A PROLOG NYELV KÖZELÍTŐ SZINTAXISA

Predikátumok, klózok

- Példa:

```
% két klózból álló predikátum definíciója, funktora: sum_tree/2
sum_tree(leaf(Val), Val).          %                      1. klóz, tényállítás
sum_tree(node(Left,Right), S) :-   %              fej   \
        sum_tree(Left, S1),        % cél        \       |
        sum_tree(Right, S2),       % cél        | törzs | 2. klóz, szabály
        S is S1+S2.                % cél        /       /
```

- Szintaxis:

⟨ Prolog program ⟩	::=	⟨ predikátum ⟩ . . .	
⟨ predikátum ⟩	::=	⟨ klóz ⟩ . . .	{azonos funktorú}
⟨ klóz ⟩	::=	⟨ tényállítás ⟩.␣ |	
		⟨ szabály ⟩.␣	{klóz funktora = fej funktora}
⟨ tényállítás ⟩	::=	⟨ fej ⟩	
⟨ szabály ⟩	::=	⟨ fej ⟩ :- ⟨ törzs ⟩	
⟨ törzs ⟩	::=	⟨ cél ⟩, . . .	
⟨ cél ⟩	::=	⟨ kifejezés ⟩	
⟨ fej ⟩	::=	⟨ kifejezés ⟩	

Prolog programok formázása

- Programok javasolt formázása:
  - Az egy predikátumhoz tartozó klózok legyenek egymás mellett a programban, közéjük ne tegyünk üres sort. A predikátumokat válasszuk el üres sorokkal.
  - A klózfejet írjuk sor elejére, minden célt lehetőleg külön sorba, néhány szóközzel beljebb kezdve

Prolog kifejezések

- Példa — egy klózfej mint kifejezés:

```
%     sum_tree(node(Left,Right), S)    % összetett kif., funktora sum_tree/2
%         |         |             |
% struktúranév      |            argumentum, változó
%                   \- argumentum, összetett kif.
```

- Szintaxis:

⟨ kifejezés ⟩	::=	⟨ változó ⟩ |	{Nincs funktora}
		⟨ konstans ⟩ |	{Funktora: ⟨ konstans ⟩ / 0 }
		⟨ összetett kifejezés ⟩ |	{Funktora: ⟨ struktúranév ⟩ / ⟨ arg.szám ⟩ }
		(⟨ kifejezés ⟩)	{Operátorok miatt, ld. később}
⟨ konstans ⟩	::=	⟨ névkonstans ⟩ |	
		⟨ számkonstans ⟩	
⟨ számkonstans ⟩	::=	⟨ egész szám ⟩ |	
		⟨ lebegőpontos szám ⟩	
⟨ összetett kifejezés ⟩	::=	⟨ struktúranév ⟩ (⟨ argumentum ⟩, . . .)	
⟨ struktúranév ⟩	::=	⟨ névkonstans ⟩	
⟨ argumentum ⟩	::=	⟨ kifejezés ⟩	

Lexikai elemek

- Példák:

```
% változó:             Fakt FAKT _fakt X2 _2 _
% névkonstans:         fakt ≡ 'fakt' 'István' [] ; ',' += ** \= ≡ '\\='
% számkonstans:        0 -123 10.0 -12.1e8
% nem névkonstans:     !=, Istvan
% nem számkonstans:    1e8 1.e2
```

- Szintaxis:

⟨ változó ⟩	::=	⟨ nagybetű ⟩⟨ alfanumerikus jel ⟩. . . | _ ⟨ alfanumerikus jel ⟩. . . |
⟨ névkonstans ⟩	::=	' ⟨ idézett karakter kar ⟩. . . ' | ⟨ kisbetű ⟩⟨ alfanumerikus jel ⟩. . . | ⟨ tapadó jel ⟩. . . | ! | ; | [] | {}
⟨ egész szám ⟩	::=	{előjeles vagy előjeltelen számjegysorozat}
⟨ lebegőpontos szám ⟩	::=	{belsejében tizedespontot tartalmazó számjegysorozat esetleges exponenssel}
⟨ idézett karakter ⟩	::=	{tetszőleges nem ' és nem \ karakter} | \ ⟨ escape szekvencia ⟩
⟨ alfanumerikus jel ⟩	::=	⟨ kisbetű ⟩ | ⟨ nagybetű ⟩ | ⟨ számjegy ⟩ | _
⟨ tapadó jel ⟩	::=	+ | - | * | / | \ | $ | ^ | < | > | = | ` | ~ | : | . | ? | @ | # | &

Szintaktikus édesítőszer: operátorok

- Példa:

```
% S is -S1+S2 ekvivalens az is(S, +(-(S1),S2)) kifejezéssel
```

- Operátoros kifejezések

⟨ összetett kifejezés ⟩ ::=

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

⟨ operátornév ⟩ ::= ⟨ struktúranév ⟩ {ha operátorként lett definiálva}

- Operátor-kezelő beépített predikátumok:
  - `op(Prioritás, Fajta, OpNév)` vagy `op(Prioritás, Fajta, [OpNév`$_1$`,OpNév`$_2$`,...])`:
    - `Prioritás`: 0–1200 közötti egész
    - `Fajta`: az `yfx`, `xfy`, `xfx`, `fy`, `fx`, `yf`, `xf` névkonstansok egyike
    - `OpNév`: tetszőleges névkonstans
    - pozitív prioritás esetén definiálja az operátor(oka)t, 0 prioritás esetén megszünteti azokat.
  - `current_op(Prioritás, Fajta, OpNév)`: felsorolja a definiált operátorokat.

Szabványos, beépített operátorok

Szabványos operátorok

```
1200  xfx  :- -->
1200   fx  :- ?-
1100  xfy  ;
1050  xfy  ->
1000  xfy  ','
 900   fy  \+
 700  xfx  < = \= =..
           =:= =< == \==
           =\= > >= is
           @< @=< @> @>=
 500  yfx  + - /\ \/
 400  yfx  * / // rem
           mod << >>
 200  xfx  **
 200  xfy  ^
 200   fy  - \
```

Egyéb beépített operátorok

```
1150   fx  dynamic multifile
           block meta_predicate
 900   fy  spy nospy
 550  xfy  :
 500  yfx  #
 500   fx  +
```

Operátorok jellemzői

- Egy operátort jellemez a fajtája és prioritása
- A fajta meghatározza az operátor-osztályt (írásmódot) és az asszociatívitást:

Fajta			Osztály	Értelmezés
bal-asszoc.	jobb-asszoc.	nem-asszoc.		
`yfx`	`xfy`	`xfx`	infix	`A op B` $\equiv$ `op(A, B)`
	`fy`	`fx`	prefix	`op A` $\equiv$ `op(A)`
`yf`		`xf`	posztfix	`A op` $\equiv$ `op(A)`

- Több-operátoros kifejezésben a zárójelezést a prioritás és az asszociatívitás határozza meg, pl.
  - `a/b+c*d` $\equiv$ `(a/b)+(c*d)` mert `/` és `*` prioritása 400, ami **kisebb** mint a `+` prioritása (500) (kisebb prioritás = **erősebb** kötés).
  - `a+b+c` $\equiv$ `(a+b)+c` mert a `+` operátor fajtája `yfx`, azaz bal-asszociatív (balra köt, balról jobbra zárójelez)
  - `a^b^c` $\equiv$ `a^(b^c)` mert a `^` operátor fajtája `xfy`, azaz jobb-asszociatív (jobbra köt, jobbról balra zárójelez)
  - `a=b=c` szintaktikusan hibás, mert az `=` operátor fajtája `xfx`, azaz nem-asszociatív

Operátorok: zárójelezés

- Induljunk ki egy teljesen zárójelezett, több operátort tartalmazó kifejezésből!
- Egy részkifejezés prioritása a (legkülső) operátorának a prioritása.
- Egy *op* prioritású operátor *ap* prioritású argumentumát körülvevő zárójelpár elhagyható ha:
  - $ap < op$ pl. `a+(b*c)` $\equiv$ `a+b*c` ($ap = 400, op = 500$)
  - $ap = op$, jobb-asszociatív operátor jobboldali argumentuma esetén, pl. `a^(b^c)` $\equiv$ `a^b^c` ($ap = 200, op = 200$)
  - $ap = op$, bal-asszociatív operátor baloldali argumentuma esetén, pl. `(1+2)+3` $\equiv$ `1+2+3`. Kivétel: ha a baloldali argumentum operátora jobb-asszociatív, azaz az előző feltétel alkalmazható.
- Példa a kivétel esetére:
  - `:- op(500, xfy, +^).`
    ```
    | ?- :- write((1 +^ 2) + 3), nl.   ⇒ (1+^2)+3
    | ?- :- write(1 +^ (2 + 3)), nl.   ⇒ 1+^2+3
    ```
  - tehát: konfliktus esetén az első operátor asszociativitása „győz”.

Operátorok — kiegészítő megjegyzések

- Azonos nevű, azonos osztályba tartozó operátorok egyidejűleg nem megengedettek.
- Egy program szövegében direktívákkal definiálhatunk operátorokat, pl.

```
:- op(500, xfx, --).                  :- op(450, fx, @).

sum_tree(@V, V).                      (...)
```

- A „vessző” kettős szerepe
  - struktúra-kifejezés argumentumait választja el
  - 1000 prioritású `xfy` operátorként működik: pl. `(p :- a,b,c) = :-(p,','(a,','(b,c)))`
  - a „pucér” vessző (`,`) nem névkonstans, de operátorként aposztrofok nélkül is írható.
  - struktúra-argumentumban 999-nél nagyobb prioritású kifejezést zárójelezni kell:

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

- Az egyértelmű elemezhetőség érdekében a Prolog szabvány kiköti, hogy
  - operandusként előforduló operátort zárójelbe kell tenni, pl. `Comp = (>)`
  - nem létezhet azonos nevű infix és posztfix operátor.
- Sok Prolog rendszerben nem kötelező betartani ezeket a megszorításokat.

Operátorok felhasználása

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

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

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

  - adatstruktúrák olvashatóbbá tételére, pl.

```
:- op(100, xfx, [.]).

sav(kén, h.2-s-o.4).
```

- Miért rosszak az operátorok?
  - egyetlen globális erőforrás, ez nagyobb projektben gondot okozhat.

Aritmetika Prologban

- Az operátorok teszik lehetővé azt is, hogy a matematikában ill. más programozási nyelvekben megszokott módon értékelhessünk kis aritmetikai kifejezéseket.
- Az `is` beépített predikátum egy aritmetikai kifejezést vár a jobboldalán (2. argumentumában), azt kiértékeli, és az eredményt egyesíti a baloldali argumentummal
- Az `=:=` beépített predikátum mindkét oldalán aritmetikai kifejezést vár, azokat kiértékeli, és csakkor sikerül, ha az értékek megegyeznek.
- Példák:

```
| ?- X = 1+2, write(X), write('  '), write_canonical(X), Y is X.
⇒              1+2                     +(1,2)    ⟹ X = 1+2, Y = 3 ? ; no
| ?- X = 4, Y is X/2, Y =:= 2.    ⟹ X = 4, Y = 2.0 ? ; no
| ?- X = 4, Y is X/2, Y = 2.      ⟹ no
```

- **Fontos**: az aritmetikai operátorokkal (+,-,...) képzett kifejezések **összetett Prolog kifejezés**t jelentenek. Csak az aritmetikai beépített predikátumok értékelik ki ezeket!
- A Prolog kifejezések alapvetően szimbolikusak, az aritmetikai kiértékelés a „kivétel”.

Klasszikus szimbolikus kifejezés-feldolgozás: deriválás

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

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

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

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

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

| ?- deriv(I, 0).
  ⟹  no
```

Operátoros példa: polinom behelyettesítési értéke

- Formula: számokból és az ‘x’ névkonstansból ‘+’ és ‘*’ operátorokkal felépülő kifejezés.
- A feladat: Egy formula értékének kiszámolása egy adott x érték esetén.

```
% erteke(Kif, X, E): A Kif formula értéke E, az x=X behelyettesítéssel.
erteke(x, X, E) :-
        E = X.
erteke(Kif, _, E) :-
        number(Kif), E = Kif.
erteke(K1+K2, X, E) :-
        erteke(K1, X, E1),
        erteke(K2, X, E2),
        E is E1+E2.
erteke(K1*K2, X, E) :-
        erteke(K1, X, E1),
        erteke(K2, X, E2),
        E is E1*E2.

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

Példasor: bináris fák kezelése

- Az egészekből álló bináris fa különböző meghatározásai:
  - Szöveges definícióként (ismétlés):
    - vagy egy levél (`leaf(V)`), ahol `V` egy egész szám
    - vagy egy csomópont (`node(L,R)`), ahol `L` és `R` egészekből álló bináris fák
  - Matematikai jelöléssel:

    $\mathtt{itree} \equiv \{\mathtt{leaf}(i) \mid i \in \mathtt{integer}\} \cup \{\mathtt{node}(l,r) \mid l,r \in \mathtt{itree}\}$
  - A Mercury típusos logikai programozási nyelv jelöléseivel:

    ```
    :- type itree --->  node(itree, itree) | leaf(int).
    ```
  - Egy **ellenőrző** Prolog predikátumként:

    ```
    itree(leaf(V)) :-
            integer(V).          % <--- osztályozó predikátum
    itree(node(L,R)) :-
            itree(L), itree(R).
    ```
- Az ilyen adattípust **megkülönböztetett unió**nak nevezzük, mert az unióban szereplő halmazokat az elemeik funktora megkülönbözteti (`leaf/1`, `node/2`)

Az egyesítés mint adat-építő és -kiválasztó művelet

- Példa:

```
% balcsonk(Fa, E): Fa egy olyan bináris fa, amelynek legfelső
%                  csomópontjában balra egy E értékű levél van
balcsonk(node(leaf(V),_), V).

% jobbcsonk(Fa, E): Fa egy olyan bináris fa, amelynek legfelső
%                   csomópontjában jobbra egy E értékű levél van
jobbcsonk(node(_,leaf(V)), V).
```

- A Prolog egyesítés egyaránt használható adat-építésre (konstrukció) és -kiválasztásra (szelekció). Példák:
  - Mindkét argumentum bemenő: ellenőrzés.

    ```
    | ?- Fa=node(leaf(1),leaf(2)), balcsonk(Fa, 1). ⟹ yes
    ```

  - A fa adott, az érték kimenő: levélérték elővétele (vagy meghiúsulás).

    ```
    | ?- Fa=node(leaf(1),leaf(2)), balcsonk(Fa, B), jobbcsonk(Fa, J).
    ⟹ B = 1, J = 2 ? ; no
    | ?- balcsonk(node(node(leaf(1),leaf(2)),leaf(3)), B). ⟹ no
    ```

  - Az érték adott, a fa kimenő: fa építése.

    ```
    | ?- balcsonk(Fa, 1), jobbcsonk(Fa, 2).⟹ Fa=node(leaf(1),leaf(2)) ? ; no
    ```

Kifejezések összerakása (folyt.)

- Vizsgáljuk a fa felépítésének részleteit!

célsorozat:	`:- balcsonk(Fa,`	`1),`	`jobbcsonk(Fa, 2)`
klózfej:	`balcsonk(node(leaf(_B),_A), _B)`		
behelyettesítés:	`_B = 1, Fa= node(leaf(1),_A)`		
új célsorozat:	`:- jobbcsonk(node(leaf(1),`	`_A),`	`2)`
klózfej:	`jobbcsonk(node(_C,`	`leaf(_D)),`	`_D)`
behelyettesítés:	`_D = 2, _A = leaf(2), _C = leaf(1)`		
eredmény:	`Fa = node(leaf(1),leaf(2))`		

- Kövessük nyomon a fenti végrehajtást Prolog hívásonként:
  - `| ?- balcsonk(Fa, 1).` $\Longrightarrow$ `Fa = node(leaf(1),_A) ? ; no`

    `Fa` értéke egy változót tartalmazó Prolog adatstruktúra, mindazon összetett (nem levél) fákat jelenti, amelyek baloldali részfája az 1 értékű levél. Ez egy **kifejezés-minta**.
  - `| ?- Fa = node(leaf(1),_A), jobbcsonk(Fa, 2).`

    $\Longrightarrow$ `Fa = node(leaf(1),leaf(2)) ? ; no`

    A `jobbcsonk` hívás a `Fa` kifejezés-mintát, **finomítja**, ebben az esetben egy tömör fára szűkíti le.

A logikai változó

- A logikai változó fogalma:
  - kifejezésként, kifejezésben egyaránt előfordulhat
  - a változók egymással is azonossá tehetők: pl. két azonos változó egy kifejezésben.
  - a változó „teljes jogú” állampolgár a (rész)kifejezések világában
- SML-ben is van mintaillesztés, de a minta csak szétszedésre használható, összerakásra nem; a mintabeli változók mindig (tömör) értéket kapnak.
- (Egyes újabb funkcionális nyelvek, pl. az Oz nyelv, támogatják a logikai változókat.)
- Példa: Az alábbi célsorozat egy két **azonos** levélből álló bináris fát épít fel a `Fa` változóban. A levelek értéke **azonos** lesz a célsorozatbeli `X` változóval:

```
balcsonk(node(leaf(V),_), V).
jobbcsonk(node(_,leaf(V)), V).
```

```
| ?- balcsonk(Fa, X), jobbcsonk(Fa, X). ⟹ Fa = node(leaf(X),leaf(X)) ? ; no
```

- Ha az összekapcsolt változók bármelyike értéket kap, a többi is erre az értékre helyettesítődik:

```
| ?- balcsonk(Fa, X), jobbcsonk(Fa, X), X = 1.
                              ⟹ X = 1, Fa = node(leaf(1),leaf(1)) ? ; no
| ?- balcsonk(Fa, X), jobbcsonk(Fa, X), balcsonk(Fa, 2).
                              ⟹ X = 2, Fa = node(leaf(2),leaf(2)) ? ; no
```

Bináris fák kezelése — fa levele

- Írjunk egy predikátumot annak eldöntésére, hogy egy adott érték szerepel-e egy fa levelében!
-
```
% fa_levele(Fa, Ertek): A Fa bináris fa levelében szerepel az Ertek szám.
fa_levele(leaf(V), V).        % ha a fa egyetlen levélből áll és a levélbeli
                              % érték megegyezik a keresettel, akkor "siker''
fa_levele(node(L,_), V) :-
        fa_levele(L, V).      % ha szerepel a bal részfában → az egészben is
fa_levele(node(_,R), V) :-
        fa_levele(R, V).      % ha szerepel a jobb részfában → az egészben is
```
- Az aláhúzásjel egy ún. semmis (void) változó, ennek minden előfordulása különböző változó!
- Példák: ellenőrzés, adott fa leveleinek felsorolása, adott levelű fák felsorolása (∞ keresési tér).

```
| ?- fa_levele(node(node(leaf(1),leaf(2)),leaf(7)), 2).  ⟹ yes
| ?- fa_levele(node(node(leaf(1),leaf(2)),leaf(7)), 3).  ⟹ no
| ?- fa_levele(node(leaf(1),leaf(7)), E). ⟹ E = 1 ? ; E = 7 ? ; no
| ?- fa_levele(Fa, 3). ⟹ Fa = leaf(3) ? ; Fa = node(leaf(3),_A) ? ;  ...
```

Összetett adatstruktúrák konjunktív és diszjunktív bejárása

- Prologban egy összetett adatstruktúrát kétféleképpen lehet bejárni:
  - konjunktívan: a részek bejárása ÉS kapcsolatban van, általában egy eredményt ad
    - pl. fa összegzése (`sum_tree`), fa ellenőrzése (`itree`), fa kiírása:

```
% faki(Fa): Fa kiírható (mindig teljesül :-). Mellékhatásként kiírja a Fa fát.
faki(leaf(V)) :-
        write(@), write(V).     % A write(X) beépített pred. kiírja az X kifejezést.
faki(node(L,R)) :-
        write('('), faki(L), write(' -- '), faki(R), write(')').

| ?- faki(node(node(leaf(1),leaf(8)),leaf(7))).     ⇒ ((@1 -- @8) -- @7)
yes
```

  - diszjunktívan: a részek bejárása VAGY kapcsolatban van, visszalépéskor új eredmény
    - pl. fa leveleinek felsorolása (`fa_levele`)
- A diszjunktív, felsoroló bejárás könnyen kiegészíthető további feltételekkel
  - Keressük egy fának az `(5,10)` intervallumba eső leveleit:

```
| ?- _Fa = node(node(leaf(1),leaf(8)),leaf(7)), fa_levele(_Fa, E), 5 < E, E < 10.
     ⟹ E = 8 ? ; E = 7 ? ; no
| ?- _Fa = (...),  fa_levele(_Fa, E), 5 < E, E < 10, write(E), write(' '), fail.
     ⇒ 8 7   ⟹ no
```

- A `fail` beépített predikátum mindig meghiúsul, pl. ún. visszalépéses ciklus szervezésére jó.

Levél elhagyása bináris fából

- Írjunk egy predikátumot annak eldöntésére, hogy egy adott érték szerepel-e egy összetett fa levelében! A predikátum adja vissza a levél elhagyása után fennmaradó fát!

```
% flm(Fa, Ertek, Marad): A Fa összetett bináris fa egy Ertek értékű
% levelének elhagyása után marad a Marad fa. (flm = fa_level_maradek)
flm(node(leaf(V),T), V, T).    % ha a bal részfa a keresett levél
                               % akkor a jobb részfa a maradék
flm(node(T,leaf(V)), V, T).    % ugyanez jobboldali levél esetére
flm(node(L0,R), V, node(L,R)) :-
        flm(L0, V, L).         % ha a bal részfából elhagyható a levél
                               % akkor ennek maradéka, kiegészítve
                               % a jobb részfával, lesz a teljes fa maradéka
flm(node(L,R0), V, node(L,R1)) :-
        flm(R0, V, R1).        % ugyanez jobb részfa esetére
```

- Az `flm/3` predikátum használható ellenőrzése, de fa szétbontására is:

```
| ?- flm(node(leaf(1),node(leaf(2),leaf(3))), 2, T).  ⟹
   T = node(leaf(1),leaf(3)) ? ; no
| ?- flm(node(leaf(1),node(leaf(2),leaf(3))), 7, T).  ⟹ no
| ?- flm(node(leaf(1),node(leaf(2),leaf(3))), X,  T). ⟹
   T = node(leaf(2),leaf(3)), X = 1 ? ;
   T = node(leaf(1),leaf(3)), X = 2 ? ;
   T = node(leaf(1),leaf(2)), X = 3 ? ; no
```

Levél beszúrása bináris fába

- Írjunk egy predikátumot arra, hogy egy adott értékű levelet egy fába minden lehetséges módon beszúrjon!
- Nem kell irnunk, már megírtuk! Az `flm` predikátum erre is jó:

```
% flm(Fa, Ertek, Marad): A Fa összetett bináris fa egy Ertek értékű
% levelének elhagyása után marad a Marad fa. Röviden: Fa − Ertek = Marad.

% flm(Fa, Ertek, Marad): A Fa (összetett) bináris fa úgy áll elő, hogy
% a Marad fába beszúrunk egy E értékű levelet. Fa = Marad + Ertek.
flm(node(leaf(V),T), V, T).     % Egy T fába beszúrhatunk egy levelet
(...)                           % úgy, hogy az egylevelű fát T elé tesszük
```

- Példák:

```
| ?- flm(Fa, 2, leaf(1)), faki(Fa), write(' '), fail.
(@2 -- @1) (@1 -- @2)                              ⟹ no
| ?- flm(Fa0, 2, leaf(1)), flm(Fa, 3, Fa0), faki(Fa), write(' '), fail.
(@3 -- (@2 -- @1)) ((@2 -- @1) -- @3) ((@3 -- @2) -- @1) ((@2 -- @3) -- @1)
(@2 -- (@3 -- @1)) (@2 -- (@1 -- @3)) (@3 -- (@1 -- @2)) ((@1 -- @2) -- @3)
((@3 -- @1) -- @2) ((@1 -- @3) -- @2) (@1 -- (@3 -- @2)) (@1 -- (@2 -- @3))  ⟹ no

negylevelu(X, Y, Z, U, Fa) :-  % Fa az X, Y, Z, U levelekből áll
     flm(Fa0, Y, leaf(X)), flm(Fa1, Z, Fa0), flm(Fa, U, Fa1).

| ?- findall(Fa, negylevelu(1,3,4,6,Fa), Fak), length(Fak,Db). ⟹  Db = 120, Fak = (...)
```

Példa: adott értékű kifejezés előállítása

- A feladat: írjunk Prolog programot a következő feladvány megoldására:
  - Az 1, 3, 4, 6 számokból a négy alapművelet felhasználásával állítsuk elő a 24 számértéket!
  - Mind a négy számot fel kell használni, tetszőleges sorrendben.
  - Tetszőleges alapműveletek használhatók, tetszőlegesen zárójelezéssel.
- Már van egy predikátumunk (`negylevelu/5`), amely adott számokból tetszőleges fát épít.
- Definiáljunk egy predikátumot, amely egy fának megfelelő aritmetikai kifejezéseket készít!

```
% fa_kif(Fa, Kif): Kif a Fa fával azonos alakú, azonos számokból álló
% aritmetikai kifejezés, amelyben a négy alapművelet fordulhat elő.
fa_kif(leaf(V), V).
fa_kif(node(L,R), Exp) :-
        fa_kif(L, E1),
        fa_kif(R, E2),
        alap4(E1, E2, Exp).

% alap4(X, Y, Kif): Kif 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).

| ?- fa_kif(node(leaf(1),node(leaf(2),leaf(3))), Kif).
Kif = 1+(2+3) ? ; Kif = 1-(2+3) ? ; Kif = 1*(2+3) ? ; Kif = 1/(2+3) ? ;
(...)
Kif = 1+2/3 ?   ; Kif = 1-2/3 ?   ; Kif = 1*(2/3) ? ; Kif = 1/(2/3) ? ; no
```

Példa: adott értékű kifejezés előállítása (folyt.)

- Korábban elkészített predikátumok:
  - adott számokból álló fákat felsoroló `negylevelu/5`
  - adott fával azonos szerkezetű aritmetikai kifejezéseket felsoroló `fa_kif/2`
- Ezekre építve könnyen megírható a feladvány megoldására használható predikátum:

```
% Kif egy a négy alapművelettel az X, Y, Z, U számokból
% felépített kifejezés, amelynek értéke Ertek.
negylevelu_erteke(X, Y, Z, U, Ertek, Kif) :-
        negylevelu(X, Y, Z, U, Fa),
        fa_kif(Fa, Kif),
        Kif =:= Ertek.

| ?- negylevelu_erteke(1,3,4,6,24,Kif).
Kif = 6 ? (1 ? 3 ? 4)  ? ; no
```

- Megjegyzések
  - Az aritmetikai eljárásokban a változók nem csak számokra, hanem tömör aritmetikai kifejezésekre is be lehetnek helyettesítve.
  - A `negylevelu_erteke` eljárás utolsó hívása helyett **nem** lenne jó: `Ertek is Kif`. Miért?