Deklaratív programozás 11. előadás

Kabódi László[1] Marussy Kristóf[2]
BME Számítástudományi és Információelméleti Tanszék
Mesterséges Intelligencia és Rendszertervezés Tanszék

[1] kabodi.laszlo@vik.bme.hu
[2] marussy@mit.bme.hu

2026. ősz

I. rész

Programozás Prologban

1 Programozás Prologban

Tartalom

1. Programozás Prologban
   - Vezérlési eljárások
   - DCG – a Definite Clause Grammar kiterjesztés
   - Egy összetettebb példaprogram
   - Összefoglaló példa
   - A példa Prolog változata

Vezérlési eljárások, a `call/1` beépített eljárás

- Vezérlési eljárás: A Prolog végrehajtáshoz kapcsolódó beépített eljárás.
- A vezérlési eljárások többsége **magasabbrendű** eljárás, azaz olyan eljárás, amely egy vagy több argumentumát eljáráshívásként értelmezi.
- A meta-eljárások fő képviselője a `call(+Cél)`:
  - `Cél` egy struktúra vagy névkonstans (vö. `callable/1`).
  - Jelentése (deklaratív szemantika): `Cél` igaz.
  - Hatása (procedurális szemantika): a `Cél` kifejezést hívássá alakítja és végrehajtja.
- A klóztörzsben célként megengedett egy `X` változó használata, ezt a rendszer egy `call(X)` hívássá alakítja át.

```
| kétszer(X) :- call(X), X.

| ?- kétszer(write(ba)), nl.   ⟹    baba
| ?- listing(kétszer).         ⟹    kétszer(X) :-
                                       call(X), call(X).
```

Vezérlési szerkezetek mint eljárások

- A `call/1` argumentumában szerepelhetnek vezérlési szerkezetek is, mert ezek beépített eljárásként is jelen vannak a Prolog rendszerben:
  - `(',')/2`: konjunkció.
  - `(;)/2`: diszjunkció.
  - `(->)/2`: if-then; `(;)/2`: if-then-else.
  - `(\+)/1`: meghiúsulásos negáció.
- A `call`-ban szereplő vezérlési szerkezetek ugyanúgy futnak, mint az interpretált (azaz `consult`-tal betöltött) kód.
- A `Cél`-beli vágó csak a `call` belsejében vág (szülője a `call(Cél)` hívás).
- Példák:

```
| ?- _Cél = (kétszer(write(ba)), write(' ')), kétszer(_Cél), nl.
baba baba
| ?- kétszer((member(X, [a,b,c,d]), write(X), fail ; nl)).
abcd
abcd
```

`call/1` példa: futási időt mérő meta-eljárás

```
% Kiírja Goal első megoldásának előállításához vagy a meghiúsuláshoz
% szükséges időt, a Txt szöveg kiséretében.
time(Txt, Goal) :-
        statistics(runtime, [T0,_]),  % T0 az indítás óta eltelt CPU idő,
                                      % msec-ban (szemétgyűjtés nélkül).
        (   call(Goal) -> Res = true
        ;   Res = false
        ),
        statistics(runtime, [T1,_]), T is T1-T0,
        format('~w futási idő = ~3d sec\n', [Txt,T]),
                % ~w formázó: kiírás a write/1 segítségével
                % ~3d formázó: I egész kiírása I/1000-ként, 3 tizedesre
        Res = true.                   % meghiúsul, ha Goal meghiúsult
```

További beépített vezérlési eljárások

- `once(Cél)`: `Cél` igaz, és csak az első megoldását kérjük. Definíciója:

  ```
  once(X) :- call(X), !.
  ```

  vagy, feltételes szerkezettel

  ```
  once(X) :- (   call(X) -> true   ).
  ```

- `true`: azonosan igaz, `fail`: azonosan hamis (mindig meghiúsul).
- `repeat`: végtelen sokszor igaz (végtelen választási pont). Definíciója:

  ```
  repeat.
  repeat :- repeat.
  ```

- A `repeat` eljárást egy mellékhatásos eljárás ismétlésére használhatjuk.
- Példa (egyszerű kalkulátor):

  ```
  bc :-  repeat, read(Expr),
              (   Expr = end_of_file -> true
              ;   Res is Expr, write(Expr = Res), nl, fail
              ),
         !.
  ```

- A végtelen választási pontot kötelező egy vágóval semlegesíteni!

Példa: magasabbrendű reláció definiálása (kieg. anyag)

- Az implikáció (`P` $\Rightarrow$ `Q`) megvalósítása negáció segítségével:

```
% P minden megoldása esetén Q igaz.
forall(P, Q) :-
     \+ (P, \+Q). % Szintaktikus emlékeztető:
                  % az első \+ után kötelező a szóköz!

| ?- _L = [1,2,3],
     % _L minden eleme pozitív:
     forall(member(X, _L), X > 0).
true ?
| ?- _L = [1,-2,3], forall(member(X, _L), X > 0).
no
| ?- _L = [1,2,3],
     % _L szigorúan monoton növő:
     forall(append(_,[A,B|_], _L), A < B).
true ?
```

- `forall/2` csak eldöntendő kérdés esetén használható.

Tartalom

1. Programozás Prologban
   - Vezérlési eljárások
   - DCG – a Definite Clause Grammar kiterjesztés
   - Egy összetettebb példaprogram
   - Összefoglaló példa
   - A példa Prolog változata

A DCG (Definite Clause Grammars) formalizmus

- DCG: előfeldolgozó eszköz nyelvtani elemzők írásához.
- DCG szabály: *Fej*`-->`*Törzs*. $\Longrightarrow$ *Fej*(A_0, A_m)`:-`*Törzs*(A_0, A_m). A törzsben:
  - `{Cél}` $\Longrightarrow$ `Cél` (akkumulálást nem végző cél)
  - `[`E_1`,`E_2`,...,`E_k`]`, $k \geq 0$ $\Longrightarrow$ A_n `= [`E_1`,`E_2`,...,`E_k`|`A_{n+1}`]` (elemek akk.-a)
  - `p(`X_1`,`X_2`,...,`X_j`)`, $l \geq 0$ $\Longrightarrow$ `p(`X_1`,`X_2`,...,`X_j`,`A_n`,`A_{n+1}`)` (akk.-t végző cél)
  - Vezérlés: konj. (`,`), diszj. (`;`), ha-akkor (`->`), vágó (`!`), negáció (`\+`)
- Példa: egy lista pozitív elemeinek kigyűjtése

```
% pe(L, Pk0, Pk): Az L számlista pozitív elemeinek listája Pk0-Pk.
% Másszóval: L pozitív elemeinek listáját Pk elé füzve kapjuk Pk0-t
pe([], Pk0, Pk) :-     Pk0 = Pk.
pe([X|L], Pk0, Pk) :-  (   X > 0 -> Pk0 = [X|Pk1], pe(L, Pk1, Pk)
                       ;   pe(L, Pk0, Pk)
                       ).
```

- A DCG jelölést használó, a fentivel azonos kódot eredményező eljárás:

```
pe2([]) -->            [].
pe2([X|L]) -->         (   {X > 0} -> [X], pe2(L)
                       ;   pe2(L)
                       ).
```

A DCG formalizmus használata nyelvtani elemzésre

- Példa – decimális számok elemzését végző `szám(L0, L)` Prolog eljárás
- Az L0, L paraméterek: karakterkódok listái

```
% szám(L0, L): Az L0-L különbséglista számjegykódok nem-üres listája
% Másszóval: L0 elejéről leelemezhető egy szám, és marad L
szám --> számjegy, számmaradék.

% számmaradék(L0, L): Az L0-L különbséglista számjegykódok listája
számmaradék --> számjegy, számmaradék ; "".            % "" ≡ []

% számjegy(L0, L): L0 = [K|L], ahol K egy számjegy kódja
számjegy --> "0";"1";"2";"3";"4";"5";"6";"7";"8";"9".  % "9" ≡ [0'9]
```

- A `számjegy/2` eljárás egy másik megvalósítása:

```
számjegy --> [K], {decimális_jegy_kódja(K)}.                    (*)

% K egy számjegy kódja.
decimális_jegy_kódja(K) :- K >= 0'0, K =< 0'9.
```

- A fenti (*) DCG szabály Prolog megfelelője:

```
számjegy(L0, L) :-
        L0 = [K|L],                 % K a következő listaelem
        decimális_jegy_kódja(K).    % megfelelő-e a K?
```

DCG nyelvtani elemzés – további részletek

- Az elemzés – a Prolog végrehajtás miatt – nem-determinisztikus, pl.

```
| ?- szám("123 abc", L).      % SWI-ben atom_codes/2 is kell
L = " abc" ?  ;               % leelemeztük a 123 számot
L = "3 abc" ?  ;              % leelemeztük a 12 számot
L = "23 abc" ?  ;             % leelemeztük az 1 számot
no
```

- A számmaradék eljárás determinisztikus változata

```
% számmaradék2(L0, L): L0-L számjegykódok maximális listája
számmaradék2 --> (   számjegy -> számmaradék2
                 ;   ""
                 ).
```

vagy

```
számmaradék3 --> számjegy, !, számmaradék3. % A vágó köré nem kell {}
számmaradék3 --> "".
```

- Futás:

```
| ?- szám2("123 abc", L).
L = " abc" ? ;          % leelemeztük a (lehető leghosszabb) 123 számot
no
```

Az elemző kiegészítése jelentéshordozó argumentumokkal

- Egy DCG szabály az elemzéssel párhuzamosan további (kimenő) argumentum(ok)ban felépítheti a kielemzett dolog „jelentését”
- Példa: szám elemzése és értékének kiszámítása:

```
% Leelemezhető egy Sz értékű nem-üres számjegysorozat
szám(Sz) --> számjegy(J), számmaradék(J, Sz).

% Leelemezhető számjegyek egy esetleg üres listája, amelynek
% az eddig leelemzett Sz0-val együtt vett értéke Sz.
számmaradék(Sz0, Sz) -->
    számjegy(J), !, {Sz1 is Sz0*10+J}, számmaradék(Sz1, Sz).
számmaradék(Sz0, Sz0) --> [].

% leelemezhető egy J értékű számjegy.
számjegy(J) --> [K], {decimális_jegy_kódja(K), J is K-0'0}.
```

```
| ?- szám(Sz, "102 56", L). ⟹   L = " 56", Sz = 102; no
```

- A `számmaradék` DCG szabály Prolog alakja:

```
számmaradék(Sz0, Sz, L0,L) :-
    számjegy(J, L0,L1),!, Sz1 is Sz0*10+J, számmaradék(Sz1, Sz, L1,L).
számmaradék(Sz0, Sz0, L0,L) :- L=L0.
```

- Itt két akkumulátorpár van: egy „kézi” (`Sz`) és egy DCG-ből generált (`L`).

Aritmetikai kifejezések elemzése

- Egyszerű aritmetikai kifejezések elemzése és kiértékelése.

```
% kif0(Z, L0, L): L0-L egy Z aritmetikai kifejezéssé elemezhető ki
kif0(X+Y) --> tag0(X), "+", !, kif0(Y).
kif0(X-Y) --> tag0(X), "-", !, kif0(Y).
kif0(X) -->   tag0(X).

tag0(X) --> szám(X).                % egyelőre
```

```
| ?- kif0(Z, "4-2+1", []). ⟹   Z = 4-(2+1)   Jobbról balra elemez!
```

- Egy lehetséges javítás

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

kifmaradék(X, Z) --> "+", tag(Y), !, kifmaradék(X+Y, Z).
kifmaradék(X, Z) --> "-", tag(Y), !, kifmaradék(X-Y, Z).
kifmaradék(X, X) --> [].

tag(Z) --> szám(X), tagmaradék(X, Z).

tagmaradék(X, Z) --> "*", szám(Y), !, tagmaradék(X*Y, Z).
tagmaradék(X, Z) --> "/", szám(Y), !, tagmaradék(X/Y, Z).
tagmaradék(X, X) --> [].
```

```
| ?- kif(Z, "5*4-2+1", []), Val is Z. ⟹   Z = 5*4-2+1, Val = 19 ? ; no
```

Tartalom

1. Programozás Prologban
   - Vezérlési eljárások
   - DCG – a Definite Clause Grammar kiterjesztés
   - Egy összetettebb példaprogram
   - Összefoglaló példa
   - A példa Prolog változata

Egy nagyobb DCG példa: „természetes” nyelvű beszélgetés

```
% mondat(Alany, Áll, LO, L): LO-L kielemezhető egy Alany alanyból és Áll
% állítmányból álló mondattá. Alany lehet első vagy második személyű
% névmás, vagy egyetlen szóból álló (harmadik személyű) alany.
mondat(Alany, Áll) -->
        {én_te(Alany, Ige)}, én_te_perm(Alany, Ige, Áll).
mondat(Alany, Áll) -->
        szó(Alany), szavak(Áll).

% én_te(Alany, Ige):
% Az Alany első/második személyű névmásnak megfelelő létige az Ige.
én_te("én", "vagyok").
én_te("te", "vagy").

% én_te_perm(Ki, Ige, Áll, LO, L): LO-L kielemezhető egy Ki
% névmásból, Ige igealakból és Áll állítmányból álló mondattá.
én_te_perm(Alany, Ige, Áll)  -->
        (   szó(Alany), szó(Ige), szavak(Áll)
        ;   szó(Alany), szavak(Áll), szó(Ige)
        ;   szavak(Áll), szó(Ige), szó(Alany)
        ;   szavak(Áll), szó(Ige)
        ).
```

Példa: „természetes” nyelvű beszélgetés – szavak elemzése

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

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

% illik(Szó0, Szó): Szó0 = Szó, vagy a kezdő kis-nagy betűben különböznek.
illik([B0|L], [B|L]) :-
        (   B = B0 -> true
        ;   abs(B-B0) =:= 32
        ).

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

% betű(K, L0, L): L0-L egy K kódú "betű" (különbözik a " .?" jelektől)
betű(K) --> [K], {\+ member(K, " .?")}.

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

Példa: „természetes” nyelvű beszélgetés – párbeszéd-szervezés

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

% Megvalósít egy párbeszédet.
párbeszéd :-
        repeat,
            read_line(L),   % beolvas egy sort, L a karakterkódok listája
            (   menet(Mondás, L, []) ->  feldolgoz(Mondás)
            ;   write('Nem értem\n'), fail
            ),
        Mondás = un, !.

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

% kérdő(Szó): Szó egy kérdőszó.
kérdő("mi").
kérdő("ki").
kérdő("kicsoda").
```

Példa: „természetes” nyelvű beszélgetés – válaszok előállítása

```
:- dynamic tudom/2.

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

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

Beszélgetős DCG példa – egy párbeszéd

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

Tartalom

1. Programozás Prologban
   - Vezérlési eljárások
   - DCG – a Definite Clause Grammar kiterjesztés
   - Egy összetettebb példaprogram
   - Összefoglaló példa
   - A példa Prolog változata

Feladat: adott értékű kifejezés előállítása

- A feladat: írjunk programot a következő feladvány megoldására:
  - Adott számokból a négy alapművelettel (+, -, *, /) építsünk egy megadott értékű aritmetikai kifejezést!
    (Feltételezhető, hogy az adott számok mind különböznek.)
  - Példa: keressünk egy olyan aritmetikai kifejezést, amely az 1, 3, 4, 6 számokból áll, és értéke 24 (nehéz feladat!)
  - Pontosítás:
    - A számok nem „tapaszthatók” össze hosszabb számokká
    - Mindegyik adott számot pontosan egyszer kell felhasználni, sorrendjük tetszőleges lehet
    - Nem minden alapműveletet kell felhasználni, egyfajta alapművelet többször is előfordulhat
    - Zárójelek tetszőlegesen használhatók
- Példák a fenti szabályoknak megfelelő, az 1, 3, 4, 6 számokból felépített aritmetikai kifejezésekre: $1 + 6 * (3 + 4)$, $(1 + 3)/4 + 6$

Hogyan ábrázoljuk az aritmetikai kifejezéseket?

- Első ötlet: füzér (string)
- Egy könnyebben kezelhető, strukturált ábrázoláshoz írjuk fel az aritmetikai kifejezés – ⟨ akif ⟩ – szintaxisát:

⟨ akif ⟩ ::= ⟨ szám ⟩ |
(⟨ akif ⟩ ⟨ művelet ⟩ ⟨ akif ⟩)
⟨ művelet ⟩ ::= + | - | * | /

(az egyértelműség kedvéért minden részkifejezést zárójelbe teszünk)

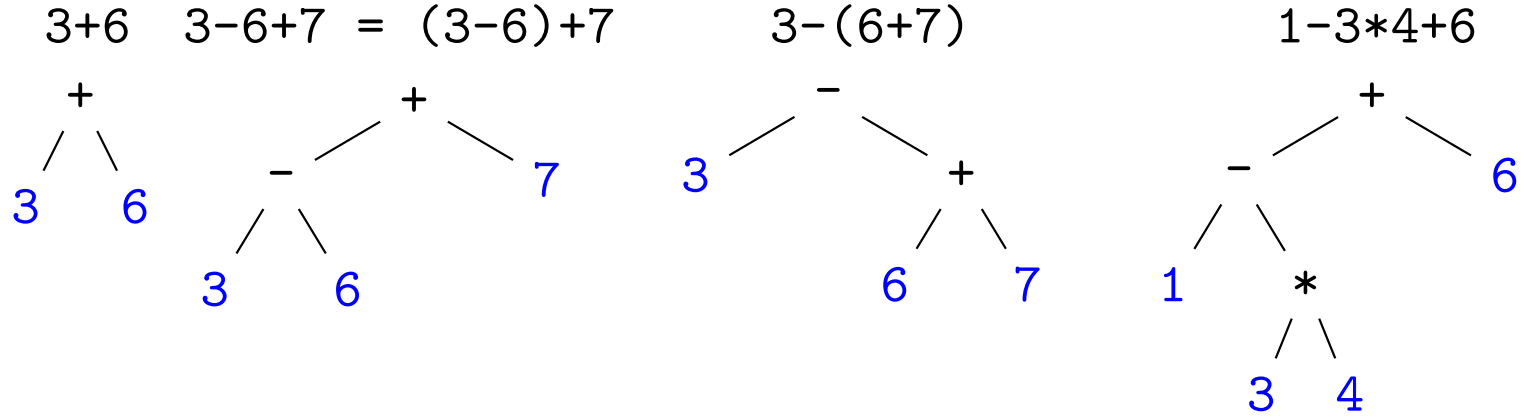
- Az ⟨ akif ⟩ adatstruktúra egy lehetséges megvalósítása C nyelven:

```
enum akif_fajta {Number, Plus, Minus, Times, Div};
struct akif { enum akif_fajta fajta;
              union { struct { int ertek;
                             } szam;
                      struct { struct akif *bal;
                               struct akif *jobb;
                             } osszetett;
                    } u;
            };
```

Az aritmetikai kifejezések matematikai absztrakciója

Milyen matematikai struktúra feleltethető meg ⟨ akif ⟩-nek?

- Egy bináris fa
  - melynek levelei számokkal vannak címkézve
  - csomópontjai pedig a négy alapművelet valamelyikével
- Példák:



A feladat megoldásának terve

- Egyszerűsített feladat:
  levelek: `1`, `3`, `4`, `6`; *műveletek:* `-`, `*`; a kifejezés elvárt értéke: `-11`
- Állítsuk elő az adott levelekkel bíró összes ⟨ akif ⟩-et, majd válogassuk ki azokat, amelyek értéke az adott szám (**brute-force, generate-and-test**)
  (n a levelek, m a műveletek száma, a példában $n = 4, m = 2$):
  1. Állítsuk elő az összes adott levélszámú, címkézetlen bináris fát (legyen f ezek száma, pl. $n = 4$ esetén $f = 5$)
  2. A csomópontokba minden lehetséges módon helyezzünk el műveleti jeleket ($f \cdot m^{n-1}$ fa)
  3. Állítsuk elő a levelek összes permutációját ($n!$ db)
  4. A csomópont-címkézett fa leveleibe írjunk be minden permutációt ($f \cdot m^{n-1} \cdot n!$ darab ⟨ akif ⟩, a példában $5 \cdot 2^{4-1} \cdot 4! = 960$)
- Számítsuk ki minden így előállított ⟨ akif ⟩ értékét, adjuk vissza azokat, amelyekre ez az elvárt számértékkel egyezik
- A feladat megoldásai: `4-(6-1)*3`, `1-(6-3)*4`, `4-3*(6-1)`, `1-4*(6-3)`.

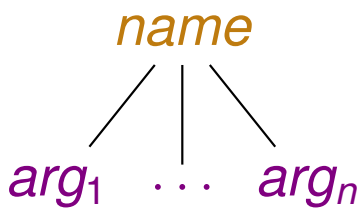
Tartalom

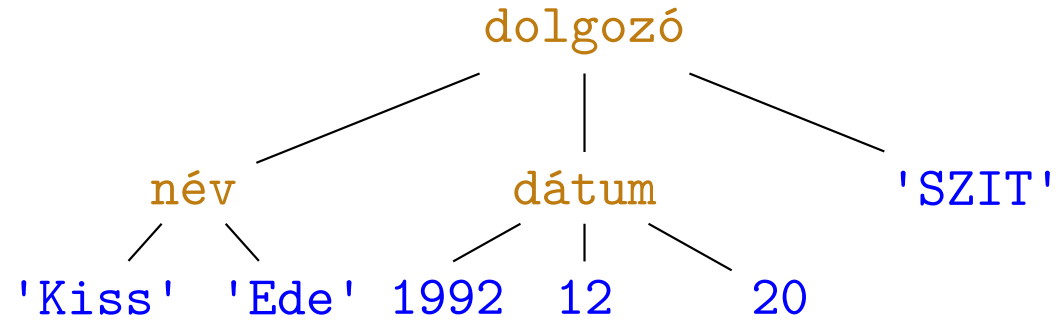
1. Programozás Prologban
   - Vezérlési eljárások
   - DCG – a Definite Clause Grammar kiterjesztés
   - Egy összetettebb példaprogram
   - Összefoglaló példa
   - A példa Prolog változata

Ismétlés: A Prolog nyelv adatfogalma

A Prolog adatokat Prolog kifejezésnek hívjuk (angolul: term). Fajtái:

- egyszerű kifejezés: számkonstans (pl. `3`, `4.25`, `-5`), névkonstans (pl. `alma`, `'SZIT'`, `+`), vagy változó (pl. `X`, `_`)
- összetett kifejezés (struktúra) kanonikus alakja: $name(arg_1,\ldots,arg_n)$
  - *name* egy névkonstans, az arg_i mezők tetsz. Prolog kifejezések
  - példa: `dolgozó(név('Kiss','Ede'), dátum(1992,12,20),'SZIT')`.
  - az összetett kifejezések valójában fastruktúrát alkotnak:





- a Prolog változó a matematikában használt változónak felel meg: egyetlen, esetleg még ismeretlen adatot jelent. A változó legfeljebb egyszer kaphat értéket. Egy Prolog változó összetett kifejezés részeként többször is megjelenhet, pl. `név(X,X)` egy olyan embert jelöl, akinek a vezetékneve és keresztneve ugyanaz (`X`).

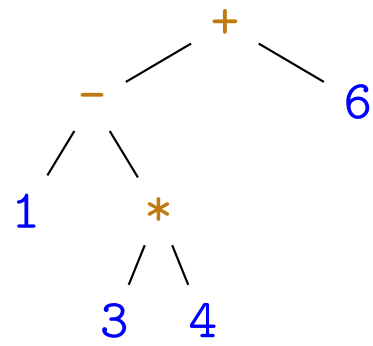
Ismétlés: Szintaktikus „édesítőszerek” Prologban

- Egy- és kétargumentumú struktúrák operátoros (infix, prefix stb.) írásmódja: `1+2≡+(1,2)`

```
| ?- write_canonical(1-3*4+6).
```

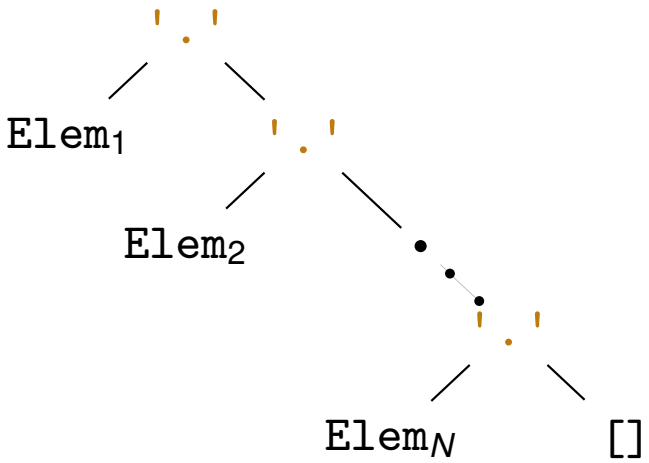
(Ez a beépített elj. a kanonikus alakot írja ki:)

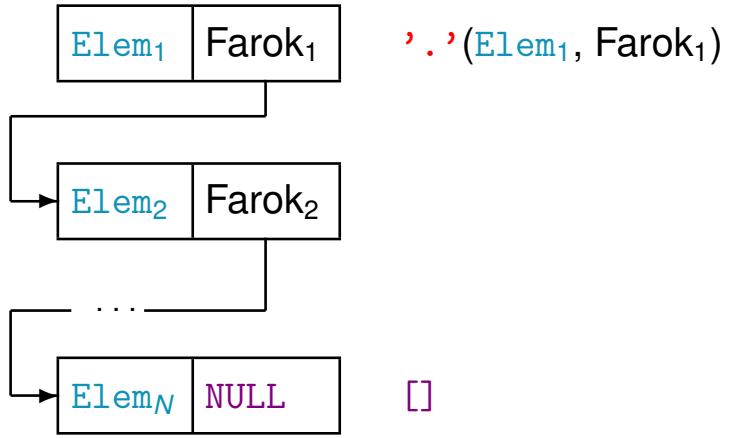
```
+(-(1,*(3,4)),6)
```



- Listák mint speciális struktúrák

```
| ?- write_canonical([6,4,3]).
'.'(6,'.'(4,'.'(3,[])))
```





Aritmetikai kifejezések Prologban – 1. lépés: ellenőrzés

Írjunk egy `kif` nevű Prolog eljárást (azaz Boole-értékű függvényt)!
(„igaz” eredmény esetén sikerről, egyébként meghiúsulásról beszélünk.)
A `kif(X)` hívás sikeresen fut le, ha `X` egy olyan kifejezés, amely számokból a négy alapművelet (+, -, *, /) segítségével épül fel (röviden, ha `X` **helyes**).

- Az alábbi sorokat helyezzük el pl. a `kif0.pl` file-ban:

```
% kif(K): K helyes, azaz K egy számokból, alapműveletekkel képzett kifejezés.
kif(K) :-   number(K).       % K helyes, ha K szám. (number beépített elj.)
kif(X+Y) :- kif(X), kif(Y). % X+Y helyes, ha X helyes és Y helyes
kif(X-Y) :- kif(X), kif(Y). % X-Y helyes, ha ...
kif(X*Y) :- kif(X), kif(Y). % ...
kif(X/Y) :- kif(X), kif(Y).
```

- Betöltése: `| ?- compile(kif0).` vagy `| ?- consult(kif0).`
- Futtatás nyomkövetés nélkül és nyomkövetéssel (`consult`-ot követően):

```
| ?- kif(alma).
no
| ?- kif(1+2).
yes
| ?-
```

```
| ?- trace, kif(alma).
        1      1 Call: kif(alma) ?
        2      2 Call: number(alma) ?
        2      2 Fail: number(alma) ?
        1      1 Fail: kif(alma) ?
no
```

Aritmetikai kifejezések ellenőrzése – továbbfejlesztett változat

- A `kif` Prolog eljárás segédeljárást használó változata:

```
% kif2(K): K számokból a négy alapművelettel képzett kifejezés.
kif2(Kif) :-
    number(Kif).
kif2(Kif) :-
    alap4(X, Y, Kif),
    kif2(X), kif2(Y).
```

- Az `alap4` segédeljárás:

```
% alap4(X, Y, Kif): A Kif kifejezés az X és Y kifejezésekből
%                    a négy alapművelet egyikével áll elő.
alap4(X, Y, X+Y).
alap4(X, Y, X*Y).
alap4(X, Y, X-Y).
alap4(X, Y, X/Y).
```

Aritmetikai kifejezés leveleinek listája

- Az alábbi eljárás ellenőrzi, hogy `Kif` egy számokból alapműveletekkel felépített kifejezés-e, és ha igen, előállítja ennek levéllistáját

```
% kif_levelek(Kif, L): A számokból alapműveletekkel felépülő Kif
%                       kifejezés leveleiben levő számok listája L.
kif_levelek(Kif, L) :-
    number(Kif), L = [Kif].   % L egyelemű, Kif-ből álló lista
    % A = B egy infix alakban írható beépített eljárás, működése:
    % A-t és B-t egyesíti: változóbehelyettesítésekkel azonos alakra hozza.
kif_levelek(Kif, L) :-
    alap4(K1, K2, Kif),
    kif_levelek(K1, L1),
    kif_levelek(K2, L2),
    append(L1, L2, L).
| ?- kif_levelek(2/3-4*(5+6), L).     ⟶ L = [2,3,4,5,6]
```

- Az `append` egy beépített eljárás, fejkommentje és példafutása:

```
% append(L1, L2, L): Az L1 és L2 listák összefűzése az L lista, L1 ⊕ L2 = L.
| ?- append([1,2], [3,4], L).          ⟶ L = [1,2,3,4]
```

Adott levéllistájú aritmetikai kifejezések előállítása

- A `kif_levelek` eljárás sajnos nem használható „visszafelé”, végtelen ciklusba esik, lásd pl. `| ?- kif_levelek(Kif, [1]).`
- Ez javítható a hívások átrendezésével és új feltételek beszúrásával:

```
% kif_levelek(+Kif, -L):
% Kif levéllistája L.
kif_levelek(Kif, L) :-
    L = [Kif],
    number(Kif).
kif_levelek(Kif, L) :-
    alap4(K1, K2, Kif),

    kif_levelek(K1, L1),
    kif_levelek(K2, L2),
    append(L1, L2, L).
```

```
% kif_levelek_v(-Kif, +L):
% Kif levéllistája L.
kif_levelek_v(Kif, L) :-
    L = [Kif],
    number(Kif).
kif_levelek_v(Kif, L) :-
    append(L1, L2, L),
    L1 \= [], L2 \= [],
    % L1, L2 nem-üres listák
    kif_levelek_v(K1, L1),
    kif_levelek_v(K2, L2),
    alap4(K1, K2, Kif).
```

- Az `X \= Y` beépített eljárás *csakkor* sikerül, ha `X = Y` meghiúsul.

```
| ?- kif_levelek_v(K, [1,3,4]).
K = 1+(3+4) ? ; K = 1-(3+4) ? ; K = 1*(3+4) ? ; K = 1/(3+4) ? ; ...
```

Adott értékű kifejezés előállítása

- Bevezető példánk megoldásához szükségesek további nyelvi elemek
  - A `lists` könyvtárban található `permutation` eljárás:

    `% permutation(L, PL): PL` *az* `L` *lista permutációja.*
  - Az `=:=` beépített aritmetikai eljárás mindkét argumentumában aritmetikai kifejezést vár, ezeket kiértékeli, és *csakkor* sikerül, ha az értékek aritmetikailag megegyeznek, pl.

    `| ?- 4+2 =:= 3*3.` ⟶ `no` `| ?- 4+2 =:= 2*3.` ⟶ `yes`
  - A `catch(Hívás, Kiv, KivHívás)` beépített eljárás végrehajtja `Hívás`-t, és ha ez egy `Kiv` kivételt dob, végrehajtja `KivHívás`-t.
- A „kedvcsináló” feladat megoldása:

```
% levelek_ertek_kif(L, Ertek, Kif): Kif az L listabeli számokból a négy
% alapművelet segítségével felépített olyan kifejezés, amelynek értéke Ertek.
levelek_ertek_kif(L, Ertek, Kif) :-
    permutation(L, PL), kif_levelek_v(Kif, PL),
    catch(Kif =:= Ertek, _, fail).  % fail mindig meghiúsul.

| ?- levelek_ertek_kif([1,3,4], 11, Kif).
Kif = 3*4-1 ? ; Kif = 4*3-1 ? ; no
```

Adott értékű kifejezés előállítása – a teljes kód

```
:- use_module(library(lists), [permutation/2]). % importálás

% Kif az L listabeli számokból felépített, Ertek értékű kifejezés.
levelek_ertek_kif(L, Ertek, Kif) :-
    permutation(L, PL), kif_levelek_v(Kif, PL),
    catch(Kif =:= Ertek, _, fail).

% Az alapműveletekkel felépített Kif levéllistája L.
kif_levelek_v(Kif, L) :-
    L = [Kif], number(Kif).
kif_levelek_v(Kif, L) :-
    append(L1, L2, L), L1 \= [], L2 \= [],
    kif_levelek_v(K1, L1), kif_levelek_v(K2, L2),
    alap4(K1, K2, Kif).

% alap4(X, Y, K): A K kifejezés X-ből és Y-ból egy alapművelettel áll elő.
alap4(X, Y, X+Y).
alap4(X, Y, X-Y).
alap4(X, Y, X*Y).
alap4(X, Y, X/Y).
```