).

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

Az imént bemutatott `numbervars1` alkalmazása a következő példa.

P18 Példa: Két kifejezés azonosságának vizsgálata

Két kifejezést akkor mondunk azonosnak, ha változó-behelyettesítés *nélkül* egyesíthetőek, azaz, ha az egyik változót tartalmaz, akkor a másik ugyanott ugyanazt a változót tartalmazza. Az `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).

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

Végül bemutatjuk az univ eljárás egy megvalósítását a `functor` és `arg` segítségével.

P19 Példa: Az univ eljárás Prolog megvalósítása

```
Kif =..  [Nev|ArgL] :-
      var(Kif), !,
      atomic(Nev),
      length(ArgL, N),
      functor(Kif, Nev, N),
      arglista(0, Kif, N, ArgL).
Kif =..  [Nev|ArgL] :-
      functor(Kif, Nev, N),
      arglista(0, Kif, N, ArgL).

% arglista(N0, Str, N, L): Str N0 utáni argumentumainak listája
% L, ahol N Str argumentumszáma (Str lehet konstans, amikoris N=0)

arglista(N, _Str, N, []) :- !.
arglista(N0, Str, N, [A|L]) :-
      N0 < N, N1 is N0+1,
      arg(N1, Str, A),
      arglista(N1, Str, N, L).
```

Ez a definíció a hibás hívások kezelésével nem foglalkozik, hiba esetén meghiúsul, vagy valamelyik részeljárás fog hibát jelezni.

4.7.3. Konstansok szétszedése és összerakása

Az univ eljárással kifejezéseket bontottunk fel kifejezésekre, de az univ számára a konstans kifejezések már lényegében felbonthatatlanok. A konstansok karakterekre bontását, ill. ezekből való felépítését teszik lehetővé az alábbi beépített eljárások.

Az `atom_codes(Atom, KódLista)` eljárás a következőképpen működik. Ha híváskor `Atom` ismert, és a $c_1c_2...c_n$ karakterekből áll, akkor a rendszer `KódListát` egyesíti egy $[k_1, k_2, ..., k_n]$ számlistával, ahol k_i a c_i karakterkódja. Ha `Atom` változó, akkor a `KódLista` karakterkód-listából összerak egy nevet, és azt írja be `Atom`-ba.

```
| ?- atom_codes(ab, Cs).
Cs = [97,98] ?

| ?- Cs = [0'b,0'c], atom_codes(Atom, Cs).
Cs = [98,99], Atom = bc ?
```

A `number_codes(Szám, KódLista)` eljárás hasonló kétirányú relációt valósít meg számok esetében. Tehát, ha a `Szám` adott, és tizes számrendszerbeli alakja a $c_1c_2...c_n$ karakterekből áll, akkor `KódLista =` $[k_1, k_2, ..., k_n]$ lesz, ahol k_i a c_i karakterkódja. Ha `Szám` változó, akkor a `KódLista` karakterkód-listából összerak egy számot, és azt írja be `Szám`-ba.

```
| ?- number_codes(1234, Cs).
Cs = [49,50,51,52] ?

| ?- Cs = [0'1,0'2], number_codes(Num, Cs).
Cs = [49,50], Num = 12 ?
```

A konstansok szétszedésére és összerakására szolgáló eljárásokkal szövegmanipulációs feladatokat oldhatunk meg. Ezekre mutatunk most egy-két példát:

```
| ?- use_module(library(lists)).

| ?- atom_codes(ab, _L), reverse(_L, _R),
     append(_L, _R, LR), atom_codes(X, LR).
X = abba, LR = [97,98,98,97] ?

% Rész olyan részatomja Atom-nak, amelyet egy
% vele közvetlenül megegyező részatom követ.
dadogó_rész(Atom, Rész) :-
        atom_codes(Atom, Cs),
        dadogó(Cs, Ds),
        atom_codes(Rész, Ds).

% dadogó(L, D): D olyan nem üres részlistája L-nek,
% amelyet egy vele megegyező részlista követ.
dadogó(L, D) :-
        D = [_|_],
        append(_, Farok, L),
        append(D, Vég, Farok),
        append(D, _, Vég).

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

4.7.4. Kifejezések rendezése: szabványos sorrend

A Prolog kifejezéseknek létezik egy szabványos sorrendje. Ez gyakran hasznos, pl. halmazokat rendezett listaként tudunk ábrázolni. A 4.6. pontban bemutatott `setof` eljárás is ezt az elvet használja: a megoldások halmazából kiszűri az egyformákat, a többit pedig rendezi a szabványos sorrend szerint.

A sorbarendezéshez szükség van egy összehasonlító relációra, amely két *tetszőleges* Prolog kifejezés sorrendjét eldönti.

Vezessük be a szabványos rendezési relációt a következőképpen. Jelentse az $X \prec Y$ formula azt, hogy X megelőzi Y-t a szabványos rendezés szerint. A $\prec$ relációt a következőképpen definiáljuk:

1. Ha X és Y azonos, akkor $X \prec Y$ és $Y \prec X$ egyike sem igaz.
2. Ha X és Y típusa különböző, akkor a típus dönt: *változó* $\prec$ *lebegőpontos szám* $\prec$ *egész szám* $\prec$ *név* $\prec$ *struktúra*.
3. Ha X és Y különböző változók, akkor rendszerfüggő módon vagy $X \prec Y$, vagy $Y \prec X$ igaz.
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:
   (a) Ha X és Y aritása különböző, $X \prec Y \Leftrightarrow X$ aritása kisebb mint Y aritása.
   (b) Egyébként, ha a rekordok neve különböző, $X \prec Y \Leftrightarrow X$ neve $\prec Y$ neve.
   (c) Egyébként balról az első nem azonos argumentum dönt (lexikografikus rendezés).

Végtelen (ciklikus) kifejezésekre a fenti rendezés nem érvényes.

Kifejezések összehasonlítására az alábbi hat beépített eljárás szolgál:

`(==)/2, (\==)/2, (@<)/2, (@=<)/2, (@>)/2, (@>=)/2`

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

Ezen eljárások minden argumentuma tisztán bemenő tetszőleges kifejezés.

Fontos megjegyezni, hogy ezek az eljárások logikailag nem tiszták, hiszen a rendezés a pillanatnyi behelyettesítettségtől függ:

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

Azt is fontos szem előtt tartani, hogy mindig a szabványos belső struktúra-alak szerint rendezzük a kifejezéseket, pl.

```
| ?- [1, 2, 3, 4] @< struktúra(1, 2, 3).
```

sikerül (ld. 6(a) szabály), hiszen a bal oldalon `.(1,...)` struktúra áll.

Az alábbi példa az eddig megismert beépített eljárások segítségével valósítja meg a $\prec$ rendezési relációt.

P20 Példa: $\prec$ egy megvalósítása

```
% T1 megelőzi T2-t a szabványos sorrendben (lényegében T1 @< T2), de
% a változókat az első előfordulás szerint rendezi.
precedes(T1, T2) :-
        \+ \+ (numbervars1(T1-T2, 0, _), prec(T1, T2)).

% T1 megelőzi T2-t, ahol mindkettőben a változók '$myvar'(n)
% konstansokra vannak már cserélve.
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): T a C osztályba tartozik
% (0 -> változó, 1 -> lebegőpontos, 2 -> egész, 3 -> atom, 4 -> struktúra).
class('$myvar'(_), C) :-   !, C = 0.
class(T, C) :- float(T),   !, C = 1.
class(T, C) :- integer(T), !, C = 2.
class(T, C) :- atom(T),    !, C = 3.
class(_T, 4).

% S1 megelőzi S2-t (struktúra-kifejezésekre és atomokra).
struct_prec(S1, S2) :-
        functor(S1, F1, N1), functor(S2, F2, N2),
        (   N1 =:= N2 ->
            (   F1 = F2 ->
                args_prec(1, N1, S1, S2)
            ;   atom_prec(F1, F2)
            )
        ;   N1 < N2
        ).

% 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).
```

4.8. Egyenlőségfajták

Prologban sokfajta beépített eljárás van, amelyben az egyenlőség fogalma szerepel. Ennek az alfejezetnek a célja az ilyen eljárások egybegyűjtése és a köztük lévő különbségek áttekintése. Vessük össze az egyenlőség-szerű, majd a nemegyenlő-szerű eljárásokat, s végül nézzünk pár példát az elvek illusztrálására. (Az alábbi példákban `X` egy behelyettesítetlen változó.)

A Prolog egyenlőség-szerű beépített eljárásai:

- `U = V`: `U` egyesítendő `V`-vel. Soha sem jelez hibát. Pl. `X = 1+2` eredménye az `X = 1+2` behelyettesítés.
- `U == V`: `U` azonos `V`-vel. Soha sem jelez hibát és soha sem helyettesít be. Pl. `X == 1+2` meghiúsul.
- `U is V`: `U` egyesítendő a `V` aritmetikai kifejezés értékével. Hiba, ha `V` nem (tömör) aritmetikai kifejezés. Pl. `X is 1+2` eredménye az `X = 3` behelyettesítés.
- `U =:= V`: Az `U` és `V` aritmetikai kifejezések értéke megegyezik. Hibát jelez, ha `U` vagy `V` nem (tömör) aritmetikai kifejezés. Pl. `X =:= 1+2` hibát jelez.

A Prolog nemegyenlő-szerű beépített eljárásai (az alábbi eljárások egyike sem helyettesít be változót):

- `U \= V`: `U` nem egyesíthető `V`-vel. Soha sem jelez hibát. Pl. `X \= 1+2` meghiúsul.
- `U \== V`: `U` nem azonos `V`-vel. Soha sem jelez hibát. Pl. `X \== 1+2` sikerül.
- `U =\= V`: A `U` és `V` aritmetikai kifejezések értéke különbözik. Hibát jelez, ha `U` vagy `V` nem (tömör) aritmetikai kifejezés. Pl. `X =\= 1+2` hibát jelez.

Példák az egyenlőség-szerű eljárások használatára (`X` és `Y` behelyettesítetlen változók):

`U`	`V`	`U = V`	`U \= V`	`U == V`	`U \== V`	`U is V`	`U =:= V`	`U =\= V`
`1`	`2`	nem	igen	nem	igen	nem	nem	igen
`a`	`b`	nem	igen	nem	igen	hiba	hiba	hiba
`1+2`	`+(1,2)`	igen	nem	igen	nem	nem	igen	nem
`1+2`	`2+1`	nem	igen	nem	igen	nem	igen	nem
`1+2`	`3`	nem	igen	nem	igen	nem	igen	nem
`3`	`1+2`	nem	igen	nem	igen	igen	igen	nem
`X`	`1+2`	`X=1+2`	nem	nem	igen	`X=3`	hiba	hiba
`X`	`Y`	`X=Y`	nem	nem	igen	hiba	hiba	hiba
`X`	`X`	igen	nem	igen	nem	hiba	hiba	hiba

Jelmagyarázat

- igen ⇒ siker.
- nem ⇒ meghiúsulás.

Az alábbi párbeszéd bemutatja a fenti táblázatban szereplő példák egy részét.

```
| ?- X = 1+2.              % X egyesíthető a +(1, 2) struktúrával
X = 1+2 ?
yes
| ?- X == 1+2.             % egy változó nem azonos +(1, 2)-vel
no
| ?- X is 1+2.             % X egyesíthető 1 + 2 értékével
X = 3
yes
| ?- X =:= 1+2.            % X-et nem lehet kiértékelni
{INSTANTIATION ERROR: _32=:=1+2 - arg 1}
| ?- +(1,2) = 1+2.         % +(1,2) egyesíthető 1+2-vel...
```

```
yes
| ?- +(1,2) == 1+2.         %   és ráadásul azonos is
yes
| ?- +(1,2) is 1+2.         % +(1,2) nem egyesíthető 1+2 értékével
no
| ?- +(1,2) =:= 1+2.        % a két kifejezés értéke egyenlő
yes
| ?- .(1,'[]') == [1].      % a két struktúra azonos
yes
| ?-
```

4.9. Modularitás

Ebben az alfejezetben röviden ismertetjük a SICStus Prolog modulkezelésének alapelveit. A Prolog nyelvhez kidolgozott különféle modul-fogalmakat a 6.1 alfejezetben tekintjük majd át.

A SICStus Prolog minden eljárást valamilyen modulban helyez el. Amíg nem intézkedünk másképp, az eljárások alaphelyzetben a `user` modulba kerülnek.

Ha Prolog programunkat strukturálni kívánjuk, akkor ún. modul-állományokat kell létrehoznunk. Egy állományban egy modult tudunk elhelyezni, az állomány első programeleme egy modul-parancs kell, hogy legyen:

```
:- module(Modulnév, [Funktor1, Funktor2, ...]).
```

Itt *Funktor1*, ... a modulból exportálni kívánt eljárások funktorai (azaz Név/Aritás alakú kifejezések, ahol Név egy atom, Aritás egy egész).

Például, ha a korábban definiált `fennsík/3` eljárást egy modulba kívánjuk foglalni, akkor a szükséges eljárásokat be kell írnunk egy állományba, mondjuk `plato.pl`-be, és ennek az állománynak az elejére el kell helyeznünk a következő parancsot:

```
:- module(plató_keresés, [fennsík/3]).
```

Ha ezután be akarjuk tölteni ezt a modult, akkor a SICStus rendszer promptjánál ki kell adnunk egy `use_module` parancsot, argumentumában az állománynévvel:

```
:- use_module(plato).
```

Ez a parancs az adott állományban levő modult betölti, és az általa exportált összes eljárást importálja a kurrens modulba (példánkban a `user` modulba). Ezáltal az importált eljárások ebből a modulból hívhatókká válnak. Ugyanezt a beépített eljárást használhatjuk a SICStus könyvtárak betöltésére is, pl. a `lists` könyvtárat a következőképpen tölthetjük be:

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

A `use_module` beépített eljárásnak van egy kétargumentumú változata, ez betölti a modult, de csak azokat az eljárásokat importálja, amelyek funktorai szerepelnek a második argumentumbeli import-listában. Például:

```
:- use_module(library(lists), [last/2]).
```

csak a `last/2` eljárást fogja láthatóvá tenni, a többi könyvtári eljárást nem. Ekkor például lehet egy saját `append` eljárásunk, anélkül, hogy ez a könyvtári példánnyal összeütközésbe kerülne.

A `use_module` parancs szerepelhet egy állományban, akár egy modul-állományban is. Ez utóbbi esetben csak ebbe a modulba fogja importálni a betöltött eljárásokat. Ugyanazt a modul-állományt több modulba is betölthetjük, ez nem jár felesleges memória-foglalással, mivel a SICStus Prolog rendszer az eljárásokat csak egy példányban tárolja.

Megjegyezzük, hogy a SICStus Prolog modulfogalma nem szigorú. Bármely betöltött eljárás meghívható, ha az ún. modul-kvalifikált hívási formát használjuk, azaz az eljáráshívás elé írjuk a modulnevet, attól a kettőspont operátorral elválasztva. Ha például a fennsík-kereső programot a fent példaként idézett módon foglaltuk modulba, és azt betöltjük, akkor a `fennsík/3` eljárást modul-kvalifikálás nélkül tudjuk hívni, de a többi eljárást is meghívhatjuk, például:

```
| ?- plató_keresés:első_fennsík([1,2,2,3], 4, F, H, L).
```

A SICStus rendszernek ez a tulajdonsága különösen hasznos modularizált programok nyomkövetésénél.

Végezetül következzék néhány, a magasabbrendű eljárások használata során felmerülő, a modularitással kapcsolatos fontos tudnivaló. A magasabbrendű (meta-) eljárás egy olyan eljárás, amelynek egy másik eljárás az argumentuma. Tipikus példák a meta-eljárásokra a 4.6. pontban bemutatott `findall, bagof, setof` eljárások. Világos, hogy ha modulközi meta-eljárásokat írunk, azaz a meta-eljárás által meghívandó eljárás más modulban van, akkor szükség van arra, hogy az eljárás meta-argumentumát modul-kvalifikált módon adjuk át. Ezért a meta-eljárásokra egy `meta_predicate` deklarációt kell megadnunk, amelyben jelezzük, hogy melyek az eljárás-argumentumok.

A meta-deklaráció formája:

```
:- meta_predicate ⟨eljárásnév⟩(⟨argleíró1⟩, ...).
```

Itt az $\langle$argleíró$_i\rangle$ lehet a `:` jel, annak jelzésére, hogy az adott argumentum egy eljárás, vagy bármilyen más atom a többi argumentum jelzésére. Ez utóbbi helyeken szokás a be- ill. kimenő jellegre utaló jeleket elhelyezni (`+`,`-`,`?`).

Például a `bagof` beépített eljárásra a következő deklaráció vonatkozik (ezt természetesen nem kell megadni, a rendszer beépítve tartalmazza):

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

4.10. Magasabbrendű eljárások

A magasabrendű eljárások tehát olyan eljárások, melyek argumentuma lehet egy másik eljárás. A megoldásgyűjtő eljárások is ide tartoznak, hiszen az az eljárás, melynek a megoldását gyűjtjük szükségképpen argumentuma a gyűjtő eljárásnak.

Először nézzünk két példát arra, hogyan lehet a `findall` megoldásgyűjtő eljárás segítségével lista-műveleteket definiálni, rekurzió nélkül.

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

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

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

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

4.10.1. Meta-eljárások megoldásgyűjtő eszközökkel

Az előző példa általánosításaként megmutatjuk, hogy nem csak egy lista páros elemeit választhatjuk ki, hanem tetszőleges Prolog predikátum szerint is megszűrhetjük a listát. A felhasznált magasabbrendű eljárás itt ismét a `findall`:

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

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

A meta-argumentum meghívása a `call(Cél)` beépített eljárással történik, ez a `Cél` (atom vagy struktúra) kifejezést hívássá alakítja és végrehajtja. Megengedett, hogy hívásként egy `X` változó szerepeljen, ez ekvivalens `call(X)` hívással, tehát a fenti `findall` hívást így is írhatjuk:

```
        findall(X, (member(X, L), Pred), FL).
```

Megint a `findall` felhasználásával mutatunk be egy példát, amely egy lista minden elemére alkalmaz egy a lista elemein értelmezett leképezést. Ez a funkcionális nyelvekből jól ismert `map` fügvénynek megfelelő Prolog eljárás:

```
% Az L lista X elemeit Pred Y-ba képezi le.
% A kapott Y értékek listája ML.
:- meta_predicate map(+, ?, :, ?, -).
map(L, X, Pred, Y, ML) :-
        findall(Y, (member(X, L), Pred), ML).

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

4.10.2. Részlegesen paraméterezett eljárások

A fenti `map` eljárás második, harmadik és negyedik argumentuma felfogható egyetlen argumentumként — egy részlegesen paraméterezett eljárásként. Ez a három kifejezés együtt egy olyan predikátumot ír le, amelynek argumentumai `X` és `Y`, és törzse `Pred`. Ha explicitté tesszük ezt az eljárást:

```
negyzet(X, Y) :- Y is X*X.
```

akkor a hívási példa így alakul:

```
| ?- map([1,2,3,4], X, negyzete(X,Y), Y, Nk).
```

Elképzelhető, hogy az elvégzendő műveletekhez további paraméterekre is szükség van, mint az alábbi példában:

```
masodfoku(L0, P, Q, L) :-
        map(L0, X, masodfoku_ertek(P,Q,X,Y), Y, L).

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

Ha bevezetjük azt a konvenciót, hogy a magasabbrendű műveletekben érdekelt argumentumok mindig a predikátum utolsó argumentum-pozícióin vannak, akkor egyetlen kifejezéssel ábrázolhatjuk az elvégzendő magasabbrendű műveletet.

Célunk tehát egy `map/3` eljárás megírása, amelynek jelentése a következő:

```
% map(L0, PartialPred, L): Az L0 lista minden elemére alkalmazva a
% PartialPred által előírt leképezést kapjuk az L listát.
```

A `PartialPred` struktúrakifejezés által előírt leképezést úgy végezhetjük el, hogy a struktúrához két argumentumot hozzáveszünk. Ezek közül az első a leképezendő érték, a második pedig a leképezés eredménye. Az így kiegészített struktúrakifejezés meghívásával végezzük el az eredmény kiszámítását. A fenti két példa esetén a `map/3` hívása tehát a következőképpen alakul:

```
masodfoku(L0, P, Q, L) :-
        map(L0, masodfoku_ertek(P,Q), L).

negyzete(L0, L) :-
        map(L0, negyzet, L).
```

Látjuk tehát, hogy a `map` eljárásnak olyan kifejezéseket adunk át, amelyek önmagukban nem hívhatók meg, hanem csak úgy, hogy ezeket a kifejezéseket ellátjuk két további paraméterrel. Ezeket a kifejezéseket tehát joggal nevezhetjük részlegesen paraméterezett eljáráshívásoknak.

Mielőtt a `map/3` eljárást a következő alfejezetben bemutatnánk, néhány segédeljárást vezetünk be. Ezeknek az lesz a feladata, hogy egy részlegesen paraméterezett eljáráshívás és a hiányzó paraméterek alapján állítsa össze a „teljesen” paraméterezett eljáráshívást, és azt hajtsa is végre.

A `call1/2` egy paraméterrel egészit ki egy részlegesen paraméterezett eljáráshívást, a `call2/3` kettővel stb. Ezek az eljárások számos Prolog megvalósításban beépítettek, de a SICStusban nem. Ezért ezeket az alábbi módon kell definiálnunk:

```
% Pred utolsó argumentumként A-val kiegészítve igaz (sikeresen fut le).
call1(Pred, A) :-
        Pred =.. FArgs, append(FArgs, [A], FArgs1),
        Pred1 =.. FArgs1, call(Pred1).

% Pred utolsó két argumentumként az A és B kifejezésekkel kiegészítve igaz.
call2(Pred, A, B) :-
        Pred =.. FArgs, append(FArgs, [A,B], FArgs2),
        Pred2 =.. FArgs2, call(Pred2).

% Pred utolsó három argumentumként az A, B és C kifejezésekkel kiegészítve igaz.
call3(Pred, A, B, C) :-
        Pred =.. FArgs,
        append(FArgs, [A,B,C], FArgs3),
        Pred3 =.. FArgs3, call(Pred3).
```

4.10.3. Részleges paraméterezést használó magasabbrendű eljárások

A `call2/3` eljárás felhasználásával megírhatjuk a `map` rekurzív definícióját. Az előző megvalósításban a `map` a lista elemein a `member` eljárás alkalmazásával ment végig (minden egyes elem után visszalépéssel). Az alábbi változatban pedig az általánosabban alkalmazható rekurzív listabejárást alkalmazzuk.

```
% map(Xs, Pred, Ys): Az Xs lista elemeire a Pred transzformációt
% alkalmazva kapjuk az Ys listát.
map([X|Xs], Pred, [Y|Ys]) :-
        call2(Pred, X, Y),
        map(Xs, Pred, Ys).
map([], _, []).

negyzet(X, Y) :- Y is X*X.
```

```
| ?- map([1,2,3,4], negyzet, L).
L = [1,4,9,16] ? ;
no
```

A `map` eljárás megvalósítása során választhattunk, hogy melyik módszert használjuk: a `findall`-t használjuk vagy rekurzívan dolgozzuk fel a listát. A funkcionális nyelvekből ismert `fold` jellegű eljárások esetén, amelyek egy művelet ismételt elvégzését biztosítják, szükség van az előző müvelet eredményére, ezért nem élhetünk az elemeket egymástól függetlenül feldolgozó `findall`-os módszerrel. Így a `foldl` és `foldr` alábbi definíciójában részlegesen paraméterezett eljárásokat használunk, és a `call3` eljárás segítségével végezzük el a meta-predikátumok meghívását.

```
% foldl(Xs, Pred, Y0, Y): Az Xs elemeire balról
% jobbra alkalmazott, a Pred által leírt
% kétargumentumú függvény Y0 kezdőértékre
% alkalmazott eredménye Y.
foldl([X|Xs], Pred, Y0, Y) :-
        call3(Pred, X, Y0, Y1),
        foldl(Xs, Pred, Y1, Y).
foldl([], _, Y, Y).

% foldr(Xs, Pred, Y0, Y): Az Xs elemeire jobbról
% balra alkalmazott, a Pred által leírt függvény
% Y0 kezdőértékre alkalmazott eredménye Y.
foldr([X|Xs], Pred, Y0, Y) :-
        foldr(Xs, Pred, Y0, Y1),
        call3(Pred, X, Y1, Y).
foldr([], _, Y, Y).

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

Példák a fenti meta-eljárások használatára:

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

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

4.11. Dinamikus adatbáziskezelés

A Prolog nyelv lehetővé teszi, hogy az ún. dinamikus eljárásokat futási időben módosítsuk, hozzáadjunk ill. elvegyünk klózokat. Ezek az eljárások ugyan általában lassabbak, mint a statikusak, de jelentősen megnövelik a programozó lehetőségeit, szabadságát. A dinamikus eljárásokat a

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

deklaráció vezeti be, ennek az adott eljárás első (a program szövegében megadott) klóza előtt kell szerepelnie.

4.11.1. Beépített eljárások

A következő eljárások segítségével klózokat vehetünk fel, kérdezhetünk le illetve törölhetünk a programból.

Az `asserta(:@Klóz)`[1] és az `assertz(:@Klóz)` eljárások a `Klóz` kifejezést klózként értelmezik és felveszik a programba. A különbség közöttük, hogy az `asserta` a predikátum első klózaként veszi fel, az `assertz` pedig az utolsóként.

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

A klózokat természetesen nem csak felvenni tudjuk a dinamikus adatbázisba, hanem szükség esetén ki is törölhetjük őket. Erre a célra szolgál a `retract(:@Klóz)` eljárás, amely keres egy olyan dinamikus eljárást, melynek van a `Klóz`-zal egyesíthető klóza. Az első megfelelő klózzal illeszti majd kitörli a klózt.

```
| ?- retract((p(2,_):-_)), listing(p),
     write(----------), nl, fail.
p(1, A) :-
        q(A).
p(2, A) :-
        r(A).
----------
p(1, A) :-
        q(A).
----------
no
```

Fontos megjegyezni, hogy az eljárás *többszörösen sikerülhet*, ugyanis visszalépéskor az eljárás újabb klózt keres, illeszti, majd kitörli azt is. A fenti példában is látszik, hogy a visszalépések során az első és harmadik klózt is kitörölte a `retract` hívás. A `retract` többszörös sikerességét felhasználva készítettük el a következő eljárást:

```
% retractall(Fej): kitörli az összes klózt,
% amelynek feje illeszthető Fej-jel.
retractall(Head) :- retract((Head :- _)), fail.
retractall(_).
```

A `retractall(:@Fej)` hívás kitörli az összes klózt amelynek feje illeszthető `Fej`-jel. Mindig sikerül (akkor is ha nincs kitörlendő klóz). Ez az eljárás beépítve megtalálható a SICStus Prologban.

Az adatbázisban klózok felvétele és törlése mellett le is kérdezhetjük azt, hogy milyen klózok szerepelnek. A `clause(:+Fej, ?Törzs)` eljárás segítségével megvizsgálhatjuk, hogy létezik-e egy interpretált (F:-T) klóz, amely egyesíthető a (`Fej`:-`Törzs`) struktúrával. Ha létezik, akkor a `(Fej :- Törzs)` klózzal illeszthető első klózt megkeresi és illeszti. Ez az eljárás is *többszörösen sikerülhet*, hasonlóan a `retract`-hoz.

4.11.2. Alkalmazási példák dinamikus predikátumokra

P21 Példa: Legnagyobb megoldás keresése

Egy `p(N)` eljárás legnagyobb megoldásának keresése (feltesszük, hogy az `N` megoldás mindig pozitív szám):

[1] Emlékeztetjük az olvasót, hogy a `@` jelölés arra utal, hogy az adott argumentum tisztán bemenő, azaz a benne szereplő változók nem kaphatnak értéket; részletesebben lásd a 3.6.3. és 5.1. pontokban . A `:` jel pedig azt jelzi, hogy az argumentum modulkvalifikált, azaz pl. egy távoli modulba is felvehetünk egy klózt, ha az `assert` paraméterében egy modulnevet szerepeltetünk, a klóztól egy `:` operátorral elválasztva.

```
max_p(_):-
      retractall(legnagyobb(_)),          % egy előző, megszakadt futásból maradhatott
      asserta(legnagyobb(0)),
      p(N),
      legnagyobb(Max),
      (  Max >= N -> fail
      ;  retract(legnagyobb(_)), asserta(legnagyobb(N)), fail
      ).
max_p(N):-
      retract(legnagyobb(N)).
```

P22 Példa: Egyszerű `findall` megvalósítása

Az alábbi `findall1` egyszerűbb mint a beépített, mert skatulyázva nem működik.

```
:- dynamic(solution/1).

:- meta_predicate findall1(?,:,?).

findall1(Templ, Goal, _Sols) :-
        call(Goal),
        asserta(solution(Templ)),  fail.
findall1(_Templ, _Goal, Sols) :-
       collect_sols([], Sols).

% A solution tényállítások argumentumában tárolt
% kifejezések megfordított listája L-L0.
collect_sols(L0, L) :-
        retract(solution(S)), !,
        collect_sols([S|L0], L).
collect_sols(L, L).

| ?- findall1(Y, (member(X, [1,2,3]), Y is X*X), S).
S = [1,4,9] ? ;
no
```

P23 Példa: Egy egyszerű nyomkövető interpreter

```
interp(true, _) :- !.
interp((G1, G2), D) :- !,
        interp(G1, D), interp(G2, D).
interp(G, D) :-
        (   trace(G, D, call)
        ;   trace(G, D, fail), fail
        ),
        D2 is D+2,
        clause(G, B), interp(B, D2),
        (   trace(G, D, exit)
        ;   trace(G, D, redo), fail
        ).

trace(G, D, Port) :-
        tab(D), % D szóközt ír ki
        write(Port), write(': '), write(G), nl.
```

Az interpreter működése - példafutás

```
:- dynamic app/3, app/4.

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

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

| ?- interp(app(_, [b,c], L, [c,b,c,b]), 0).
call: app(_117,[b,c],_167,[c,b,c,b])
  call: app(_117,_572,[c,b,c,b])
  exit: app([],[c,b,c,b],[c,b,c,b])
  call: app([b,c],_167,[c,b,c,b])
  fail: app([b,c],_167,[c,b,c,b])
  redo: app([],[c,b,c,b],[c,b,c,b])
    call: app(_779,_572,[b,c,b])
    exit: app([],[b,c,b],[b,c,b])
  exit: app([c],[b,c,b],[c,b,c,b])
  call: app([b,c],_167,[b,c,b])
    call: app([c],_167,[c,b])
      call: app([],_167,[b])
      exit: app([],[b],[b])
    exit: app([c],[b],[c,b])
  exit: app([b,c],[b],[b,c,b])
exit: app([c],[b,c],[b],[c,b,c,b])

L = [b] ?
```

4.12. Nyelvtani elemzés Prologban — Definite Clause Grammars

A Prolog visszalépéses keresési módszere nagyszerűen alkalmazható nyelvtanok elemzésére is.

4.12.1. Példasor: számok elemzése

Tekintsük a bináris számokat leíró alábbi egyszerű nyelvtant!

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

Tételezzük fel, hogy beolvastunk egy karakter-sorozatot és a kapott karakter-kód listáról szeretnénk eldönteni, hogy az megfelel-e pl. a fenti szabályok által leírt szintaxisnak. Ehhez célszerű a nyelvtan minden egyes nem-terminális jelének (pl. ⟨szám⟩, ⟨számmaradék⟩, ...) egy kétargumentumú Prolog eljárást megfeleltetnünk (az egyszerűség kedvéért ugyanazt a nevet adva neki mint a nem-terminálisnak). Az eljárás első argumentuma egy karakter-kód lista lesz, és azt várjuk tőle, hogy sikerüljön, ha ennek a listának egy kezdőszelete kielemezhető az adott nem-terminálisnak megfelelő szabályok szerint; egyébként hiúsuljon meg. Siker esetén elvárjuk, hogy a második argumentumban adja vissza a bemenő listának a kielemzett kódsorozat elhagyásával keletkező zárószeletét. A két lista kapcsolatát röviden úgy fogjuk leírni, hogy a bemenő listáról *leelemezhető* az adott nem-terminális és marad a kimenő lista.

Tekintsük a ⟨szám⟩ nem-terminálist, és vizsgáljuk meg, milyen Prolog eljárás feleltethető meg neki a fenti elvek szerint:

```
% L0 kódlistáról leelemezhető egy <szám>, marad L.
szám(L0, L) :-
        jegy(L0, L1), számmaradék(L1, L).
```

A Prolog elemző követi a nyelvtani szabály szerkezetét: az eljárástörzsben sorra hívjuk a szabály jobboldalán szereplő nem-terminálisoknak megfeleltetett eljárásokat, a lista-argumentumokat akkumulátor-szerűen összekapcsolva. Ez nem meglepő, hiszen akkumulátorokat állapotváltozások leírására tudunk használni, és itt is van egy állapot-fogalmunk: a bemenő karakterfolyamból még le nem elemzett karakterek listája.

A `számmaradék` nem-terminálisnak a jobboldalán két alternatíva áll, ebből két Prolog szabályt készítünk. A második, üres alternatívából egy olyan ága keletkezik az elemzőnek, amely változtatás nélkül visszaadja a bemenő listát:

```
% Leelemezhető jegyek egy esetleg üres listája.
számmaradék(L0, L) :-
        jegy(L0, L1), számmaradék(L1, L).
számmaradék(L, L).
```

Miután az akkumulátor-párok láncolása egy mechanikus feladat, ez rábízható a Prolog fordítóprogramra. A Prolog rendszerek többsége ugyanis megengedi, hogy programelemként ún. definit klóz nyelvtani (Definite Clause Grammar, DCG) szabályokat szerepeltessünk. Ezeket a Prolog rendszer a program beolvasása során Prolog szabályokká alakítja.

A fenti két Prolog eljárásnak megfelelő DCG szabály:

```
% Leelemezhető egy szám
szám -->
        jegy, számmaradék.

% Leelemezhető jegyek egy esetleg üres listája.
számmaradék -->
        jegy, számmaradék.
számmaradék --> [].
```

A szabály bal- és jobboldalát tehát a `-->` operátorral kell elválasztani, az egymás után írást a `','`, az üres jelsorozatot a `[]` jelöli. Fontos megjegyezni, hogy DCG „tényállítások” nincsenek, tehát az üres törzsű nyelvtani szabályokat úgy kell a DCG jelölésre átírni, hogy a törzsbe `[]`-t írunk (lásd a `számmaradék` második klózát).

Vegyük észre, hogy a `számmaradék` eljárás nem-determinisztikus: először megpróbálja a lehető leghosszabb jegy-sorozatot leelemezni, de visszalépés esetén hajlandó ennek minden kezdőszeletét felsorolni, hiszen ezek mind megfelelnek a ⟨számmaradék⟩ szintaxisának. Ez utóbbi megoldások általában nem vezetnek eredményre, mert „normális” nyelvtan egy ⟨szám⟩ után nem enged meg ⟨jegy⟩-gyel kezdődő nem-terminálist (hiszen akkor sokértelmű lenne az elemzés). Hogy elkerüljük a felesleges visszalépéseket, célszerű a `számmaradék` első DCG szabályában egy vágót elhelyezni, amely átkerül a szabályból generált Prolog kódba is. Ezzel biztosítjuk, hogy ⟨számmaradék⟩-nak csak a maximális hosszúságú jegy-sorozatot tekintjük:

```
% Leelemezhető jegyek egy maximális esetleg üres listája.
számmaradék -->
        jegy, !, számmaradék.
számmaradék --> [].
```

A fenti DCG szabályokból a beolvasás során az előzőleg leírt Prolog eljárások keletkeznek:

```
| ?- listing.
szám(A, B) :-
        jegy(A, C),
        számmaradék(C, B).

számmaradék(A, B) :-
        jegy(A, C), !,
        számmaradék(C, B).
számmaradék(A, B) :-
        B=A.
yes
```

A DCG szabályokban megengedett a diszjunktív jelölés is, a ; használatával. Például a `számmaradék` DCG szabály így is írható (az első a vágó nélküli, a másik vágót tartalmazó változattal azonos hatású):

```
% Leelemezhető jegyek egy esetleg üres listája.
számmaradék -->
        (   jegy, számmaradék.
        ;   []
        ).

% Leelemezhető jegyek egy maximális, esetleg üres listája.
számmaradék -->
        (   jegy -> számmaradék.
        ;   []
        ).
```

Nézzük most, hogy a terminális jelet is tartalmazó ⟨jegy⟩ nyelvtani szabályt hogyan alakíthatjuk át Prolog eljárássá!

```
% jegy(L0, L): L0-ból leelemezhető egy jegy kódja, marad L.
jegy([0'0|L], L).
jegy([0'1|L], L).
```

A terminálisok elemzése nagyon egyszerű, hiszen csak ellenőrizni kell, hogy a listában az adott elem jön, és ha igen, akkor a lista farkát kell kiadni maradékként.

A DCG nyelvtanok a terminálisokra is tartalmaznak jelölést: ezeket egy vagy többelemű listaként kell a DCG szabály jobboldalán szerepeltetni:

```
% Leelemezhető egy jegy kódja.
jegy --> [0'0].
jegy --> [0'1].
```

Ezekből a DCG szabályokból egy kicsit más kód generálódik, mint amit az előbb felírtunk:

```
jegy(A, B) :-
        'C'(A, 48, B).
jegy(A, B) :-
        'C'(A, 49, B).
```

Itt `'C'/3` egy beépített eljárás (azért van ilyen furcsa neve, mert közvetlen alkalmazása a felhasználó számára nem ajánlott), amelynek definíciója:

```
'C'([C|L], C, L).
```

Fejlesszük most tovább elemzőnket, hogy `jegy`-ként decimális számjegyeket is elfogadjon. Ezt a következőképpen tehetjük meg:

```
% leelemezhető egy jegy kódja.
jegy --> [J], {jegykód(J)}.

% J egy jegy kódja.
jegykód(J):-  J >= 0'0, J =< 0'9.
```

Az `jegy` új változatából a következő Prolog klóz keletkezik:

```
jegy(L0, L) :- 'C'(L0, J, L), jegykód(J).
```

Látjuk tehát, hogy a DCG szabályban, a terminálist jelölő listában egy változót is irhatunk, és ezzel mintegy lekérdezzük a soron következő terminálist (karakter-kódot). Egy tetszőleges Prolog hívást is beszúrhatunk az elemzési folyamatba, ha azt a szabályban kapcsos zárójelben szerepeltetjük. Ez a két lehetőség így együtt lehetővé teszi, hogy egy Prolog szabályban döntsük el, hogy elfogadjuk-e a következő terminális jelet.

A DCG szabályokat argumentumokkal is elláthatjuk, így a fenti számelemzőt kiegészíthetjük úgy, hogy siker esetén adja vissza a szám értékét. A `számmaradék` eljárás az általa leelemzett karakterkód-listát adja vissza a paraméterében.

```
% leelemezhető az Sz szám
szám(Sz) -->
        jegy(J), számmaradék(M),
        {number_codes(Sz, [J|M])}.

% leelemezhető jegyek egy esetleg üres JL listája.
számmaradék([J|JL]) -->
        jegy(J), !,
        számmaradék(JL).
számmaradék([]) --> [].

% J a leelemzett jegy kódja.
jegy(J) --> [J], {jegykód(J)}.
```

Ezek a paraméteres nem-terminálisok is éppúgy kiegészülnek egy akkumulátor argumentum-párral, mint a paraméter nélküliek. A DCG $\longrightarrow$ Prolog transzformáció mindig az argumentumlista végére teszi a kiegészítő argumentumokat. Például a fenti első szabály Prolog alakja a következő lesz:

```
szám(Sz, L0, L) :-
        jegy(J, L0, L1),
        számmaradék(M, L1, L),
        number_codes(Sz, [J|M]).
```

4.12.2. A DCG nyelvtani szabályok szerkezete — összefoglalás

- Nem-terminális: tetszőleges *hívható* kifejezés (atom vagy struktúra).
- Terminális: *tetszőleges* Prolog kifejezés; a 0, 1 vagy több terminális sorozata listaként helyezhető el a DCG szabályokban.
- Feltételek: tetszőleges Prolog hívás elhelyezhető `{}` zárójelekbe zárva.
- A DCG szabály alakja: Baloldal `-->` Jobboldal .
- Baloldal: egy nem-terminális, amit opcionálisan terminálisok listája követhet.

- Jobboldal: Egymás után írás (`,`), diszjunkció (`;`), ha-akkor és ha-akkor-egyébként (`->`) és negáció (`\+`) segítségével épül fel terminálisokból, nem-terminálisokból és Prolog feltételekből. A vágó (`!`) „automatikusan" Prolog hívássá alakul (mintha `{!}` volna), de pl. `true` nem.

Általános példa

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

lefordított alakja:

```
p(A,...,L0,L):-
      q0(B,...,L0,L1), ...Li-1 = [X|Li], qi(C,...,Li,Li+1),...
      Cel, ..., qn(D,...,Ln,L)
```

4.12.3. További példák

P24 Példa: Szótárbeolvasás

Ez a példa egy szótárprogram beolvasó-elemző részét mutatja be.

A `szotar_elem_be/1` eljárás beolvas és kielemez egy sort, amelyben szavak sorozata kell álljon. A szavakban csak kis- és nagybetűk fordulhatnak elő, elválasztásukra egy vagy több szóköz szolgál. A sorban legfeljebb egy egyenlőség- vagy minusz-jel is előfordulhat. Egy később megirandó szótár-programban (P32) ez utóbbi alak szolgál majd az egymásnak megfeleltetett kifejezések (szó-sorozatok) megadására, míg az olyan sorok, amelyekben nincs = vagy - jel, a megadott szósorozat lefordítását kérik majd.

A `szotar_elem_be(Kif)` eljáráshívás a következő sor belső alakját adja vissza `Kif`-ben, ez egy atomokból álló lista, vagy két ilyen lista a - operátorral összekötve. Ha az elemzés nem sikerül, az eljárás hibát jelez, és mindaddig új sort olvas, amíg egy helyes alakot nem talál.

A soron következő karakter beolvasására a `get_code(Kód)` eljárás szolgál, amely a karakter kódját egyesíti a `Kód` argumentummal.

```
% Kif a következő sor kielemzett formája
szotar_elem_be(Kif):-
        rd_line(L),
        (   szotar_elem(Kif, L, []) -> true
        ;   format('Hibas sor: ~s~n', [L]), szotar_elem_be(Kif)
        ).

% szotar_elem(Kif, L1, L): az L1 kódlistáról
% kielemezhető egy szótar-elem, a maradék kódlista L.
szotar_elem(Elem) -->
        szokoz, szosor(Szosor1),
        (   elvalaszto ->  szosor(Szosor2), {Elem = Szosor1-Szosor2}
        ;   {Elem = Szosor1}
        ).

% elvalaszto --> leelemezhető egy elvalasztó.
elvalaszto--> [0'-], szokoz ; [0'=], szokoz.

% szosor(SL) --> leelemezhető az SL szólista
szosor([S|SL]) -->
        szo(S), szokoz,
        (   szosor(SL)
        ;   {SL = []}
        ).
```

```
% leelemezhető az S szó
szo(S) --> betu(B), szomaradek(BL), {atom_codes(S, [B|BL])}.

% szomaradek(BL) --> leelemezhető betűk egy esetleg
% üres listája, BL a leelemzett betűk kódlistája.
szomaradek([B|BL]) -->
        betu(B), !, szomaradek(BL).
szomaradek([]) --> [].

% B a leelemzett betű kódja.
betu(B) --> [B], {betukod(B)}.

% B egy betű kódja.
betukod(B):- B >= 0'a, B =< 0'z ; B >= 0'A, B =< 0'Z.

% leelemezhető szóközök esetleg üres sorozata
szokoz --> [0' ], !, szokoz ; [].

% L a következő sor karakterkódjainak listája.
rd_line(L):-
        get_code(C), rd_line(C, L).

% rd_line(C, L): L a következő sor karakterkódjainak
% listája, ha C a soronkövetkező karakter kódja.
rd_line(10, []):- !.
rd_line(C, [C|L]):-
        get_code(C1), rd_line(C1, L).

| ?- szotar_elem_be(K).
|: six = hat.
Hibas sor: six = hat.
|: six = hat

K = [six]-[hat] ? ;

no
| ?- szotar_elem_be(K).
|: Prolog clause = Prolog kloz

K = ['Prolog',clause]-['Prolog',kloz] ? ;

no
```

P25 Példa: Kifejezés kiértékelése

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

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

```
tag(Z) --> szám(Z).

| ?- kif(Z, "10*10-6*6", "").
Z = 64 ? ;
no
| ?- kif(Z, "10*10-6*6", L).
L = [], Z = 64 ? ;
L = [42,54], Z = 94 ? ;
L = [45,54,42,54], Z = 100 ? ;
L = [42,49,48,45,54,42,54], Z = 10 ? ;
no
```

Vegyük észre, hogy a fenti program „mohón” elemez, tehát először a leghosszabb kielemezhető kifejezést értékét adja vissza, de visszalépéskor hajlandó ennek részkifejezésként értelmezhető kezdőszeleteit is elfogadni. Ha a kifejezés után egy lezáró karaktert is elvárunk, akkor ezzel kizárhatjuk a részkifejezések elfogadását.

Vegyük észre, hogy a program az azonos prioritású műveleteket jobbról balra zárójelezi:

```
| ?- kif(Z, "4-2+1", []).
Z = 1 ?
```

Egy lehetséges javítás a kifejezés-maradék, tag-maradék stb. nyelvtani fogalmak bevezetése. Az alábbi részlet a kifejezés-elemzést mutatja be, a tagok elemzését végző szabályok analóg módon írhatók meg.

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

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

P26 Példa: “természetes” nyelvű beszélgetés

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

% mondat(Alany, Áll, L0, L): L0-L kielemezhető egy Alany alanyból
% és Áll állítmányból álló mondattá.
% Alany lehet első vagy második személyű névmás, vagy egyetlen szóból
% álló (harmadik személyű) alany.
mondat(én, Áll) --> perm(én, Áll).
mondat(te, Áll) --> perm(te, Áll).
mondat(Alany, Áll) --> szó(Alany), szavak(Áll).

% perm(Ki, Áll, L0, L): L0-L kielemezhető egy Ki (első vagy második személyű)
% névmásból és Áll állítmányból álló mondattá.
perm(Ki, Áll) --> névmás(Ki), létige(Ki), szavak(Áll).
perm(Ki, Áll) --> névmás(Ki), szavak(Áll), létige(Ki).
perm(Ki, Áll) --> szavak(Áll), létige(Ki), névmás(Ki).
perm(Ki, Áll) --> szavak(Áll), létige(Ki).

% névmás(Ki, L0, L): L0-L egy Ki névmás.
névmás(én) --> "én", köz.        névmás(te) --> "te", köz.
névmás(én) --> "Én", köz.        névmás(te) --> "Te", köz.
```

```
% létige(Ki, L0, L): L0-L egy Ki névmással egyeztetett létige.
létige(én) --> "vagyok", köz.    létige(te) --> "vagy", köz.

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

% szó(Sz, L0, L): L0-L egy Sz betűsorozatból álló (nem üres) szó.
% Betűnek számít minden a ".?" jelektől különböző karakter.
szó([B|Sz]) --> betű(B), szómaradék(Sz), köz.

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

% betű(K, L0, L): L0-L egy K kódú betű.
betű(K) --> [K], {non_member(K, " .?")}.

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

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

% kérdő(Szó): Szó egy kérdőszó.
kérdő("mi").   kérdő("ki").   kérdő("kicsoda").
kérdő("Mi").   kérdő("Ki").   kérdő("Kicsoda").

% Egy párbeszédet valósít meg.
párbeszéd :-
     repeat, rd_line(L),
     (   menet(M, L, []) -> feldolgoz(M)
     ;   write('Nem értem\n'), fail
     ), M = un, !.

% feldolgoz(Mondás): feldolgozza a felhasználótól érkező Mondás üzenetet.
feldolgoz(un) :- write('Én is.\n').
feldolgoz(kijelent(Alany, Áll)) :-
     assertz(tudom(Alany,Áll)), write('Felfogtam.\n').
feldolgoz(kérdez(Alany)) :-
     tudom(Alany, _), !, válasz(Alany).
feldolgoz(kérdez(_)) :-
     write('Nem tudom.\n').

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

```
    ).
válasz(_).
```

A fenti programban hivatkozott `rd_line/1` definícióját lásd a P24 példában.

Beszélgetős DCG példa — egy párbeszéd

```
| ?- párbeszéd.
|: Magyar legény vagyok én.
Felfogtam.
|: Ki vagyok én?
Magyar legény
|: Péter kicsoda?
Nem tudom.
|: Péter tanuló.
Felfogtam.
|: Péter jó tanuló.
Felfogtam.
|: Péter kicsoda?
tanuló
jó tanuló
|: Boldog vagyok.
Felfogtam.
|: Te egy Prolog program vagy.
Felfogtam.
|: Ki vagyok én?
Magyar legény
Boldog
|: Okos vagy.
Felfogtam.
|: Te vagy a világ közepe.
Felfogtam.
|: Ki vagy  te?
egy Prolog program
Okos
a világ közepe
|: Valóban?
Nem értem.
|: Unlak.
Én is.
```

4.12.4. DCG használata elemzésen kívül

A DCG formalizmust kényelmesen használhatjuk akkumulálást végző eljárások tömörebb írására. Listák akkumulálása esetén az elemi akkumulálási lépést a `[X]` DCG terminális jelölésként írhatjuk le.

Például a klasszikus `append` eljárást a következőképpen definiálhatjuk DCG szabályok segítségével:

```
append(L1, L2, L12) :-
        app(L1, L12, L2).

app([]) --> [].
app([X|L]) --> [X], app(L).

| ?- listing(app).
app([], A, B) :-
        B=A.
app([A|B], C, D) :-
        'C'(C, A, E),
        app(B, E, D).

| ?- append([alma,korte], [szilva], L).
L = [alma,korte,szilva] ? ;
no
```

Az alábbi példa egy listaszűrő eljárást DCG megvalósítását mutatja be.

```
% M az L lista N-nél nagyobb elemeinek listája.
nagyobb(L, N, M) :- nagyobb(L, N, M, []).

% nagyobb(L, N, M, M0): M-M0 az L lista N-nél
% nagyobb elemeinek listája.
```

```
nagyobb([], _) --> [].
nagyobb([X|L], N) --> {X > N}, !, [X], nagyobb(L, N).
nagyobb([_|L], N) --> nagyobb(L, N).
```

A DCG szabályokat használhatjuk nem csak listák, hanem tetszőleges kifejezések akkumulálására is, de ez esetben az elemi akkumulálási lépést a DCG-n kívül kell megírni:

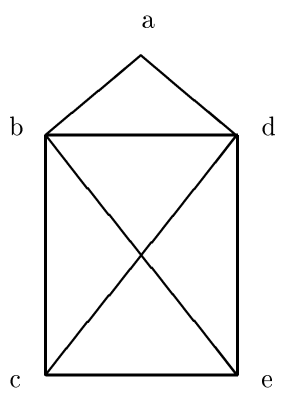
```
% Az L lista összege S.
sum(L, S) :- sum1(L, 0, S).

% sum1(L, S0, S): Az L lista összege S-S0.
sum1([]) --> [].
sum1([X|L]) --> add(X), sum1(L).

% X+S0= S.
add(X, S0, S) :- S is S0+X.
```

4.13. Nagyobb példák

Ebben az alfejezetben két közepes méretű Prolog példát ismertetünk: egy közismert feladványtípust megoldó programot, és egy rendezett bináris fákat kezelő eljárásgyűjteményt.

P27 Példa: Ábrarajzoló

Írjunk Prolog programot amely egy összefüggő vonallal megrajzolja pl. az alábbi ábrát:



Először el kell határoznunk milyen formában ábrázoljuk a program bemenetét, a gráfot és a várt eredményt, a vonalat:

megrajzolandó gráf:	élek listája
él:	egyik végpont - másik végpont a fenti gráf leírása pl. lehet: `[a-b,a-d,b-c,b-d,b-e,c-d,c-e,d-e]`
összefüggő vonal:	olyan gráfleírás (éllista), ahol a minden él végpontja a következő él kezdőpontjával azonos. pl.: `[c-b,b-a,a-d,d-b,b-e,e-c,c-d,d-e]`

Döntésünknek a következő típus-specifikációk felelnek meg:

```
% :- type  graf == list(el).
% :- type    el ---> atom-atom.
% :- type vonal == graf.
```

A függelék

Fogalom-tár

argumentum (argument)

Lásd struktúra.

argumentumszám, aritás (arity)

Egy eljárás vagy egy struktúra argumentumainak a száma.

atom (atom)

Egy névkonstans. A kisbetűvel kezdődő és alfanumerikus karakterekkel folytatódó sorozatot, a csupa „speciális” karakterből álló sorozatot és az aposztrófok közé zárt tetszőleges sorozatot atomnak tekinti a rendszer. Az atomok egyesítése gyors, ábrázolásuk tömör, viszont drága a létrehozásuk és szétszedésük.

atomi kifejezés (atomic term)

Lásd konstans.

behelyettesítés (binding)

Amikor egy változót egyesítünk valamivel (akár egy másik változóval is), akkor azt mondjuk, hogy a változót behelyettesítettük.

beépített eljárás (built-in predicate)

Egy rendszer által definiált reláció.

cél (goal)

Egy eljáráshívás. Egy cél vagy sikerül, vagy meghiúsul. Egy cél többféleképpen is sikerülhet.

deklaráció (declaration)

A fordítónak szóló állítások, amelyek a program értelmezését befolyásolják. Példa: operátor-deklaráció.

determinisztikus (determinate)

Egy hívás determinisztikus, ha legfeljebb egyféleképpen sikerülhet. Egy eljárás determinisztikus, ha tetszőleges hívása determinisztikus.

dinamikus eljárás (dynamic predicate)

Olyan eljárás, amelyhez futási időben adhatunk és amelyből futási időben vehetünk el klózokat.

diszjunkció (disjunction)

Vagy kapcsolat. Prologban a vagy kapcsolatban levő relációkat pontosvesszővel elválasztva egymás után írjuk, és általában zárójelekkel vesszük körbe.

egyesítés (unification)

Kifejezések mintaillesztéssel történő azonos alakra hozása.

egyhosszúságú atom (one-char atom)

Egyetlen jelből álló atom. Ezeket nevezzük karakternek.

egyszerű kifejezés (simple term)

Egy nem struktúra kifejezés, azaz változó vagy konstans.

eljárás-doboz (procedure box)

A Prolog programok végrehajtásának leírására használt (fogalmi) segédeszköz.

eljárás (procedure)

Egy beépített vagy egy felhasználó által definiált eljárás.

előfordulás-ellenőrzés (occurs-check)

Egy változó és egy struktúra egyesítésekor annak ellenőrzése, hogy a változó nem szerepel-e a struktúrában.

exportál (export)

Egy modul azokat az eljárásokat exportálja, amelyeket a modulon kívülről is elérhetővé akar tenni.

értékadás (instantiation)

Amikor egy változót egy nem változóval egyesítünk, akkor az értéket kap. (Az értékadás lehet részleges is, amennyiben a kifejezés, amivel egyesítettünk nem tömör.)

értékkel bíró (instantiated)

Egy változó értékkel bír, ha egyesítettük egy nem változó kifejezéssel.

értékkel nem bíró (unbound)

Egy változó, amit még soha nem egyesítettünk változótól különböző kifejezéssel.

fej (head)

Egy klóz következmény része.

felhasználó által definiált eljárás (user defined procedure)

Mindazon klózok összessége, amelyek feje adott nevű és argumentumszámú.

fordítás (compile)

Az a folyamat, amikor a program szövegéből lefordított alakot állítunk elő. A lefordított alak gyorsabban fut, mint az interpretált alak.

funktor (functor)

Egy struktúra nevéből és argumentumszámából képzett pár. Írásban általában / jellel elválasztva írjuk. Például a `rendező('Luis Buñuel', 1900, 1983)` struktúra funktora `rendező`/3.

füzér (string)

Karakterkódok listája.

importál (import)

Ha fel akarunk használni egy (másik) modulban definiált eljárást, akkor a használat előtt azt importálni kell.

indexelés (indexing)

Arra szolgál, hogy egy adott hívás esetén a rendszer ne próbálkozzon olyan klózokkal, amelyek semmiképp nem sikerülhetnek. Az indexelést a rendszer automatikusan végzi.

kampó eljárás (hook predicate)

Olyan eljárás, amit a felhasználó definiál, de közvetlenül a rendszer hívja meg.

karakterkód (character code)

Egy karakter numerikus kódja.

kifejezés (term)

A Prologban előforduló adatfajták gyűjtőneve.

klóz (clause)

Egy tényállítás vagy egy szabály.

konjunkció (conjunction)

És kapcsolat. Prologban az és kapcsolatban levő relációkat vesszővel elválasztva egymás után írjuk.

konstans (constant)

Egy egész vagy egy lebegőpontos szám, vagy egy atom. Az `atomic`/1 beépített eljárás pontosan ezekre igaz.

konzultálás (consult)

Az a folyamat, amikor a program szövegéből interpretált alakot állítunk elő. Az interpretált alak lassabban fut, mint a lefordított alak, de kilistázható (`listing`) és egy kicsit részletesebben nyomkövethető.

kérdés, célsorozat (query)

Célok konjunkciója. Ha egy célsorozatot közvetlenül az interpreternek adjuk oda, kérdésnek is nevezzük.

lista (list)

Vagy a `'[]'` (nil) atom, vagy egy `'.'`/2 struktúra, melynek második argumentuma egy lista. Ha egy lista vége nem nil, hanem változó, akkor nyitott végű listáról beszélünk.

meghívható kifejezés (callable term)

Egy atom vagy egy struktúra. A `callable`/1 eljárás pontosan ezekre igaz.

mellékhatás (side-effect)

Logikailag nem értelmezhető hatás.

meta eljárás (meta-predicate)

Olyan eljárás, amely meghívja valamelyik argumentumát.

meta hívás (meta-call)

Egy Prolog kifejezés hívásként való értelmezése. Lásd a `call/1` eljárást!

modul (module)

Eljárások olyan (legbővebb) halmaza, amelyek azonos modul-deklaráció hatókörébe esnek.

névtelen változó (anonymous variable)

A névtelen változó egy minden más változótól különböző változó. Jelölése: `_`.

operátor (operator)

Olyan név, amelyet (operátor-deklaráció) segítségével prefix, infix vagy postfix alakban használhatóvá tettünk.

összetett kifejezés (compound term)

Lásd struktúra.

parancs (directive)

Fordítási időben végrehajtandó Prolog hívás.

precedencia (precedence)

Az a szám, amely megmondja egy operátorról, hogy mennyire szorosan köt.

predikátum (predicate)

Lásd eljárás.

program (database)

A felhasználó által definiált eljárások halmaza.

pár (pair)

A `'-'/2` struktúra. Általánosan bármely kétargumentumú struktúrát nevezhetünk párnak, de ilyenkor meg kell mondani mi a neve.

redukciós lépés (reduction step)

A végrehajtásnak az a lépése, amikor egy hívást egy illeszkedő fejű klóz törzsével helyettesítünk.

rekurzió (recursion)

Egy eljárás rekurzív, ha végrehajtásához szükség van önmaga (esetleg közvetett) meghívására.

statikus eljárás (static predicate)

Egy nem dinamikus eljárás.

struktúra (structure)

Egy Prolog kifejezés. A strúktúráknak van egy neve és egy vagy több argumentuma. Például a `személy('Weöres Sándor', 1913, 1989)` struktúra neve `személy` és három argumentuma van. A `compund/1` eljárás pontosan ezekre igaz.

szabály (rule)

Egy klóz egy vagy több feltétellel. Egy szabály azt fejezi ki, hogy a törzséből következik a feje. Például a `jókedvű(attila) :- kertész(attila).` szabály azt fejezi ki, hogy ha Attila kertész, akkor jókedvű.

Változókat használva általánosabb szabályokat is felírhatunk:
`jóhírű(X) :- tejet_iszik(X), pipázik(X).`
Ezzel azt állítjuk, hogy minden pipázó és tejivó X-re igaz, hogy jó a híre.

szemantika (semantics)

A Prolog kifejezések „értelme”.

szemétgyűjtés (garbage collection)

Az elérhetetlenné vált adatok által foglalt memóriaterületek újra felhasználhatóvá tétele.

szintaxis (syntax)

A nyelvtan azon része, amely azt írja le, hogyan állnak elő szimbólumokból érvényes Prolog kifejezések.

szülő (parent)

Az a hivás, amelynek redukciója során a kérdéses cél belekerült a célsorozatba. Más szóval az a hívás, amely a kérdéses célt tartalmazó klóz fejével illesztődött.

tényállítás (fact)

Egy klóz, amelynek nincs feltétel része, azaz a törzse üres. Egy tényállítás azt fejezi ki, hogy az adott reláció fennáll az argumentumaira. Példák tényállításokra (és lehetséges interpretációjukra):

```
király(henrik, anglia).   % Henrik Anglia királya volt.
csőre_van(madár).         % A madaraknak van csőrük.
alkalmazott(lilla, adatfeldolgozás, 55000).
                          % Lilla az adatfeldolgozási osztályon dolgozik,
                          % keresete 55000.
```

tömör (ground)

Olyan Prolog kifejezés, amelyben nem szerepel értékkel nem bíró változó.

törzs (body)

Egy szabály feltétel része.

visszalépés (backtracking)

Ha egy egyszer már kielégített célt más módon újra megpróbálunk kielégíteni, akkor visszalépésről beszélünk.

változó (variable)

Egy név ami egy (esetleg ismeretlen) értéket jelöl a programban. Prologban a változókat nagybetűvel vagy aláhúzással kezdődő és alfanumerikus karaktersorozattal folytatódó szöveggel jelöljük.