

Operátorok: zárójelezés

- Induljunk ki egy teljesen zárójelezett, több operátort tartalmazó kifejezésből!
- Egy részkiefejezés 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 aposztrófok 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 írására, pl.  $x \text{ is } (Y+3) \bmod 4$
  - 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, eljárásnév 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 írassunk le 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 csak akkor 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 kifejezést** 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
```

## A PROLOG LOGIKAI ALAPJAI

## 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
```

## Logikai alapfogalmak Prolog megfelelői

- A logika nyelvének elemei: (rövid összefoglaló, vö. a Matematikai Logika c. tárgy anyagával)
  - Kifejezés (*term*): változókból és konstansokból függvények segítségével épül fel, pl  $f(a, g(X))$ , ahol  $f$  kétargumentumú,  $g$  egyargumentumú függvéynév,  $a$  konstansnév (azaz 0-argumentumú függvéynév) és  $X$  változónév.
  - Elemi állítás: egy relációjel, megfelelő számú argumentummal ellátva, ahol az argumentumok kifejezések, pl.  $osztja(X * Y, X)$ .
  - Állítás (*formula*): elemi állításokból logikai összekötő jelekkel (pl.  $\wedge, \vee, \neg, \rightarrow$ ) és kvantorok ( $\forall, \exists$ ) alkalmazásával épül fel, pl.  $\forall X(X < 0 \rightarrow \neg X < X * 2)$ .
  - Prolog konvenciók:
    - A változóneveket nagybetűvel vagy aláhúzásjellel kezdjük.
    - Kétargumentumú függvénykifejezéseket, állításokat infix alakban is írhatunk, pl.  $X + 2 * Y \equiv +(X, *(2, Y)), X < X * 2 \equiv (X, *(X, 2))$
    - A függvények (és konstansok) nevét kisbetűvel kezdjük, vagy aposztrófok közé tesszük. Speciális jelek ill. jelsorozatokat is megengedettek függvény, konstans, vagy állítás neveként (pl.  $+, *, <$ ).

## A logika nyelvének megszorítása

- A következtetési folyamat hatékonyabbá tételéhez érdemes a logikai nyelvet szűkíteni.

- Bevezetjük a klóz (*clause*) fogalmát. Egy klóz az alábbi alakú logikai állítás:

$$\forall X_1 \dots X_j ((F_1 \vee \dots \vee F_n) \leftarrow (T_1 \wedge \dots \wedge T_m))$$

- az implikáció bal (következmény) oldala a klóz **feje**
- az implikáció feltétele a klóz **törzse**, a törzsbeli konjunkció elemeit (rész)**célok**nak is hívjuk
- $F_i$  és  $T_j$  elemi állítások,  $n, m \geq 0$ , azaz a fej és a törzs is lehet üres.
- $X_1 \dots X_j$ : a klózban szereplő összes változó.

- A fentivel ekvivalens logikai alak (vö.  $A \leftarrow B \equiv A \vee \neg B$ ):

$$\forall X_1, \dots, X_j (F_1 \vee \dots \vee F_n \vee \neg T_1 \vee \dots \vee \neg T_m)$$

- Klózok egyszerűsített írásmódja:  $F_1, \dots, F_n: \neg T_1, \dots, T_m$ . Ha  $m = 0$ , a  $:$  jelet elhagyjuk.

- Példák — vigyázat, ezek általános klózok, nem feltétlenül megengedettek Prologban!

```
ferfi(X), no(X) :- ember(X). % Aki ember az férfi vagy nő.
:- ferfi(X), no(X). % $\forall X \neg (ferfi(X) \wedge no(X))$
 % Nincs olyan dolog, ami férfi és nő is.

szereti(X, X) :- szent(X). % Minden szentnek maga felé hajlik a keze.
szent('István'). % István szent.
```

## PROLOG PROGRAMOK JELENTÉSE, VÉGREHAJTÁSA

## Deklaratív szemantika – klózok logikai alakja

- A matematikai logikában bevezetik az általános klóz fogalmát:

$$F_1, \dots, F_n: \neg T_1, \dots, T_m \quad \forall X (F_1 \vee \dots \vee F_n \vee \neg T_1 \vee \dots \vee \neg T_m)$$

- Definit klóz (*definite clause*) vagy Horn klóz (*Horn clause*):

olyan klóz, amelynek fejében legfeljebb egy elemi állítás szerepel ( $n \leq 1$ ).

- Horn klózok osztályozása

- Ha  $n = 1, m > 0$ , akkor a klózt **szabálynak** hívjuk, pl.

```
nagyszuloje(U,N) :- szuloje(U,Sz), szuloje(Sz,N).
```

logikai alak:  $\forall UNSz (nagyszuloje(U,N) \leftarrow szuloje(U,Sz) \wedge szuloje(Sz,N))$

ekvivalens alak:  $\forall UN (nagyszuloje(U,N) \leftarrow \exists Sz (szuloje(U,Sz) \wedge szuloje(Sz,N)))$

- $n = 1, m = 0$  esetén a klóz **tényállítás**, pl.

```
szuloje('Imre', 'István').
```

logikai alakja változatlan.

- $n = 0, m > 0$  esetén a klóz egy **célsorozat**, pl.

```
:- nagyszuloje('Imre', X).
```

logikai alak:  $\forall X \neg nagyszuloje('Imre', X)$ , azaz  $\neg \exists X nagyszuloje('Imre', X)$

- Ha  $n = 0, m = 0$ , akkor **üres klózról** beszélünk, jele:  $\square$ . Logikailag üres diszjunkció, azaz azonosan hamis.

## A logika függvényeinek szerepe Prologban

- A függvényjelek szerepe

- A Prolog az ún. egyenlőségmentes logikára (*equality-free logic*) épül, tehát két függvénykifejezés egyenlőségéről nem állíthatunk semmit.
- Emiatt Prolog-ban a logika függvényei *kizárólag* ún. konstruktor-függvények lehetnek:  $f(x_1, \dots, x_n) = z \Leftrightarrow (z = f(y_1, \dots, y_n) \wedge x_1 = y_1) \wedge \dots \wedge (x_n = y_n)$
- Például  $leaf(X) = Z \Leftrightarrow Z = leaf(Y) \wedge X = Y$ , azaz  $leaf(X)$  minden más értéktől különböző, egyedi érték.

- Példa:

```
sum_tree(leaf(Value), Value).
sum_tree(node(Left,Right), S) :-
 sum_tree(Left, S1), sum_tree(Right, S2), S is S1+S2.
```

```
| ?- sum_tree(node(leaf(1),leaf(2)), Sum). => Sum = 3 ?
| ?- sum_tree(Tree, 3). => Tree =leaf(3) ?
```

- A kérdésben felépített  $node(leaf(1), leaf(2))$  „függvénykifejezést” az eljárás *egyértelmű* módon szétbontja.
- A mintaillesztés (egyesítés) kétirányú: szétbontásra és építésre is alkalmas.

## A Prolog deklaratív szemantikája

- Deklaratív szemantika
  - Segédfogalom: egy kifejezés/állítás **példánya**: belőle változók behelyettesítésével előálló kifejezés/állítás.
  - Egy célsorozat lefutása **siker**, ha a célsorozat törzsének egy példánya logikai **következménye** a programnak (a programbeli klózok konjunkciójának).
  - A futás eredménye a példányt előállító **behelyettesítés**.
  - Egy célsorozat többféleképpen is lefuthat sikeresen.
  - Egy célsorozat futása **sikertelen**, ha egyetlen példánya sem következménye a programnak.
- Példa:
 

```
szuloje('Imre', 'István'). (sz1)
szuloje('Imre', 'Gizella'). (sz2)
szuloje('István', 'Géza'). (sz3)
szuloje('István', 'Sarolt'). (sz4)
szuloje('Gizella', 'Civakodó Henrik'). (sz5)
szuloje('Gizella', 'Burgundi Gizella'). (sz6)

nagyszuloje(Gy, N) :- szuloje(Gy, Sz), szuloje(Sz, N). (nsz)
:- nagyszuloje('Imre', N). (cel)
```

  - (sz1) + (sz3) + (nsz) következménye: nagyszuloje('Imre', 'Géza'), tehát (cel) sikeresen fut le az N = 'Géza' behelyettesítéssel.
  - Egy másik sikeres lefutás, pl. (sz1)+(sz4)+(nsz) alapján N = 'Sarolt'.

## Deklaratív szemantika

- Miért jó a deklaratív szemantika?
  - A program **dekomponálható**: külön-külön vizsgálhatjuk az egyes predikátumokat (sőt az egyes klózokat).
  - A program **verifikálható**: a predikátumok szándékolt jelentésének ismeretében eldönthető, hogy az egyes klózok igaz állításokat fogalmaznak-e meg.
  - Egy predikátum szándékolt jelentését nagyon fontos egy ún. **fejkommentben**, azaz az argumentumok kapcsolatát leíró kijelentő mondatban megfogalmazni. Példák:
    - Fejkommentek: % szuloje(Gy, Sz): Gy szülője Sz.  
% nagyszuloje(Gy, NSz): Gy nagyszülője NSz.

```
nagyszuloje(Gy, N) :- szuloje(Gy, Sz), szuloje(Sz, N).
A klóz jelentése: Ha Gy szülője Sz és Sz szülője N, akkor Gy nagyszülője N. Ez megfelel elvárásainknak, igaz állításként elfogadható.
```
  - Fejkommentek: % sum\_tree(T, Sum): A T fa levélösszege Sum.  
% E is Kif: A Kif aritm. kif. értéke E. (is infix!)

```
sum_tree(node(L,R), S) :- sum_tree(L, S1), sum_tree(R, S2), S is S1+S2.
A klóz jelentése: Ha az L fa levélösszege S1, az R fa levélösszege S2, és S1+S2 értéke S akkor a node(L,R) fa levélösszege S. Ez is egy igaz állítás.
```

## Deklaratív szemantika (folyt.)

- Miért nem elég a deklaratív szemantika?
  - A deklaratív szemantika egy általános következményfogalomra épít.
  - A következtetés szükségképpen többirányú, tehát kereséssel jár.
  - Végtelen keresési tér esetén a következtető is **végtelen ciklusba** eshet.
  - Véges keresési tér esetén is lehet a keresés nagyon **rossz hatékonyságú**.
  - Egyes **beépített predikátumok** csak bizonyos feltételek mellett képesek működni. Pl. S is S1+S2 hibát jelez, ha S1 vagy S2 ismeretlen mennyiség. Emiatt
 

```
sum_tree(node(L,R), S) :- S is S1+S2, sum_tree(L, S1), sum_tree(R, S2).
```

 logikailag helyes, de működésképtelen.
- Ezek miatt fontos, hogy a Prolog programozó ismerje a Prolog pontos végrehajtási mechanizmusát is, azaz a nyelv **procedurális szemantikáját**.
- Jelszó: **Gondolkodj deklaratívan, ellenőrizz procedurálisan!**  
Azaz: miután megírtad deklaratív programodat, gondold végig azt is, hogy jó lesz-e a procedurális végrehajtása (nem esik-e végtelen ciklusba, elég hatékony-e, működésképesek-e a beépített predikátumok stb.)!

## A Prolog procedurális szemantikája

- A Prolog végrehajtási mechanizmusa többféleképpen is leírható. Különböző megadási módok:
  - Az ún. SLD rezolúciós tételbizonyítási módszer (nagyon tömören lásd alább)
  - egy cél-redukción alapuló tételbizonyítási módszer (lásd a következő fölőliákon)
  - mintaillesztésen alapuló visszalépéses eljárás-szervezés (részletesen lásd később).
- A Prologban alkalmazott rezolúciós tételbizonyítási módszerről:
  - SLD resolution: Linear resolution with a Selection function for Definite clauses.
  - A célsorozat **tagadja** a keresett dolgok létezését, pl. 'Imre'-nek nincs nagyszülője:
 

```
:- nagyszuloje('Imre', N). ≡ ¬∃N nagyszuloje('Imre', N)
```
  - A célsorozat és egy programklóz ún. rezolvenseként kapunk egy újabb célsorozatot.
  - A rezolúciós lépéseket addig ismétljük, amíg el nem jutunk az üres klózhoz (zsákutcák esetén visszalépést alkalmazva).
  - Ha ez sikerül, akkor ezzel **indirekt** módon beláttuk, hogy a célsorozat törzse következik a programból, hiszen a törzs negáltjából és a programból következik az azonosan hamis □.
  - A rezolúciós bizonyítás konstruktív, siker esetén behelyettesíti a célsorozat változóit — ez a keresett válasz (pl. N = 'Géza').
  - További válaszok alternatív bizonyításokkal állíthatók elő.

## A Prolog mint cél-redukciós tételbizonyító

- Alapgondolat: a megoldandó célt redukáljuk (visszavezetjük) olyan részcélokra, amelyekből ő következik.
- Példaprogram
 

```
szuloje('Imre', 'István'). (sz1)
szuloje('Imre', 'Gizella'). (sz2)
szuloje('István', 'Géza'). (...) (sz3)

nagyszuloje(Gy, N) :- szuloje(Gy, Sz), szuloje(Sz, N). (nsz)
```
- A kezdeti célsorozat: `:- nagyszuloje('Imre', N).`  
(Most a célsorozatot úgy tekintjük mint bizonyítandó állítások sorozatát.)
- Kiegészítjük a célsorozatot egy vagy több speciális céllal, a keresett változók értékének megőrzése érdekében:
 

```
:- nagyszuloje('Imre', N), write(N).
```
- A célsorozatot ismételten **redukáljuk** (lásd következő fólia), amíg csak `write` cél marad:
 

```
[red. a (nsz) klózzal] :- szuloje('Imre', Sz), szuloje(Sz, N), write(N).
[red. a (sz1) klózzal] :- szuloje('István', N), write(N).
[red. a (sz3) klózzal] :- write('Géza').
```
- A futás eredményét a `write` argumentumból olvashatjuk ki.

## A redukciós lépés

- A példa érintett klózzai és a célsorozat:
 

```
szuloje('Imre', 'István'). (sz1)
szuloje('István', 'Géza'). (sz3)
nagyszuloje(Gy, N) :- szuloje(Gy, Sz), szuloje(Sz, N). (nsz)
:- nagyszuloje('Imre', N), write(N).
```
- Redukciós lépés: egy célsorozat + egy rá vonatkozó klóz  $\Rightarrow$  új célsorozat.
- A redukciós lépést a vonatkozó predikátum **minden** klózára sorra megkíséreljük:
  - A célsorozat **első** elemét a klóz fejével azonos alakra hozzuk, változók behelyettesítésével.
  - Mind a klózt, mind a célsorozatot **specializáljuk** a kívánt behelyettesítések elvégzésével. A példában előállítjuk (nsz) speciális esetét:
 

```
nagyszuloje('Imre', N) :- szuloje('Imre', Sz), szuloje(Sz, N). (nsz*)
```
  - Az első célt helyettesítjük a klóz törzsével, azaz ezt a célt egy előfeltételére redukáljuk. A példában az új célsorozat: `szuloje('Imre', Sz), szuloje(Sz, N), write(N).`
- A következő lépésben az (sz1) klózzal redukálunk, a **célsorozatot** specializálva az `Sz = 'István'` behelyettesítéssel: `szuloje('István', N), write(N).`  
Mivel tényállítással redukálunk, üres törzset helyettesítünk, így a célsorozat hossza csökken.
- A (sz3) ténnyel való hasonló redukciós lépés eredménye: `write('Géza').`

## Redukciós lépés — további részletek

- Változók kezelése
  - A változók hatásköre egy klózra terjed ki (vö.  $\forall X_1 \dots X_j (F \leftarrow T)$ ).
  - A redukciós lépés előtt a klózt le kell másolni, a változókat szisztematikusan újakra cserélve (vö. rekurzíó).
- **Egyesítés:** két kifejezés/állítás azonos alakra hozása, változók behelyettesítésével.
  - A változókat tetszőleges kifejezéssel lehet helyettesíteni, akár más változóval is.
  - Az egyesítés a **legáltalánosabb** közös alakot állítja elő. Pl.
 

|                                   |              |                                              |
|-----------------------------------|--------------|----------------------------------------------|
| <code>sum_tree(leaf(X), X)</code> | közös alakja | <code>sum_tree(leaf(X), X)</code> és nem pl. |
| <code>sum_tree(T, V)</code>       |              | <code>sum_tree(leaf(0), 0)</code>            |
  - Az egyesítés eredménye a legáltalánosabb közös alakot előállító behelyettesítés. Ez változó-átnevezéstől eltekintve egyértelmű. A példában: `T=leaf(X), V=X.`
  - Példák:
 

|                                     |                                        |                                           |
|-------------------------------------|----------------------------------------|-------------------------------------------|
| <i>Hívás:</i>                       | <i>Fej:</i>                            | <i>Behelyettesítés:</i>                   |
| <code>nagyszuloje('Imre', N)</code> | <code>nagyszuloje(Gy, NSz)</code>      | <code>Gy = 'Imre', NSz = N</code>         |
| <code>szuloje('Imre', Sz)</code>    | <code>szuloje('Imre', 'István')</code> | <code>Sz = 'István'</code>                |
| <code>szuloje('Imre', Sz)</code>    | <code>szuloje('István', 'Géza')</code> | <i>nem egyesíthető</i>                    |
| <code>szereti('István', Kit)</code> | <code>szereti(X, X)</code>             | <code>X = 'István', Kit = 'István'</code> |
| <code>szereti(Ki, Kit)</code>       | <code>szereti(X, X)</code>             | <code>X = Ki, Kit = Ki</code>             |

## Választási pontok, visszalépés

- A példában „szerencsénk” volt, a redukciós lépések sorozata elvezetett egy megoldáshoz.
- Az általános esetben zsákutcába, egy nem redukálható célsorozathoz is juthatunk, pl.
 

```
:- nagyszuloje('Imre', 'Civakodó Henrik'). (nsz)
:- szuloje('Imre', Sz), szuloje(Sz, 'Civakodó Henrik'). (sz1): szuloje('Imre', 'István')
:- szuloje('István', 'Civakodó Henrik'). ???
```
- A 2. célsorozatot az (sz1) klózzal redukáltuk, de a megoldáshoz az (sz2): `szuloje('Imre', 'Gizella')` vezet — nem csak az első egyesíthető klózfejet kell kezelnünk, hanem az összeset!
- Ha nem az utolsó klózzal redukálunk, akkor létrehozunk egy **választási pontot**, ebben elmentjük a célsorozatot és azt, hogy melyik klózzal redukáltuk.
- **Zsákutca**, vagy **új megoldás** kérése esetén visszatérünk a legutóbbi (legfiatalabb) választási ponthoz és ott a **fennmaradó** (még ki nem próbált) klózok között folytatjuk a keresést.
- Ha egy választási pontnál nem találunk újabb klózt, újabb visszalépés következik. Ha nincs választási pont ahova visszaléphetnénk, akkor a célsorozat futása meghiúsul.
- A fenti példában: visszatérünk a második lépéshez, és ott az (sz2) klózzal próbálkozunk:
 

```
(...)
:- szuloje('Imre', Sz), szuloje(Sz, 'Civakodó Henrik'). (sz1)
:- szuloje('Gizella', 'Civakodó Henrik'). (sz5)
□
```

## Visszalépéses keresés szemléltetése keresési fával

```

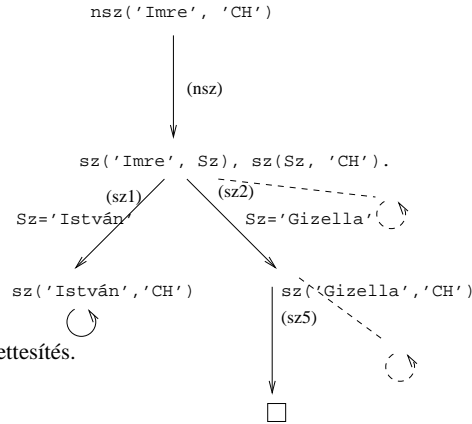
sz('Imre', 'István'). % (sz1)
sz('Imre', 'Gizella'). % (sz2)
sz('István', 'Géza'). % (sz3)
sz('István', 'Sarolt'). % (sz4)
sz('Gizella', 'CH'). % (sz5)
sz('Gizella', 'BG'). % (sz6)

```

```

nsz(Gy, N) :-
 sz(Gy, Sz), sz(Sz, N). % (nsz)

```



● A keresési fa

- csomópontjai a végrehajtási állapotok
- címkék:

- csomópontokban: célsorozatok,
- éleken: a kiválasztott klóz és a behelyettesítés.

● A Prolog keresés: a keresési fa bejárása

- balról jobbra,
- mélységi (depth-first) kereséssel.

- A szaggatott vonalak sikertelen klózkérésre utalnak, az ún. első argumentum szerinti indexelés a felsőt kiküszöböli.

## A keresési tér bejárásának nyomkövetése

- Egy (szerkesztett) párbeszéd a reduktions nyomkövetővel, a meghíúsuló egyesítéseket elhagytuk.

```

|| ?- nagyszuloje('Imre', 'Civakodó Henrik').
G0: nagyszuloje('Imre', 'Civakodó Henrik') ?
 <--- ujsor leütésére folytatja
 Trying clause 1 of nagyszuloje/2 ... successful
(1) {Gyerek_1 = 'Imre', Nagyszulo_1 = 'Civakodó Henrik'}<--- változó-átnevezés

G1: szuloje('Imre', Szulo_1), szuloje(Szulo_1, 'Civakodó Henrik') ?
 Trying clause 1 of szuloje/2 ... successful
(1) {Szulo_1 = 'István'}

----G2: szuloje('István', 'Civakodó Henrik') ?
(...)
|<<<< Failing back to goal G1
 <--- G3-G8 6 sikertelen klózzillesztés
 <--- Van-e másik szuloje 'Imre'-nek?
 Trying clause 2 of szuloje/2 ... successful
(2) {Szulo_1 = 'Gizella'}

----G9: szuloje('Gizella', 'Civakodó Henrik') ?
 Trying clause 5 of szuloje/2 ... successful
(5) {}
----G14: [] ?
 <--- üres klóz, siker
 |+++ Solution: ?
 |<<<< Failing back to goal G1
 <--- az előző főlián alsó szaggatott

|<<<< No more choices
 <--- az előző főlián felső szaggatott

```

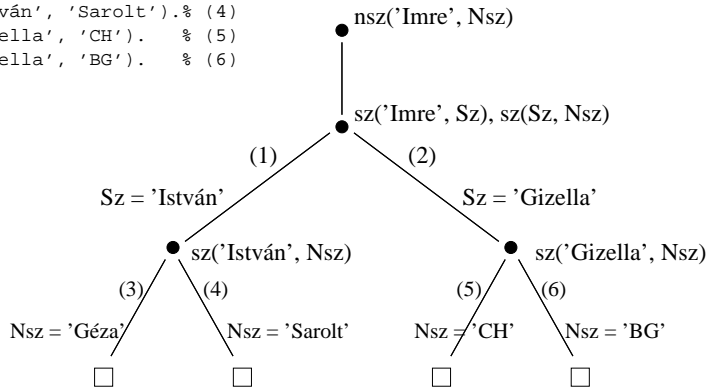
## Keresési fa — újabb példa

```

sz('Imre', 'István'). % (1)
sz('Imre', 'Gizella'). % (2)
sz('István', 'Géza'). % (3)
sz('István', 'Sarolt'). % (4)
sz('Gizella', 'CH'). % (5)
sz('Gizella', 'BG'). % (6)

nsz(Gy, N) :-
 sz(Gy, Sz), sz(Sz, N).

```



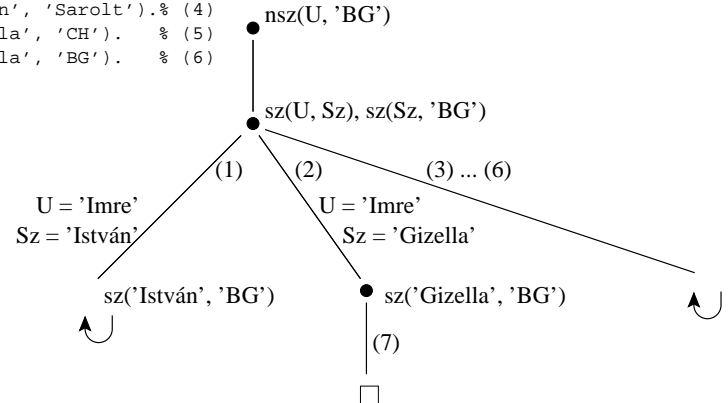
## Keresési fa — még újabb példa

```

sz('Imre', 'István'). % (1)
sz('Imre', 'Gizella'). % (2)
sz('István', 'Géza'). % (3)
sz('István', 'Sarolt'). % (4)
sz('Gizella', 'CH'). % (5)
sz('Gizella', 'BG'). % (6)

nsz(U, 'BG') :-
 sz(U, Sz), sz(Sz, 'BG').

```



## A PROLOG ELJÁRÁSOS MODELLJEI

### A Prolog végrehajtás eljárásos modelljei

- Az azonos funktorú klózek alkotnak egy eljárást
- Egy eljárás meghívása a hívás és klózfej mintaillesztésével (egyesítésével) történik
- A végrehajtás lépéseinek modellezése:
  - Eljárás-redukciós modell
    - Lényegében ugyanaz mint a cél-redukciós modell.
    - Az alaplépés: egy hívás-sorozat (azaz célsorozat) redukálása egy klóz segítségével (ez a már ismert redukciós lépés).
    - Visszalépés: visszatérünk egy korábbi célsorozathoz, és újabb klózzal próbálkozunk.
    - A modell előnyei: pontosan definiálható, a keresési tér szemléltethető
  - Eljárás-doboz modell
    - Az alapgondolat: egymásba skatulyázott eljárás-dobozok kapuin lépünk be és ki.
    - Egy eljárás-doboz kapui: hívás (belépés), sikeres kilépés, sikertelen kilépés.
    - Visszalépés: új megoldást kérünk egy már lefutott eljárástól (újra kapu).
    - A modell előnyei: közel van a hagyományos rekurzív eljárásmodellhez, a Prolog beépített nyomkövetője is ezen alapul.

LP-79

### A eljárás-redukciós végrehajtási modell

- A redukciós végrehajtási modell alapgondolata
  - A végrehajtás egy állapota: egy célsorozat
  - A végrehajtás kétféle lépésből áll:
    - redukciós lépés: egy célsorozat + klóz  $\rightarrow$  új célsorozat
    - zsákutca esetén visszalépés: visszatérés a legutolsó választási ponthoz
  - Választási pont:
    - létrehozása: olyan redukciós lépés amely nem a legutolsó klózzal illesztett
    - aktiválása: visszalépéskor visszatérünk a választási pont célsorozatához és a **további** klózek között keresünk illeszthetőt (Emiatt a választási pontban a célsorozat mellett az illesztett klóz sorszámát is tárolni kell.)
    - az ún. indexelés segít a választási pontok számának csökkentésében
- A redukciós modell keresési fával szemléltethető
  - A végrehajtás során a fa csomópontjait járjuk be mélységi kereséssel
  - A fa gyökerétől egy adott pontig terjedő szakaszon kell a választási pontokat megjegyezni — ez a választási verem (choice point stack)

LP-80

### A redukciós modell alapeleme: redukciós lépés

- Redukciós lépés: egy célsorozat redukálása egy újabb célsorozattá
  - egy programklóz segítségével (az első cél felhasználói eljárást hív):
    - A klózt **lemásoljuk**, minden változót szisztematikusan új változóra cserélve.
    - A célsorozatot szétbontjuk az első hívásra és a maradékra.
    - Az első hívást **egyesítjük** a klózfejjel
    - A szükséges behelyettesítéseket elvégezzük a klóz **törzsen** és a **célsorozat** maradékán is
    - Az új célsorozat: a klóztörzs és utána a maradék célsorozat
    - Ha a hívás és a klózfej nem egyesíthető, akkor a redukciós lépés meghiúsul.
  - egy beépített eljárás segítségével (az első cél beépített eljárást hív):
    - A célsorozatot szétbontjuk az első hívásra és a maradékra.
    - A beépített eljárás hívást végrehajtjuk.
    - Ez lehet sikeres (változó-behelyettesítésekkel), vagy lehet sikertelen.
    - Siker esetén a behelyettesítéseket elvégezzük a célsorozat maradékán.
    - Az új célsorozat: az (első hívás elhagyása után fennmaradó) maradék célsorozat.
    - Ha a beépített eljárás hívása sikertelen, akkor a redukciós lépés meghiúsul.

## A Prolog végrehajtási algoritmus

- (Kezdeti beállítások:)* A verem üres,  $CS := c\acute{e}lsorozat$
- (Beépített eljárások:)* Ha  $CS$  első hívása beépített akkor hajtjuk végre,
  - Ha sikertelen  $\Rightarrow$  6. lépés.
  - Ha sikeres,  $CS := a$  redukciós lépés eredménye  $\Rightarrow$  5. lépés.
- (Klőzszámítóló kezdőértékezése:)*  $I = 1$ .
- (Redukciós lépés:)* Tekintsük  $CS$  első hívására vonatkozható klőzok listáját. Ez indexelés nélkül a predikátum összes klőza lesz, indexelés esetén ennek egy megszárt részsorozata. Tegyük fel, hogy ez a lista  $N$  elemű.
  - Ha  $I > N \Rightarrow$  6. lépés.
  - Redukciós lépés a lista  $I$ -edik klőza és a  $CS$  célsorozat között.
  - Ha sikertelen, akkor  $I := I+1 \Rightarrow$  4. lépés.
  - Ha  $I < N$  (nem utolsó), akkor veremljük  $\langle CS, I \rangle$ -t.
  - $CS := a$  redukciós lépés eredménye
- (Siker:)* Ha  $CS$  üres, akkor sikeres vég, egyébként  $\Rightarrow$  2. lépés.
- (Sikertelenség:)* Ha a verem üres, akkor sikertelen vég.
- (Visszalépés:)* Ha a verem nem üres, akkor leemeljük a veremből  $\langle CS, I \rangle$ -t,  $I := I+1$ , és  $\Rightarrow$  4. lépés.

## Indexelés (előzetes)

- Mi az indexelés?
  - egy hívásra vonatkozható (potenciálisan illeszthető) klőzok gyors kiválasztása,
  - egy eljárás klőzainak **fordítási idejű** csoportosításával.
- A legtöbb Prolog rendszer, így a SICStus Prolog is, az első fej-argumentum alapján indexel (first argument indexing).
- Az indexelés alapja az első fejargumentum külső funkora:
  - $C$  szám vagy névkonstans esetén  $C/0$ ;
  - $R$  nevű és  $N$  argumentumú struktúra esetén  $R/N$ ;
  - változó esetén nem értelmezett (minden funktohoz besoroltatik).
- Az indexelés megvalósítása:
  - Fordítási időben minden funktohoz elkészítjük az alkalmazható klőzok listáját
  - Futáskor lényegében konstans idő alatt elő tudjuk venni a megfelelő klőzlistát
  - Fontos:* ha egyelemű a részhalmaz, nem hozunk létre választási pontot!
- Például  $szuloje('István', X)$  kételemű klőzlistára szűkít, de  $szuloje(X, 'István')$  mind a 6 klőzt megtartja (mert a SICStus Prolog csak az első argumentum szerint indexel)

## Redukciós modell — előnyök és hátrányok

- Előnyök
  - (viszonylag) egyszerű és (viszonylag) precíz definíció
  - a keresési tér megjeleníthető, grafikus szemléltethető
- Hátrányok
  - az eljárásokból való kilépést elfedi, pl.
 

|              |                 |                                        |
|--------------|-----------------|----------------------------------------|
| $p :- q, r.$ | $G0: p ?$       |                                        |
| $q :- s, t.$ | $G1: q, r ?$    |                                        |
| $s.$         | $G2: s, t, r ?$ |                                        |
| $t.$         | $G3: t, r ?$    |                                        |
| $r.$         | $G4: r ?$       | $\Leftarrow q\text{-ből való kilépés}$ |
|              | $G5: [] ?$      |                                        |
  - nem jól illeszkedik a Prolog megvalósítások tényleges végrehajtási mechanizmusához
  - nem alkalmazható „igazi” Prolog programok nyomkövetésére (hosszú célsorozatok)
- Ezért van létjogosultsága egy másik modellnek:
  - eljárás-doboz (procedure box) modell
  - (szokás még 4-kapus doboz ill. Byrd doboz modellnek is nevezni)
  - a Prolog rendszerek nyomkövető szolgáltatása erre a modellre épít

## Az eljárás-doboz modell

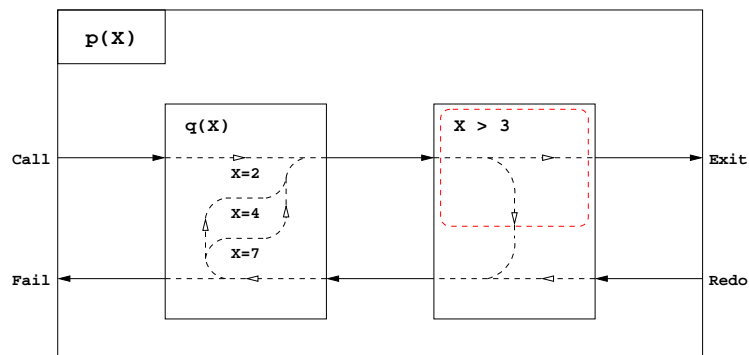
- A Prolog eljárás-végrehajtás két fázisa
  - előre menő végrehajtás: egymásba skatulyázott eljárás-belépések és -kilépések
  - visszafelé menő végrehajtás: újabb megoldás kérése egy már lefutott eljárástól
- Egy egyszerű példa
 
$$q(2). \quad q(4). \quad q(7). \quad p(X) :- q(X), X > 3.$$
  - Belépünk a  $p/1$  eljárásba (Hívási kapu, Call port)
  - Belépünk a  $q/1$  eljárásba (Call)
  - A  $q/1$  eljárás sikeresen lefut a  $q(2)$  eredménnyel (Kilépési kapu, Exit port)
  - A  $> /2$  eljárásba belépünk a  $2 > 3$  hívással (Call)
  - A  $> /2$  eljárás sikertelenül fut le (Meghiúsulási kapu, Fail port)
  - (visszafelé menő futás): visszatérünk (a már lefutott)  $q/1$ -be, újabb megoldást kérve (Újra kapu, Redo Port)
  - A  $q/1$  eljárás sikeresen lefut a  $q(4)$  eredménnyel (Exit)
  - A  $4 > 3$  eljárás hívással a  $> /2$ -be belépünk majd sikeresen kilépünk (Call, Exit)
  - A  $p/1$  eljárás sikeresen lefut  $p(4)$  eredménnyel (Exit)



## Eljárás-doboz modell — grafikus szemléltetés

q(2). q(4). q(7).

p(X) :- q(X), X > 3.



## Eljárás-doboz modell — egyszerű nyomkövetési példa

## Az előző példa nyomkövetése SICStus Prologban

q(2). q(4). q(7).

p(X) :- q(X), X > 3.

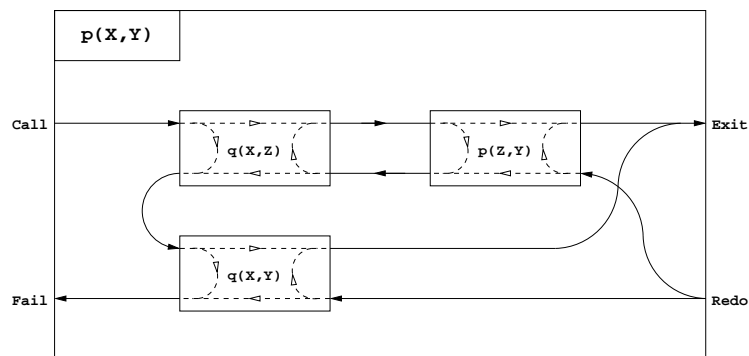
```
| ?- trace, p(X).
 1 1 Call: p(_463) ?
 2 2 Call: q(_463) ?
? 2 2 Exit: q(2) ? % ? ≡ nondeterminisztikus
kilépés
 3 2 Call: 2>3 ?
 3 2 Fail: 2>3 ?
? 2 2 Redo: q(2) ? % visszafelé menő végrehajtás
 2 2 Exit: q(4) ?
? 4 2 Call: 4>3 ?
 4 2 Exit: 4>3 ?
? 1 1 Exit: p(4) ?
X = 4 ? ;
 1 1 Redo: p(4) ? % visszafelé menő végrehajtás
 2 2 Redo: q(4) ? % visszafelé menő végrehajtás
 2 2 Exit: q(7) ?
 5 2 Call: 7>3 ?
 5 2 Exit: 7>3 ?
 1 1 Exit: p(7) ?
X = 7 ? ;
no
```

## Eljárás-doboz: egy összetettebb példa

p(X,Y) :- q(X,Z), p(Z,Y).

p(X,Y) :- q(X,Y).

q(1,2). q(2,3). q(2,4).



## Eljárás-doboz modell — „kapcsolási” alapelvek

- Hogyan építhető fel egy „szülő” eljárás doboza a benne hívott eljárások dobozaiból?
- Feltehető, hogy a klózfejekben (különböző) változók vannak, a fej-egyesítéseket hívás(okk)á alakítva
- Előre menő végrehajtás:
  - A szülő Hívás kapuját az első klóz első hívásának Hívás kapujára kötjük.
  - Egy rész-eljárás Kilépési kapuját
    - a következő hívás Hívás kapujára, vagy,
    - ha nincs következő hívás, akkor a szülő Kilépési kapujára kötjük
- Visszafelé menő végrehajtás:
  - Egy rész-eljárás Meghiúsulási kapuját
    - az előző hívás Újra kapujára, vagy,
    - ha nincs előző hívás, akkor a következő klóz első hívásának Hívás kapujára, vagy
    - ha nincs következő klóz, akkor a szülő Meghiúsulási kapujára kötjük
  - A szülő Újra kapuját mindegyik klóz utolsó hívásának Újra kapujára kötjük
    - mindig arra a klózra térünk vissza, amelyben legutoljára volt a vezérlés

## Eljárás-doboz modell — OO szemléletben

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

## OO szemléletű dobozok: p/2 „következő megoldás” metódusának C++ kódja

```

boolean p::next()
{ switch(clno) {
 case 0: // entry point for the Call port
 clno = 1; // enter clause 1: p(X,Y) :- q(X,Z), p(Z,Y).
 qptr = new q(x, &z); // create a new instance of subgoal q(X,Z)
 redoll:
 if(!qptr->next()) { // if q(X,Z) fails
 delete qptr; // destroy it,
 goto cl2; // and continue with clause 2 of p/2
 }
 pptr = new p(z, py); // otherwise, create a new instance of subgoal p(Z,Y)
 case 1: // (enter here for Redo port if clno==1)
 /* redo12: */
 if(!pptr->next()) { // if p(Z,Y) fails
 delete pptr; // destroy it,
 goto redoll; // and continue at redo port of q(X,Z)
 }
 return TRUE; // otherwise, exit via the Exit port
 cl2:
 clno = 2; // enter clause 2: p(X,Y) :- q(X,Y).
 qbptr = new q(x, py); // create a new instance of subgoal q(X,Y)
 case 2: // (enter here for Redo port if clno==1)
 /* redo21: */
 if(!qbptr->next()) { // if q(X,Y) fails
 delete qbptr; // destroy it,
 return FALSE; // and exit via the Fail port
 }
 return TRUE; // otherwise, exit via the Exit port
 } }

```

## Visszalépéses keresés — egy aritmetikai példa

- Példa: „jó” számok keresése
- A feladat: keressük meg azokat a kétjegyű számokat amelyek négyzete háromjegyű és a szám fordítottjával kezdődik
- A program:

```

% decl(J): J egy pozitív decimális számjegy.
decl(1). decl(2). decl(3). decl(4).
decl(5). decl(6). decl(7). decl(8). decl(9).

```

```

% dec(J): J egy decimális számjegy.
dec(0).
dec(J) :- decl(J).

```

```

% Szam négyzete háromjegyű és a Szam fordítottjával kezdődik.
joszam(Szam):-
 decl(A), dec(B),
 Szam is A * 10 + B, Szam * Szam // 10 == B * 10 + A.

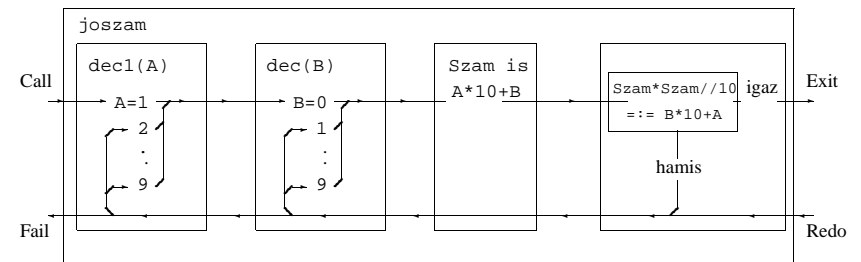
```

## Prolog végrehajtás — a 4-kapus doboz modell

```

joszam(Szam):-
 decl(A), dec(B),
 Szam is A * 10 + B, Szam * Szam // 10 == B * 10 + A.

```



## Visszalépéses keresés — számintervallum felsorolása

- `dec(J)` felsorolta a 0 és 9 közötti egész számokat
- Általánosítás: soroljuk fel az  $N$  és  $M$  közötti egészeket ( $N$  és  $M$  maguk is egészek)

```
% between(M, N, I): M =< I =< N, I egész.
between(M, N, M) :-
 M =< N.
between(M, N, I) :-
 M < N,
 M1 is M+1,
 between(M1, N, I).
```

```
% dec(X): X egy decimális számjegy
dec(X) :- between(0, 9, X).
```

```
| ?- between(1, 2, _X), between(3, 4, _Y), Z is 10*_X+_Y.
Z = 13 ? ;
Z = 14 ? ;
Z = 23 ? ;
Z = 24 ? ;
no
```

## A SICStus eljárás-doboz alapú nyomkövetése — legfontosabb parancsok

- Alapvető nyomkövetési parancsok

- `h <RET> (help)` — parancsok listázása
- `c <RET> (creep)` vagy `<RET>` — továbblépés minden kapunál megálló nyomkövetéssel
- `l <RET> (leap)` — csak töréspontnál áll meg, de a dobozokat építi
- `z <RET> (zip)` — csak töréspontnál áll meg, dobozokat nem épít
- `+ <RET>` ill. `- <RET>` — töréspont rakása/eltávolítása a kurrens predikátumra
- `s <RET> (skip)` — eljárástörzs átlépése (`Call/Redo`  $\Rightarrow$  `Exit/Fail`)
- `o <RET> (out)` — kilépés az eljárástörzsből

- A Prolog végrehajtást megváltoztató parancsok

- `u <RET> (unify)` — a kurrens hívást végrehajtás helyett egyesíti egy beolvasott kifejezéssel.
- `r <RET> (retry)` — újakezdi a kurrens hívás végrehajtását (ugrás a `Call` kapura)

- Információ-megjelenítő és egyéb parancsok

- `w <RET> (write)` — a hívás kiírása mélység-korlátozás nélkül
- `b <RET> (break)` — új, beágyazott Prolog interakciós szint létrehozása
- `n <RET> (notrace)` — nyomkövető kikapcsolása
- `a <RET> (abort)` — a kurrens futás abbahagyása

## TOVÁBBI VEZÉRLÉSI SZERKEZETEK

### Diszjunkció, példa: az „őse” predikátum

- Az „őse” reláció a „szülője” reláció tranzitív lezártja: a szülő ős (1), és az ős őse is ős (2), azaz:

```
% ose0(E, Os): E ose Os.
ose0(E, Sz) :- szuloje(E, Sz). % (1)
ose0(E, Os) :- ose0(E, Os0), ose0(Os0, Os). % (2)
```

- Az `ose0` definíciója matematikailag helyes, de végtelen Prolog keresési teret ad:

```
szuloje(gyerek,apa). szuloje(gyerek,anya). szuloje(anya,nagyapa).

| ?- ose0(gyerek, Os).
 Os = apa ? ; Os = anya ? ; {néhány másodperc után:}
 ! Resource error: insufficient memory
```

- A végtelen rekurzió oka: Az `:- ose0(apa, X)`. cél esetén az (1) klóz megghiúsul, (2) pedig egy `:- ose0(apa, Y), ose0(Y, X)`. célsorozathoz vezet stb.

- A balrekurziót kiküszöbölve kapjuk:

```
ose1(E, Sz) :- szuloje(E, Sz). % (3)
ose1(E, Os) :- szuloje(E, Sz), ose1(Sz, Os). % (4)

| ?- ose1(gyerek, Os).
Os = apa ? ; Os = anya ? ; Os = nagyapa ? ; no
```

- Ez minden `szuloje(X, Y)` részcélt kétszer hajt végre: (3)-ban és (4)-ben.

## A diszjunkció

- Az `ose1` predikátum hatékonyabbá tehető klózai összevonásával:

```
ose2(E, Os) :- szuloje(E, Sz), maga_vagy_ose(Sz, Os).
```

```
maga_vagy_ose(E, E). (1)
```

```
maga_vagy_ose(E, Os) :- ose2(E, Os).
```

- A `maga_vagy_ose` predikátum egy ún. **diszjunkció** bevezetésével kiküszöbölhető:

```
ose3(E, Os) :-
 szuloje(E, Sz),
 (Os = Sz
 ; ose3(Sz, Os)
).
```

- A SICStus Prolog ténylegesen úgy implementálja a fenti diszjunkciót, hogy bevezet egy `maga_vagy_ose`-vel azonos segéd-predikátumot és az `ose3` klózt `ose2`-vé alakítja.
- (Ismétlés:) Az  $X=Y$  beépített predikátum a két argumentumát egyesíti.
- Az  $=/2$  eljárás egy tényállítással definiálható:  $U = V \equiv (U, U), \text{vö. (1)}$ .

## A diszjunkció mint szintaktikus édesítőszert

- A diszjunkció akárhány tagú lehet. A ‘;’ művelet gyengébben köt mint a ‘,’ ezért a diszjunkciót mindig zárójelbe tesszük, míg az ágait nem kell zárójellezni. Példa, „szabványos” formázással:

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

- A diszjunkció egy segéd-predikátummal mindig kiküszöbölhető

- Megkeressük azokat a változókat, amelyek a diszjunkcióban és azon kívül is előfordulnak
- A segéd-predikátumnak ezek a változók lesznek az argumentumai
- A segéd-predikátum minden klóza megfelel a diszjunkció egy ágának

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

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

- A diszjunkció szemantikáját ezzel a segéd-predikátumos átalakítással definiáljuk.

## Diszjunkció — megjegyzések

- Az egyes klózok ‘ÉS’ vagy ‘VAGY’ kapcsolatban vannak?

- A program klózai **ÉS** kapcsolatban vannak, pl.

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

jelentése: Imre szülője István **ÉS** Imre szülője Gizella.

- Az **ÉS** kapcsolatban levő klózok alternatív (VAGY kapcsolatban levő) válaszokhoz vezetnek:

```
:- szuloje('Imre', Sz). => Sz = 'István' ? ; Sz = 'Gizella' ? ; no
```

A „Ki Imre szülője?” kérdésre a válasz: István vagy Gizella.

- A fenti két klózos predikátum átalakítható egyetlen klózzá, diszjunkció segítségével:

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

A konjunkció ezáltal diszjunkcióvá alakult (vö. De Morgan azonosságok).

- Általánosan: tetszőleges predikátum egyklózosá alakítható:

- a klózokat átalakítjuk azonos fejlűvé, új változók és egyenlőségek bevezetésével:

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

- a klóztörzseket egy diszjunkcióvá fogjuk össze, amely az új predikátum törzse (lásd (\*)).

## Negáció

- Feladat: Keressünk (adatbázisunkban) egy olyan szülőt, aki **nem** nagyszülő!

- Ehhez negációra van szükségünk:

- Meghiúsulásos negáció: a `\+` hívás szerkezet lefuttatja hívást, és pontosan akkor sikerül, ha a hívás meghiúsult.

- Egy megoldás:

```
| ?- szuloje(_, X), \+ nagyszuloje(_, X).
X = 'István' ? ;
X = 'Gizella' ? ;
no
```

- Egy ekvivalens megoldás:

```
| ?- szuloje(_Gy, X), \+ szuloje(_, _Gy).
X = 'István' ? ;
X = 'Gizella' ? ;
no
```

- Mi történik ha a két hívást megcseréljük?

```
| ?- \+ szuloje(_, _Gy), szuloje(_Gy, X).
no
```

## A meghíúsulás negáció (NF — Negation by Failure)

- $A \setminus +$  hívás beépített meta-eljárás (vö.  $\not\vdash$  — nem bizonyítható)
  - végrehajtja a hívás hívást,
  - ha hívás sikeresen lefutott, akkor meghíúsul,
  - egyébként (azaz ha hívás meghíúsult) sikerül.
- $\setminus +$  hívás futása során hívás legfeljebb egy megoldása áll elő
- $\setminus +$  hívás sohasem helyettesít be változót
- Gondok a meghíúsulás negációval:
  - „zárt világ feltételezése” (CWA) — ami nem bizonyítható, az nem igaz.
 

```
| ?- \+ szuloje('Imre', X). ----> no
| ?- \+ szuloje('Géza', X). ----> true ?
```
  - $\setminus + H$  deklaratív szemantikája:  $\neg\exists X(H)$ , ahol  $X$  a  $H$ -ban a hívás pillanatában behelyettesítetlen változókat jelöli.
 

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

## Példa: együtttható meghatározása lineáris kifejezésben

- Formula: számokból és az 'x' névkonstansból '+' és '\*' operátorokkal épül fel.
  - `% :- type kif == {x} \ number \ {kif+kif} \ {kif*kif}.`
  - Lineáris formula: a '\*' operátor legalább egyik oldalán szám áll.
 

```
% egyhat(Kif, E): A Kif lineáris formulában az x együttthatója E.
egyhat(x, 1).
egyhat(Kif, E) :-
 number(Kif), E = 0.
egyhat(K1+K2, E) :-
 egyhat(K1, E1),
 egyhat(K2, E2),
 E is E1+E2.
egyhat(K1*K2, E) :-
 number(K1),
 egyhat(K2, E0),
 E is K1*E0.
egyhat(K1*K2, E) :-
 number(K2),
 egyhat(K1, E0),
 E is K2*E0.
```
- ```
| ?- egyhat(((x+1)*3)+x+2*(x+x+3), E). | ?- egyhat(2*3+x, E).
E = 8 ? ;                                E = 1 ? ;
no                                         E = 1 ? ; no
```

Együtttható meghatározása: többszörös megoldások kiküszöbölése

- negáció alkalmazásával:


```
(...)
egyhat(K1*K2, E) :-
    number(K1), egyhat(K2, E0), E is K1*E0.
egyhat(K1*K2, E) :-
    \+ number(K1),
    number(K2), egyhat(K1, E0), E is K2*E0.
```
- hatékonyabban, feltételes kifejezéssel:


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

Feltételes kifejezések

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


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


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

Feltételes kifejezések (folyt.)

● Procedurális szemantika

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

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

● Többszörös elágaztatás skatulyázott feltételes kifejezésekkel:

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

● Az egyébként rész elhagyható, alapértelmezése: fail.

● A \+ felt negáció kiváltható a (felt -> fail ; true) feltételes kifejezéssel.

Feltételes kifejezés — példák

● Faktoriális

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

● Jelentése azonos a sima diszjunkciós alakkal (lásd komment), de annál hatékonyabb, mert nem hagy maga után választási pontot.

● Szám előjele

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

A Prolog adatfogalma, a Prolog kifejezés (ismétlés, rendszerezés)

● egyszerű adatok:

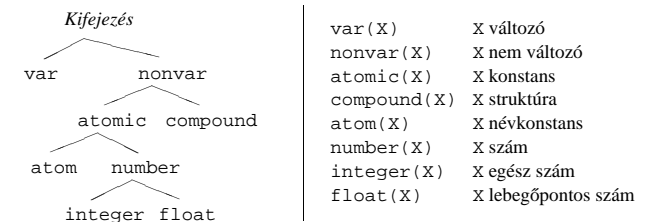
- konstansok
 - egész számok (gyakorlatilag végtelen méretűek)
 - lebegőpontos számok
 - névkonstansok (SICStus Prologban max 65535 karakteresek)
- változók

● összetett adatok:

- struktúra-kifejezés: $\langle \text{struktúranév} \rangle (\langle \text{arg}_1 \rangle, \dots, \langle \text{arg}_n \rangle)$
- $\langle \text{struktúranév} \rangle$ egy tetszőleges névkonstans
- $\langle \text{arg}_i \rangle$ tetszőleges kifejezés
- Az argumentumok száma, n , 1 és 255 közé eshet SICStus Prologban
- Az argumentumszámot *aritás*nak is hívjuk.
- A struktúra-kifejezés *funktora*: $\langle \text{struktúranév} \rangle / n$

A Prolog kifejezések

● Prolog kifejezések osztályozása — osztályozó beépített predikátumok



● Egy osztályozó predikátum az argumentuma pillanatnyi állapotát ellenőrzi, logikailag nem tisztá:

```
| ?- X = 1, integer(X).           ==> yes
| ?- integer(X), X = 1.           ==> no
| ?- atom('István'), atom(istvan). ==> yes
| ?- compound(leaf(X)).           ==> yes
| ?- compound(X).                 ==> no
```

A Prolog alapvető adatkezelő művelete: az egyesítés

- **Egyesítés (unification):** két Prolog kifejezés (pl. egy eljáráshívás és egy klózfaj) azonos alakra hozása, változók esetleges behelyettesítésével.
- **Példák**
 - **Bemenő paraméterátadás** — a fej változóit helyettesíti be:
 - hívás: `nagyszuloje('Imre', Nsz)`,
 - fej: `nagyszuloje(Gy, N)`,
 - behelyettesítés: `Gy = 'Imre', N = Nsz`
 - **Kimenő paraméterátadás** — a hívás változóit helyettesíti be:
 - hívás: `suloje('Imre', Sz)`,
 - fej: `suloje('Imre', 'István')`,
 - behelyettesítés: `Sz = 'István'`
 - **Bemenő/kimenő paraméterátadás** — a fej és a hívás változóit is behelyettesíti:
 - hívás: `sum_tree(leaf(5), Sum)`
 - fej: `sum_tree(leaf(V), V)`
 - behelyettesítés: `V = 5, Sum = 5`

Egyesítés: változók behelyettesítése

- **A behelyettesítés fogalma**
 - A behelyettesítés egy olyan függvény, amely bizonyos változókhoz kifejezéseket rendel.
 - Példa: $\sigma = \{X \leftarrow a, Y \leftarrow s(b, B), Z \leftarrow C\}$. Itt $Dom(\sigma) = \{X, Y, Z\}$
 - A σ behelyettesítés x -hez a -t, y -hoz $s(b, B)$ -t z -hez C -t rendel. Jelölés: $X\sigma = a$ stb.
 - A behelyettesítés-függvény természetes módon kiterjeszthető az összes kifejezésre:
 - $K\sigma$: σ alkalmazása K kifejezésre: σ behelyettesítéseit *egyidejűleg* elvégezzük K -ban.
 - Példa: $f(g(Z, h), A, Y)\sigma = f(g(C, h), A, s(b, B))$
 - A σ és θ behelyettesítések kompozíciója ($\sigma \otimes \theta$) — egymás utáni alkalmazásuk
 - A $\sigma \otimes \theta$ behelyettesítés az $x \in Dom(\sigma)$ változókhoz az $(x\sigma)\theta$ kifejezést, a többi $y \in Dom(\theta) \setminus Dom(\sigma)$ változóhoz $y\theta$ -t rendel ($Dom(\sigma \otimes \theta) = Dom(\sigma) \cup Dom(\theta)$):

$$\sigma \otimes \theta = \{x \leftarrow (x\sigma)\theta \mid x \in Dom(\sigma)\} \cup \{y \leftarrow y\theta \mid y \in Dom(\theta) \setminus Dom(\sigma)\}$$
 - Pl. $\theta = \{X \leftarrow b, B \leftarrow d\}$ esetén $\sigma \otimes \theta = \{X \leftarrow a, Y \leftarrow s(b, d), Z \leftarrow C, B \leftarrow d\}$
- Egy G kifejezés **általánosabb** mint egy S , ha létezik olyan ρ behelyettesítés, hogy $S = G\rho$
 - Példa: $G = f(A, Y)$ általánosabb mint $S = f(1, s(Z))$, mert $\rho = \{A \leftarrow 1, Y \leftarrow s(Z)\}$ esetén $S = G\rho$.

Egyesítés: legáltalánosabb egyesítő

- A és B kifejezések egyesíthetők ha létezik egy olyan σ behelyettesítés, hogy $A\sigma = B\sigma$. Ezt az $A\sigma = B\sigma$ kifejezést A és B egyesített alakjának nevezzük.
- Két kifejezésnek általában több egyesített alakja lehet.
 - Példa: $A = f(X, Y)$ és $B = f(s(U), U)$ egyesített alakja pl.
 - $K_1 = f(s(a), a)$ a $\sigma_1 = \{X \leftarrow s(a), Y \leftarrow a, U \leftarrow a\}$ behelyettesítéssel
 - $K_2 = f(s(U), U)$ a $\sigma_2 = \{X \leftarrow s(U), Y \leftarrow U\}$ behelyettesítéssel
 - $K_3 = f(s(Y), Y)$ a $\sigma_3 = \{X \leftarrow s(Y), U \leftarrow Y\}$ behelyettesítéssel
- A és B legáltalánosabb egyesített alakja egy olyan C kifejezés, amely A és B minden egyesített alakjánál általánosabb
 - A fenti példában K_2 és K_3 legáltalánosabb egyesített alakok
- **Tétel:** A legáltalánosabb egyesített alak, változó-átnevezéstől eltekintve egyértelmű.
- A és B legáltalánosabb egyesítője egy olyan $\sigma = mgu(A, B)$ behelyettesítés, amelyre $A\sigma$ és $B\sigma$ a két kifejezés legáltalánosabb egyesített alakja.
 - A fenti példában σ_2 és σ_3 legáltalánosabb egyesítő.
- **Tétel:** A legáltalánosabb egyesítő, változó-átnevezéstől eltekintve egyértelmű.

Az egyesítési algoritmus

- Az egyesítési algoritmus
 - bemenete: két Prolog kifejezés: A és B
 - feladata: a két kifejezés egyesíthetőségének eldöntése
 - eredménye: sikeresség esetén a legáltalánosabb egyesítő ($mgu(A, B)$) előállítás.
- Az egyesítési algoritmus, $\sigma = mgu(A, B)$ előállítása
 1. Ha A és B azonos változók vagy konstansok, akkor $\sigma = \{\}$ (üres behelyettesítés).
 2. Egyébként, ha A változó, akkor $\sigma = \{A \leftarrow B\}$.
 3. Egyébként, ha B változó, akkor $\sigma = \{B \leftarrow A\}$.
 4. Egyébként, ha A és B azonos nevű és argumentumszámú összetett kifejezések és argumentum-listáik A_1, \dots, A_N ill. B_1, \dots, B_N , és
 - a. A_1 és B_1 legáltalánosabb egyesítője σ_1 ,
 - b. $A_2\sigma_1$ és $B_2\sigma_1$ legáltalánosabb egyesítője σ_2 ,
 - c. $A_3\sigma_1\sigma_2$ és $B_3\sigma_1\sigma_2$ legáltalánosabb egyesítője σ_3 ,
 - d. ...
 akkor $\sigma = \sigma_1 \otimes \sigma_2 \otimes \sigma_3 \otimes \dots$
 5. Minden más esetben a A és B nem egyesíthető.

Egyesítési példák

- $A = \text{sum_tree}(\text{leaf}(V), V), B = \text{sum_tree}(\text{leaf}(5), S)$
 - (4.) A és B neve és argumentumszáma megegyezik
 - (a.) $\text{mgu}(\text{leaf}(V), \text{leaf}(5))$ (4., majd 2. szerint) $= \{V \leftarrow 5\} = \sigma_1$
 - (b.) $\text{mgu}(V\sigma_1, S) = \text{mgu}(5, S)$ (3. szerint) $= \{S \leftarrow 5\} = \sigma_2$
 - tehát $\text{mgu}(A, B) = \sigma_1 \otimes \sigma_2 = \{V \leftarrow 5, S \leftarrow 5\}$
- $A = \text{node}(\text{leaf}(X), T), B = \text{node}(T, \text{leaf}(3))$
 - (4.) A és B neve és argumentumszáma megegyezik
 - (a.) $\text{mgu}(\text{leaf}(X), T)$ (3. szerint) $= \{T \leftarrow \text{leaf}(X)\} = \sigma_1$
 - (b.) $\text{mgu}(T\sigma_1, \text{leaf}(3)) = \text{mgu}(\text{leaf}(X), \text{leaf}(3))$ (4., majd 2. szerint) $= \{X \leftarrow 3\} = \sigma_2$
 - tehát $\text{mgu}(A, B) = \sigma_1 \otimes \sigma_2 = \{T \leftarrow \text{leaf}(3), X \leftarrow 3\}$

Egyesítési példák a gyakorlatban

- Az egyesítéssel kapcsolatos beépített eljárások:
 - $X = Y$ egyesíti a két argumentumát, meghiúsul, ha ez nem lehetséges.
 - $X \backslash = Y$ sikerül, ha két argumentuma nem egyesíthető, egyébként meghiúsul.

● Példák:

```
| ?- 3+(4+5) = Left+Right.
      Left = 3, Right = 4+5 ?
| ?- node(leaf(X), T) = node(T, leaf(3)).
      T = leaf(3), X = 3 ?
| ?- X*Y = 1+2*3.                % mert 1+2*3 ≡ 1+(2*3)
      no
| ?- X*Y = (1+2)*3.
      X = 1+2, Y = 3 ?
| ?- f(X, 3/Y-X, Y) = f(U, B-a, 3).
      B = 3/3, U = a, X = a, Y = 3 ?
| ?- f(f(X), U+2*2) = f(U, f(3)+Z).
      U = f(3), X = 3, Z = 2*2 ?
```

Az egyesítés kiegészítése: előfordulás-ellenőrzés (*occurs check*)

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

● Példák:

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

LISTÁK PROLOGBAN

A Prolog lista-fogalma

• A Prolog lista

- Az üres lista a [] névkonstans. A nem-üres lista '(Fej, Farok) struktúra ahol
 - Fej a lista feje (első eleme), míg
 - Farok a lista farka, azaz a fennmaradó elemekből álló lista.
- A listák írhatók egyszerűsített alakban („szintaktikus édesítés”).
- Megvalósításuk optimalizált, időben és helyben is hatékonyabb, mint a „közönséges” struktúráké.

• Példa

```
számlista.(E,L) :-
    number(E), számlista(L).
számlista([]).

| ?- listing(számlista).
számlista([A|B]) :-
    number(A),
    számlista(B).
számlista([]).

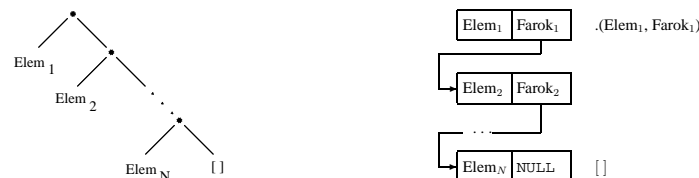
| ?- számlista([1,2]).    % [1,2] == .(1,.(2,[])) == [1|[2|[]]]
yes
| ?- számlista([1,a,f(2)]).
no
```

Listák írásmódjai

• Egy N elemű lista lehetséges írásmódjai:

- alapstruktúra-alak: $.(Elem_1, .(Elem_2, \dots, .(Elem_N, []) \dots))$
- ekvivalens lista-alak: $[Elem_1, Elem_2, \dots, Elem_N]$
- kevésbé kényelmes ekvivalens alak: $[Elem_1 | [Elem_2 | \dots | [Elem_N | []] \dots]]$

• A listák fastruktúra alakja és megvalósítása



Listák jelölése — szintaktikus édesítőszerek

- az alapvető édesítés: $[Fej | Farok] \equiv .(Fej, Farok)$
- N -szeri alkalmazás kevesebb zárójellel:

$$[Elem_1, Elem_2, \dots, Elem_N | Farok] \equiv [Elem_1 | [Elem_2 | \dots | [Elem_N | Farok] \dots]]$$
- Ha a fark []: $[Elem_1, Elem_2, \dots, Elem_N] \equiv [Elem_1, Elem_2, \dots, Elem_N | []]$
 - | ?- [1,2] = [X|Y]. $\Rightarrow X = 1, Y = [2] ?$
 - | ?- [1,2] = [X,Y]. $\Rightarrow X = 1, Y = 2 ?$
 - | ?- [1,2,3] = [X|Y]. $\Rightarrow X = 1, Y = [2,3] ?$
 - | ?- [1,2,3] = [X,Y]. $\Rightarrow no$
 - | ?- [1,2,3,4] = [X,Y|Z]. $\Rightarrow X = 1, Y = 2, Z = [3,4] ?$
 - | ?- L = [1|_], L = [_ ,2|_]. $\Rightarrow L = [1,2|_A] ?$ % nyílt végű
 - | ?- L = .(1,[2,3|[]]). $\Rightarrow L = [1,2,3] ?$
 - | ?- L = [1,2|. (3,[])]. $\Rightarrow L = [1,2,3] ?$
 - | ?- [X|[3-Y/X|Y]] = .(A, [A-B,6]). $\Rightarrow A=3, B=[6]/3, X=3, Y=[6] ?$

Tömör és minta-kifejezések, lista-minták, nyílt végű listák

- (Ismétlés:) Tömör (ground) kifejezés: változót nem tartalmazó kifejezés
- Minta: egy általában nem nem tömör kifejezés, mindazon kifejezéseket „képviseli”, amelyek belőle változó-behelyettesítéssel előállnak.
- Lista-minta: listát (is) képviselő minta.
- Nyílt végű lista: olyan lista-minta, amely bármilyen hosszú listát is képvisel.
- Zárt végű lista: olyan lista(-minta), amely egyféle hosszú listát képvisel.

Zárt végű	Milyen listákat képvisel	Nyílt végű	Milyen listákat képvisel
[X]	egyelemű	X	tetszőleges
[X,Y]	kételemű	[X Y]	nem üres (legalább 1 elemű)
[X,X]	két egyforma eleméből álló	[X,Y Z]	legalább 2 elemű
[X,1,Y]	3 elemből áll, 2. eleme 1	[a,b Z]	legalább 2 elemű, elemei: a, b, ...

A logikai változó

- A logikai változó fogalma:
 - kifejezésként, kifejezésben egyaránt előfordulhat, vö. a változókat a (lista) mintákban.
 - két változó azonossá tehető (azaz egyesíthető): 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** elemből álló listát épít fel az L változóban. Az elemek értéke **azonos** lesz a célsorozatbeli x változóval:

```
első_eleme([E|_], E).
második_eleme([_,E|_], E).
```

```
| ?- első_eleme(L, X), második_eleme(L, X). => L = [X,X|_A] ? ; no
```

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

```
| ?- első_eleme(L, X), második_eleme(L, X), X = alma.
    => X = alma, L = [alma,alma|_A] ? ; no
| ?- első_eleme(L, X), második_eleme(L, X), második_eleme(L, bor)
    => X = bor, L = [bor,bor|_A] ? ; no
```

Listák összefűzése: az append/3 eljárás

- `append(L1, L2, L3)`: Az L3 lista az L1 és L2 listák elemeinek egymás után fűzésével áll elő (jelöljük: $L3 = L1 \oplus L2$) — két megoldás:

```
append0([], L2, L) :- L = L2.
append0([X|L1], L2, L) :-
    append0(L1, L2, L3), L = [X|L3].
```

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

```
> append([1,2,3],[4],A)
(2) > append0([2,3],[4],B), A=[1|B]
(2) > append0([3],[4],C), B=[2|C], A=[1|B]
(2) > append0([1],[4],D), C=[3|D], B=[2|C], A=[1|B]
(1) > D=[4], C=[3|D], B=[2|C], A=[1|B]
BIP > C=[3,4], B=[2|C], A=[1|B]
BIP > B=[2,3,4], A=[1|B]
BIP > A=[1,2,3,4]
BIP > []
L = [1,2,3,4] ?
```

```
> append([1,2,3],[4],A), write(A)
(2) > append([2,3],[4],B), write([1|B])
(2) > append([3],[4],C), write([1,2|C])
(2) > append([1],[4],D), write([1,2,3|D])
(1) > write([1,2,3,4])
[1,2,3,4]
BIP > []
L = [1,2,3,4] ?
```

- Az `append0/append(L1, ...)` komplexitása: futási ideje arányos L1 hosszával.
- Miért jobb az `append/3` mint az `append0/3`?
 - `append/3` **jobbrekurzív**, ciklussal ekvivalens (nem fogyaszt vermet)
 - `append([1,...,1000],[0],[2,...])` azonnal, `append0(...)` 1000 lépésben hiúsul meg
 - `append/3` használható szétszedésre is (lásd később), míg `append0/3` nem.

Lista építése *előlről* — nyílt végű listákkal

- Az `append` eljárás már az első redukciónál felépíti az eredmény fejtét! (az eredményparaméter egy lista-minta lesz, a farok még ismeretlen, vö. logikai változó)

```
append([], L, L).
append([X|L1], L2, [X|L3]) :- append(L1, L2, L3).
| ?- append([1,2,3], [4], Ered) => Ered = [1|A], append([2,3], [4], A)
```

- Haladó nyomkövetési lehetőségek ennek demonstrálására

- `library(debugger_examples)` —példák a nyomkövető programozására, új parancsokra
- új parancs: 'N <név>' —fókuszált argumentum elnevezése
- szabványos parancs: '^ <argszám>' —adott argumentumra fókuszálás
- új parancs: 'P [<név>]' —adott nevű (ill összes) kifejezés kiírása

```
| ?- use_module(library(debugger_examples)).
| ?- trace, append([1,2,3],[4,5,6],A).
1      1 Call: append([1,2,3],[4,5,6],_543) ? ^ 3
1      1 Call: ^3 _543 ? N Ered
1      1 Call: ^3 _543 ? P => Ered = _543
2      2 Call: append([2,3],[4,5,6],_2700) ? P => Ered = [1|_2700]
3      3 Call: append([3],[4,5,6],_3625) ? P => Ered = [1,2|_3625]
4      4 Call: append([], [4,5,6], _4550) ? P => Ered = [1,2,3|_4550]
4      4 Exit: append([], [4,5,6], [4,5,6]) ? P => Ered = [1,2,3,4,5,6]
3      3 Exit: append([3],[4,5,6],[3,4,5,6]) ?
2      2 Exit: append([2,3],[4,5,6],[2,3,4,5,6]) ?
1      1 Exit: append([1,2,3],[4,5,6],[1,2,3,4,5,6]) ?
=> A = [1,2,3,4,5,6] ? ; no
```

Listák megfordítása

- Naív (négyzetes lépésszámú) megoldás

```
% nrev(L, R): Az R lista az L megfordítása.
nrev([], []).
nrev([X|L], R) :-
    nrev(L, RL),
    append(RL, [X], R).
```

- Lineáris lépésszámú megoldás

```
% reverse(R, L): Az R lista az L megfordítása.
reverse(R, L) :- revapp(L, [], R).

% revapp(L1, L2, R): L1 megfordítását L2 elé fűzve kapjuk R-t.
revapp([], R, R).
revapp([X|L1], L2, R) :-
    revapp(L1, [X|L2], R).
```

- A `lists` könyvtár tartalmazza az `append/3` és `reverse/2` eljárások definícióját.
- A könyvtár betöltése:

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

append és revapp — listák gyűjtési iránya

• Prolog megvalósítás

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

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

• C++ megvalósítás

```
struct link { link *next;
             char elem;
             link(char e): elem(e) {}
};
typedef link *list;

list append(list list1, list list2)
{ list list3, *lp = &list3;
  for (list p=list1; p; p=p->next)
  { list newl = new link(p->elem);
    *lp = newl; lp = &newl->next;
  }
  *lp = list2;
  return list3;
}

list revapp(list list1, list list2)
{ list l = list2;
  for (list p=list1; p; p=p->next)
  { list newl = new link(p->elem);
    newl->next = l; l = newl;
  }
  return l;
}
```

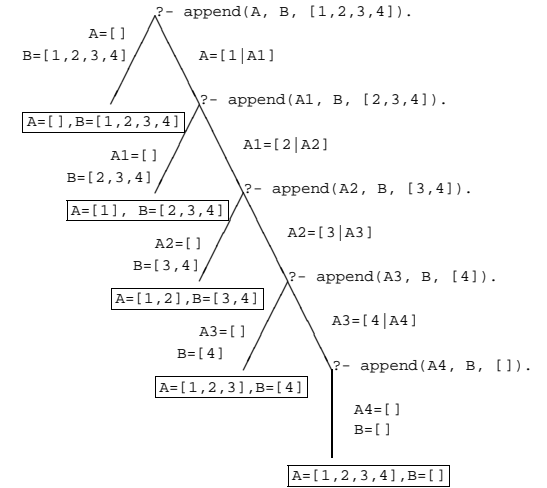
Deklaratív programozás. BME VIK, 2006. tavaszi félév

(Logikai Programozás)

Listák szétbontása az append/3 segítségével

```
% append(L1, L2, L3):
% Az L3 lista az L1 és L2
% listák elemeinek egymás
% után fűzésével áll elő.
append([], L, L).
append([X|L1], L2, [X|L3]) :-
    append(L1, L2, L3).
```

```
| ?- append(A, B, [1,2,3,4]).
A = [], B = [1,2,3,4] ? ;
A = [1], B = [2,3,4] ? ;
A = [1,2], B = [3,4] ? ;
A = [1,2,3], B = [4] ? ;
A = [1,2,3,4], B = [] ? ;
no
```



Deklaratív programozás. BME VIK, 2006. tavaszi félév

(Logikai Programozás)

Variációk appendre 1. — Három lista összefűzése

- Az append/3 keresési tere **véges**, ha első és harmadik argumentuma közül legalább az egyik zárt végű lista.

- $\text{append}(L1, L2, L3, L123) : L1 \oplus L2 \oplus L3 = L123$

```
append(L1, L2, L3, L123) :-
    append(L1, L2, L12), append(L12, L3, L123).
```

- Nem hatékony, pl.: $\text{append}([1, \dots, 100], [1, 2, 3], [1], L)$ 103 helyett 203 lépés!
- Szétszedésre nem alkalmas — végtelen választási pontot hoz létre

- Szétszedésre is alkalmas, hatékony változat

```
% L1 \oplus L2 \oplus L3 = L123, ahol vagy L1 és L2, vagy L123 adott (zárt végű).
append(L1, L2, L3, L123) :-
    append(L1, L23, L123), append(L2, L3, L23).
```

- Az első append/3 hívás nyílt végű listát állít elő:


```
| ?- append([1,2], L23, L).      =>      L = [1,2|L23] ?
```
- Az L3 argumentum behelyettesíthetősége (nyílt vagy zárt végű lista-e) nem számít.

Deklaratív programozás. BME VIK, 2006. tavaszi félév

(Logikai Programozás)

Mintakeresés append/3-mal

- Párban előforduló elemek

```
% párban(Lista, Elem): A Lista számlistának Elem olyan
% eleme, amelyet egy ugyanilyen elem követ.
párban(L, E) :-
    append(_, [E,E|_], L).
```

```
| ?- párban([1,8,8,3,4,4], E).
E = 8 ? ; E = 4 ? ; no
```

- Dadogó részek

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

```
| ?- dadogó([2,2,1,2,2,1], D).
D = [2] ? ; D = [2,2,1] ? ; D = [2] ? ; no
```

Deklaratív programozás. BME VIK, 2006. tavaszi félév

(Logikai Programozás)

Keresés listában

- `member(E, L)`: E az L lista eleme


```
member(Elem, [Elem|_]).
member(Elem, [_|Farok]) :-
    member(Elem, Farok).
```

<pre>member(Elem, [Fej Farok]) :- (Elem = Fej ; member(Elem, Farok)).</pre>

- A `member/2` felhasználási lehetőségei
 - Eldöntendő (igen-nem) kérdés:


```
| ?- member(2, [1,2,3]).           => yes
```
 - Lista elemeinek felsorolása:


```
| ?- member(X, [1,2,3]).           => X = 1 ? ; X = 2 ? ; X = 3 ? ; no
| ?- member(X, [1,2,1]).           => X = 1 ? ; X = 2 ? ; X = 1 ? ; no
```
 - Listák közös elemeinek felsorolása – mindkét fenti hívásmintát használja:


```
| ?- member(X, [1,2,3]),
    member(X, [5,4,3,2,3]).         => X = 2 ? ; X = 3 ? ; X = 3 ? ; no
```
 - Egy értéket egy (nyílt végű) lista elemévé tesz, végtelen választás!


```
| ?- member(1, L).                 => L = [1|_A] ? ; L = [_A,1|_B] ? ;
    L = [_A,_B,1|_C] ? ; ...
```
- A `member/2` keresési tere **véges**, ha második argumentuma zárt végű lista.

member/2 általánosítása: select/3

- `select(Elem, Lista, Marad)`: Elemet a Listából elhagyva marad Marad.


```
select(Elem, [Elem|Marad], Marad). % Elhagyjuk a fejet, marad a farok.
select(Elem, [X|Farok], [X|Marad0]) :- % Marad a fej,
    select(Elem, Farok, Marad0). % a farokból hagyunk el elemet.
```
- Felhasználási lehetőségek:


```
| ?- select(1, [2,1,3], L).         % Adott elem elhagyása
    L = [2,3] ? ; no
| ?- select(X, [1,2,3], L).         % Akármelyik elem elhagyása
    L=[2,3], X=1 ? ; L=[1,3], X=2 ? ; L=[1,2], X=3 ? ; no
| ?- select(3, L, [1,2]).           % Adott elem beszűrése!
    L = [3,1,2] ? ; L = [1,3,2] ? ; L = [1,2,3] ? ; no
| ?- select(3, [2|L], [1,2,7,3,2,1,8,9,4]).
    % Beszűrhető-e 3 az [1,...]-ba
    no % úgy, hogy [2,...]-t kapjunk?
| ?- select(1, [X,2,X,3], L).
    L = [2,1,3], X = 1 ? ; L = [1,2,3], X = 1 ? ; no
```
- A `lists` könyvtár tartalmazza a `member/2` és `select/3` eljárások definícióját is.
- A `select/3` keresési tere **véges**, ha 2. és 3. argumentuma közül legalább az egyik zárt végű.

Listák permutációja

- `permutation(Lista, Perm)`: Lista permutációja a Perm lista.
(Az alábbi definíció a `library(lists)` könyvtárból származik:)


```
permutation([], []).
permutation(Lista, [Elso|Perm]) :-
    select(Elso, Lista, Maradek),
    permutation(Maradek, Perm).
```
- Felhasználási példák:


```
| ?- permutation([1,2], L).
    L = [1,2] ? ; L = [2,1] ? ; no
| ?- permutation([a,b,c], L).
    L = [a,b,c] ? ; L = [a,c,b] ? ; L = [b,a,c] ? ;
    L = [b,c,a] ? ; L = [c,a,b] ? ; L = [c,b,a] ? ;
    no
| ?- permutation(L, [1,2]).
    L = [1,2] ? ;
    végtelen keresési tér
```
- Ha `permutation/2`-ben az első argumentum ismeretlen, akkor a `select` hívás keresési tere végtelen!

TÍPUSOK PROLOGBAN

Példa: Bináris fák

- 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:

$$\text{itree} \equiv \{\text{leaf}(i) \mid i \in \text{int}\} \cup \{\text{node}(l,r) \mid l,r \in \text{itree}\}$$
 - A bevezetendő típus-jelölésekkel (két ekvivalens megfogalmazás):


```
:- type itree == {node(itree, itree)} \\/ {leaf(int)}.
:- type itree ---> node(itree, itree) | leaf(int).
```
 - Egy **ellenőrző** Prolog predikátumként:


```
itree(leaf(V)) :-
    integer(V).
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`)

Típusok leírása Prologban

- Típusleírás: (tömör) Prolog kifejezések egy halmazának megadása
- Alaptípusok leírása: `int, float, number, atom, any`
- Új típusok felépítése:

$$\{\text{str}(T_1, \dots, T_n)\} \equiv \{\text{str}(e_1, \dots, e_n) \mid e_1 \in T_1, \dots, e_n \in T_n\}, n \geq 0$$
 Példa: `{személy(atom,atom,int)}` az olyan `személy/3` funktorú struktúrák halmaza, amelyben az első két argumentum `atom`, a harmadik egész.
- Típusok, mint halmazok úniója képezhető a `\\/` operátorral.

$$\{\text{személy}(atom,atom,int)\} \\/ \{\text{atom-atom}\} \\/ \text{atom}$$
- Egy típusleírás elnevezhető (kommentben): `:- type tnév == tleírás.`

```
:- type t1 == {atom-atom} \\/ atom.,
:- type ember == {ember-atom} \\/ {semmi}.
```
- Megkülönböztetett unió: csupa különböző funktorú összetett típus uniója. Ha S_1, \dots, S_n mind különböző funktorú, alkalmazható az egyszerűsített (Mercury) jelölés:


```
:- type T == { S1 } \\/ ... \\/ { Sn }. => :- type T ---> S1 ; ... ; Sn.   Példák:
:- type ember ---> ember-atom; semmi.
:- type fa ---> leaf(int) ; node(fa,fa).
```

Típusok leírása Prologban — folytatás

- Paraméteres típusok — példák


```
:- type pair(T1, T2) ---> T1 - T2.      % egy '-' nevű kétarg.-ú struktúra,
                                       % első arg. T1, a második T2 típusú.
:- type tree(T) ---> leaf(T)           % T típusú elemekből álló
   ; node(tree(T), tree(T)).          % bináris fa
:- type assoc_tree(KeyT, ValueT)      % KeyT és ValueT típusú
   == tree(pair(KeyT, ValueT)).       % párokból álló fa
:- type szótár == assoc_tree(szó, szó).
:- type szó == atom.
```
- Típusdeklarációk szintaxisa

<code><típusdeklaráció></code>	<code>::= <típuselnevezés> <típuskonstrukció></code>
<code><típuselnevezés></code>	<code>::= :- type <típusazonosító> == <típusleírás> .</code>
<code><típuskonstrukció></code>	<code>::= :- type <típusazonosító> ---> <megkülönb. unió> .</code>
<code><megkülönb. unió></code>	<code>::= <konstruktor> ; ...</code>
<code><konstruktor></code>	<code>::= <névkonstans> <struktúranév> (<típusleírás>, ...)</code>
<code><típusleírás></code>	<code>::= <típusazonosító> <típusváltozó> { <konstruktor> } </code> <code><típusleírás> \\/ <típusleírás></code>
<code><típusazonosító></code>	<code>::= <típusnév> <típusnév> (<típusváltozó>, ...)</code>
<code><típusnév></code>	<code>::= <névkonstans></code>
<code><típusváltozó></code>	<code>::= <változó></code>

Predikátumtípus-deklarációk

- Predikátumtípus-deklaráció


```
:- pred <eljárásnév> (<típusazonosító>, ...)
```
- Példa:


```
:- pred sum_tree(tree(int), int).
```
- Predikátummód-deklaráció (Nem kötelező, több is megadható)


```
:- mode <eljárásnév> (<módazonosító>, ...) ahol <módazonosító> ::= in | out | inout.
```

 (Mercury-ban az `inout` módazonosító nem megengedett.)
- Példák:


```
:- mode sum_tree(in, in).      % ellenőrzés
:- mode sum_tree(in, out).    % fa-összeg előállítása
:- mode sum_tree(out, in).    % adott összegű fa építése
```
- Vegyes típus- és móddeklaráció


```
:- pred <eljárásnév> (<típusazonosító> : <módazonosító>, ...)
```
- Példa:


```
:- pred between(int::in, int::in, int::out).
```

Móddeklaráció: a SICStus kézikönyv által használt alak

- A SICStus kézikönyv egy másik jelölést használ a bemenő/kimenő argumentumok jelzésére, pl.

```
sum_tree(+T, ?Sum).
```

- Mód-jelölő karakterek:

- + bemenő argumentum (behelyettesített)
- - kimenő argumentum (behelyettesítetlen)
- : eljárás-paraméter (meta-eljárásokban)
- ? tetszőleges

A Prolog szintaxis összefoglalása

- A Prolog szintaxis alapelvei

- Minden programelem kifejezés!
- A szükséges összekötő jelek (', ', ':', :- -->): szabványos operátorok.
- A beolvasott kifejezést funktora alapján osztályozzuk:
 - *kérdés*: `?- Cél.`
Célt lefuttatja, és a változó-behelyettesítéseket kiírja (ez az alapértelmezés az ún. top-level interaktív felületen).
 - *parancs*: `:- Cél.`
A Célt csendben lefuttatja. Pl. deklaráció (operátor, ...) elhelyezésére.
 - *szabály*: `Fej :- Törzs.`
A szabályt felveszi a programba.
 - *nyelvtani szabály*: `Fej --> Törzs.`
Prolog szabállyá alakítja és felveszi (lásd a DCG nyelvtan).
 - *tényállítás*: `Minden egyéb kifejezés.`
Üres törzsű szabályként felveszi a programba.

A PROLOG SZINTAXIS

A Prolog nyelv-változatok

- A SICStus rendszer két üzemmódja
 - *iso* Az ISO Prolog szabványnak megfelelő.
 - *sicstus* Korábbi változatokkal kompatibilis.
 - Állítása: `set_prolog_flag(language, Mód).`
 - Különbségek:
 - szintaxis-részletek, pl. a `0x1ff` szám-alak csak ISO módban,
 - beépített eljárások viselkedésének kisebb eltérései.
 - az eddig ismertett eljárások hatása lényegében nem változik.

Szintaktikus édesítőszer — összefoglalás, gyakorlati tanácsok

Operátoros kifejezések alapstruktúra alakra hozása

- Zárójelezzük be a kifejezést, az operátorok prioritása és fajtája alapján, például $-a+b*2 \Rightarrow ((-a)+(b*2))$.
- Hozzuk az operátoros kifejezéseket alapstruktúra alakra:
 $(A \text{ Inf } B) \Rightarrow \text{Inf}(A,B)$, $(\text{Pref } A) \Rightarrow \text{Pref}(A)$, $(A \text{ Postf}) \Rightarrow \text{Postf}(A)$
 Példa: $((-a)+(b*2)) \Rightarrow (-a) + *(b,2) \Rightarrow +(-a), *(b,2)$.
- Trükkös esetek:
 - A vesszőt névként idézni kell: pl. $(pp,(qq;rr)) \Rightarrow ', '(pp,;(qq,rr))$.
 - *Szám* \Rightarrow negatív számkonstans, de - *Egyéb* \Rightarrow prefix alak.
 Példa: $-1+2 \Rightarrow +(-1,2)$, de $-a+b \Rightarrow +(-a),b$.
 - Név(...)* \Rightarrow struktúrakifejezés;
Név (...) \Rightarrow prefix operátoros kifejezés. Példák:
 $-(1,2) \Rightarrow -(1,2)$ (változatlan), de
 $-(1,2) \Rightarrow -(', '(1,2))$.

Szintaktikus édesítőszer — listák, egyébek

Listák alapstruktúra alakra hozása

- Farok-megadás betoldása.
 $[1,2] \Rightarrow [1,2|[]]$. $[[X|Y]] \Rightarrow [[X|Y]|[]]$
 - Vessző (ismételt) kiküszöbölése $[Elem1,Elem2\dots] \Rightarrow [Elem1|[Elem2\dots]]$.
 $[1,2|[]] \Rightarrow [1|[2|[]]]$
 $[1,2,3|[]] \Rightarrow [1|[2,3|[]]] \Rightarrow [1|[2|[3|[]]]]$
 - Struktúrakifejezéssé alakítás: $[Fej|Farok] \Rightarrow .(Fej,Farok)$.
 $[1|[2|[]]] \Rightarrow .(1,.(2,[]))$, $[[X|Y]|[]] \Rightarrow .((X,Y),[])$
- ### Egyéb szintaktikus édesítőszer:
- Karakterkód-jelölés: $0'Kar$.
 $0'a \Rightarrow 97$, $0'b \Rightarrow 98$, $0'c \Rightarrow 99$, $0'd \Rightarrow 100$, $0'e \Rightarrow 101$
 - Fűzér (string): "xyz..." \Rightarrow az xyz... karakterek kódját tartalmazó lista
 $"abc" \Rightarrow [97,98,99]$, "" $\Rightarrow []$, "e" $\Rightarrow [101]$
 - Kapcsos zárójelezés: $\{Kif\} \Rightarrow \{ \}(Kif)$ (egy $\{ \}$ nevű, egyargumentumú struktúra — a $\{ \}$ jelpár egy önálló lexikai elem, egy névkonstans).
 - Bináris, hexa stb. alak (csak iso módban), pl. $0b101010$, $0x1a$.

Kifejezések szintaxisa — kétszintű nyelvtanok

Egy részlet egy „hagyományos” nyelv kifejezés-szintaxisából:

$$\begin{aligned} \langle \text{kifejezés} \rangle ::= & \langle \text{tag} \rangle \\ & | \langle \text{kifejezés} \rangle \langle \text{additív művelet} \rangle \langle \text{tag} \rangle \\ \langle \text{tag} \rangle ::= & \langle \text{tényező} \rangle \\ & | \langle \text{tag} \rangle \langle \text{multiplikatív művelet} \rangle \langle \text{tényező} \rangle \\ \langle \text{tényező} \rangle ::= & \langle \text{szám} \rangle | \langle \text{azonosító} \rangle | (\langle \text{kifejezés} \rangle) \end{aligned}$$

Ugyanez kétszintű nyelvtannal:

$$\begin{aligned} \langle \text{kifejezés} \rangle ::= & \langle \text{kif } 2 \rangle \\ \langle \text{kif } N \rangle ::= & \langle \text{kif } N-1 \rangle \\ & | \langle \text{kif } N \rangle \langle N \text{ prioritású művelet} \rangle \langle \text{kif } N-1 \rangle \\ \langle \text{kif } 0 \rangle ::= & \langle \text{szám} \rangle | \langle \text{azonosító} \rangle | (\langle \text{kif } 2 \rangle) \\ \{ \text{az additív ill. multiplikatív műveletek prioritása } 2 \text{ ill. } 1 \} \end{aligned}$$

Prolog kifejezések szintaxisa

$$\begin{aligned} \langle \text{programelem} \rangle ::= & \langle \text{kifejezés } 1200 \rangle \langle \text{záró-pont} \rangle \\ \langle \text{kifejezés } N \rangle ::= & \langle \text{op } N \text{ fx} \rangle \langle \text{köz} \rangle \langle \text{kifejezés } N-1 \rangle \\ & | \langle \text{op } N \text{ fy} \rangle \langle \text{köz} \rangle \langle \text{kifejezés } N \rangle \\ & | \langle \text{kifejezés } N-1 \rangle \langle \text{op } N \text{ fxf} \rangle \langle \text{kifejezés } N-1 \rangle \\ & | \langle \text{kifejezés } N-1 \rangle \langle \text{op } N \text{ xfy} \rangle \langle \text{kifejezés } N \rangle \\ & | \langle \text{kifejezés } N \rangle \langle \text{op } N \text{ yfx} \rangle \langle \text{kifejezés } N-1 \rangle \\ & | \langle \text{kifejezés } N-1 \rangle \langle \text{op } N \text{ xf} \rangle \\ & | \langle \text{kifejezés } N \rangle \langle \text{op } N \text{ yf} \rangle \\ & | \langle \text{kifejezés } N-1 \rangle \\ \langle \text{kifejezés } 1000 \rangle ::= & \langle \text{kifejezés } 999 \rangle , \langle \text{kifejezés } 1000 \rangle \\ \langle \text{kifejezés } 0 \rangle ::= & \langle \text{név} \rangle (\langle \text{argumentumok} \rangle) \\ & \{ A \langle \text{név} \rangle \text{ és a } (\text{közvetlenül egymás után áll!} \} \\ & | (\langle \text{kifejezés } 1200 \rangle) | \{ \langle \text{kifejezés } 1200 \rangle \} \\ & | \langle \text{lista} \rangle | \langle \text{füzér} \rangle \\ & | \langle \text{név} \rangle | \langle \text{szám} \rangle | \langle \text{változó} \rangle \end{aligned}$$

Kifejezések szintaxisa — folytatás

$\langle \text{op } N T \rangle ::=$	$\langle \text{név} \rangle$ {feltéve, hogy $\langle \text{név} \rangle$ N prioritású és T típusú operátornak lett deklarálva}
$\langle \text{argumentumok} \rangle ::=$	$\langle \text{kifejezés } 999 \rangle$ $\langle \text{kifejezés } 999 \rangle, \langle \text{argumentumok} \rangle$
$\langle \text{lista} \rangle ::=$	$[]$ $[\langle \text{listakif} \rangle]$
$\langle \text{listakif} \rangle ::=$	$\langle \text{kifejezés } 999 \rangle$ $\langle \text{kifejezés } 999 \rangle, \langle \text{listakif} \rangle$ $\langle \text{kifejezés } 999 \rangle \langle \text{kifejezés } 999 \rangle$
$\langle \text{szám} \rangle ::=$	$\langle \text{előjeltelen szám} \rangle$ $+ \langle \text{előjeltelen szám} \rangle$ $- \langle \text{előjeltelen szám} \rangle$
$\langle \text{előjeltelen szám} \rangle ::=$	$\langle \text{természetes szám} \rangle$ $\langle \text{lebegőpontos szám} \rangle$

Kifejezések szintaxisa — megjegyzések

- A $\langle \text{kifejezés } N \rangle$ -ben $\langle \text{köz} \rangle$ csak akkor kell ha az őt követő kifejezés nyitó-zárójellel kezdődik.

```
| ?- op(500, fx, succ).
yes
| ?- write_canonical(succ(1,2)), nl, write_canonical(succ(1,2)).
succ('','(1,2))
succ(1,2)
```

- A $\langle \{ \text{kifejezés} \} \rangle$ azonos a $\langle \{ \langle \text{kifejezés} \rangle \} \rangle$ struktúrával, ez pl. a DCG nyelvtanoknál hasznos.

```
| ?- write_canonical({a}).
{a}
```

- Egy $\langle \text{füzér} \rangle$ " jelek közé zárt karaktersorozat, általában a karakterek kódjainak listájával azonos.

```
| ?- write("baba").
[98,97,98,97]
```

A Prolog lexikai elemei 1. (ismétlés)

- $\langle \text{név} \rangle$
 - kisbetűvel kezdődő alfanumerikus jelsorozat (ebben megengedve kis- és nagybetűt, számjegyeket és aláhúzásjelet);
 - egy vagy több ún. speciális jelből (+-*/\\$\^<>= `~: . ?@#&) álló jelsorozat;
 - az önmagában álló ! vagy ; jel;
 - a [] { } jelpárok;
 - idézőjelek (') közé zárt tetszőleges jelsorozat, amelyben \ jellel kezdődő escape-szekvenciákat is elhelyezhetünk.
- $\langle \text{változó} \rangle$
 - nagybetűvel vagy aláhúzással kezdődő alfanumerikus jelsorozat.
 - az azonos jelsorozattal jelölt változók egy klózon belül azonosaknak, különböző klózokban különbözőeknek tekintődnek;
 - kivétel: a semmis változók (_) minden előfordulása különböző.

A Prolog lexikai elemei 2.

- $\langle \text{természetes szám} \rangle$
 - (decimális) számjegysorozat;
 - 2, 8 ill. 16 alapú számrendszerben felírt szám, ilyenkor a számjegyeket rendre a 0b, 0o, 0x karakterekkel kell prefixálni (csak iso módban)
 - karakterkód-konstans 0'c alakban, ahol c egyetlen karakter (vagy egy ilyet jelölő escape-szekvencia)
- $\langle \text{lebegőpontos szám} \rangle$
 - mindenképpen tartalmaz tizedespontot
 - mindkét oldalán legalább egy (decimális) számjegyvel
 - e vagy E betűvel jelzett esetleges exponens

Megjegyzések és formázó-karakterek

- Megjegyzések (comment)
 - A % százalékjeltől a sor végéig
 - A /* jelpártól a legközelebbi */ jelpárig.
- Formázó elemek
 - szóköz, újsor, tabulátor stb. (nem látható karakterek)
 - megjegyzés
- A programszöveg formázása
 - formázó elemek (szóköz, újsor stb.) szabadon elhelyezhetők;
 - kivétel: struktúrakifejezés neve után nem szabad formázó elemet tenni;
 - prefix operátor és (közé kötelező formázó elemet tenni;
 - < záró-pont): egy . karakter amit egy formázó elem követ.

PROLOG PÉLDÁK

A régi jegyzet bevezető példája: útvonalkeresés

- A feladat:
 - Tekintsük (autóbusz)járatok egy halmazát.
 - Mindegyik járhoz a két végpont és az útvonal hossza van megadva.
 - Írjunk Prolog eljárást, amellyel megállapítható, hogy két pont összeköthető-e pontosan N csatlakozó járattal!
- Átfogalmazás: egy súlyozott irányítatlan gráfban két pont közötti utat keresünk. Élek:


```
% járat(A, B, H): Az A és B városok között van járat, és hossza H km.
járat('Budapest', 'Prága', 515).
járat('Budapest', 'Bécs', 245).
járat('Bécs', 'Berlin', 635).
járat('Bécs', 'Párizs', 1265).
```
- Irányított élek:


```
% útszakasz(A, B, H): A-ból B-be eljuthatunk egy H úthosszú járattal.
útszakasz(Kezdet, Cél, H) :-
(   járat(Kezdet, Cél, H)
;   járat(Cél, Kezdet, H)
).
```

Az útvonalkeresési feladat — folytatás

- Adott lépésszámú útvonal (él-sorozat) és hossza:

```
% útvonal(N, A, B, H): A és B között van (pontosan)
% N szakaszból álló útvonal, amelynek összhossza H.
útvonal(0, Hová, Hová, 0).
útvonal(N, Honnan, Hová, H) :-
    N > 0,
    N1 is N-1,
    útszakasz(Honnan, Közben, H1),
    útvonal(N1, Közben, Hová, H2),
    H is H1+H2.
```

- Futási példa:

```
| ?- útvonal(2, 'Párizs', Hová, H).
    H = 1900, Hová = 'Berlin' ? ;
    H = 2530, Hová = 'Párizs' ? ;
    H = 1510, Hová = 'Budapest' ? ;
    no
```

Körmentes út keresése

- Könyvtár betöltése, adott funktorú eljárások importálásával:

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

- Segéd-argumentum: az érintett városok listája, fordított sorrendben

```
% útvonala_2(N, A, B, H): A és B között van (pontosan)
% N szakaszból álló körmentes útvonala, amelynek összhossza H.
útvonala_2(N, Honnan, Hová, H) :-
    útvonala_2(N, Honnan, Hová, [Honnan], H).

% útvonala_2(N, A, B, Kizártak, H): A és B között van pontosan
% N szakaszból álló körmentes, Kizártak elemein át nem menő H hosszú út.
útvonala_2(0, Hová, Hová, Kizártak, 0).
útvonala_2(N, Honnan, Hová, Kizártak, H) :-
    N > 0, N1 is N-1, útszakasz(Honnan, Közben, H1),
    \+ member(Közben, Kizártak),
    útvonala_2(N1, Közben, Hová, [Közben|Kizártak], H2), H is H1+H2.
```

- Példa-futás:

```
| ?- útvonala_2(2, 'Párizs', Hová, H).
    H = 1900, Hová = 'Berlin' ? ;
    H = 1510, Hová = 'Budapest' ? ; no
```

Továbbfejlesztés: körmentes út keresése, útvonal-gyűjtéssel

- Az alapötlet: a `Kizártak` listában gyűlik a (fordított) útvonal.
- A rekurzív eljárásban szükséges egy **új argumentum**, hogy az útvonalat kiadjuk!

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

```
% útvonala_3(N, A, B, Út, H): A és B között van (pontosan)
% N szakaszból álló körmentes Út útvonala, amelynek összhossza H.
útvonala_3(N, Honnan, Hová, Út, H) :-
    útvonala_3(N, Honnan, Hová, [Honnan], Fűt, H),
    reverse(Fűt, Út).

% útvonala_3(N, A, B, Fűt0, Fűt, H): A és B között van pontosan
% N szakaszból álló körmentes, Fűt0 elemein át nem menő H hosszú út.
% Fűt = (az A → B útvonal megfordítása) ⊕ Fűt0.
útvonala_3(0, Hová, Hová, Fordűt, Fordűt, 0).
útvonala_3(N, Honnan, Hová, Fordűt0, Fordűt, H) :-
    N > 0, N1 is N-1, útszakasz(Honnan, Közben, H1),
    \+ member(Közben, Fordűt0),
    útvonala_3(N1, Közben, Hová, [Közben|Fordűt0], Fordűt, H2), H is H1+H2.

| ?- útvonala_3(2, 'Párizs', _, Út, H).
    H = 1900, Út = ['Párizs', 'Bécs', 'Berlin'] ? ;
    H = 1510, Út = ['Párizs', 'Bécs', 'Budapest'] ? ; no
```

Súlyozott gráf ábrázolása éllistával

- A gráf ábrázolása

- a gráf élek listája,
- az él egy három-argumentumú struktúra,
- argumentumai: a két végpont és a súly.

- Típus-definíció

```
% :- type él ---> él(pont, pont, súly).
% :- type pont == atom.
% :- type súly == int.
% :- type gráf == list(él).
```

- Példa

```
hálózat([él('Budapest', 'Bécs', 245),
        él('Budapest', 'Prága', 515),
        él('Bécs', 'Berlin', 635),
        él('Bécs', 'Párizs', 1265)]).
```

Ismétlődésmentes útvonal keresése listával ábrázolt gráfban

```
:- use_module(library(lists), [select/3]).
```

```
% útvonala_4(N, G, A, B, L, H): A G gráfban van egy A-ból
% B-be menő N szakaszból álló L út, melynek összhossza H.
útvonala_4(0, _Gráf, Hová, Hová, [Hová], 0).
útvonala_4(N, Gráf, Honnan, Hová, [Honnan|Út], H) :-
    N > 0, N1 is N-1,
    select(Él, Gráf, Gráf1),
    él_végpontok_hossz(Él, Honnan, Közben, H1),
    útvonala_4(N1, Gráf1, Közben, Hová, Út, H2),
    H is H1+H2.
```

```
% él_végpontok_hossz(Él, A, B, H): Az Él irányítatlan él
% végpontjai A és B, hossza H.
él_végpontok_hossz(él(A,B,H), A, B, H).
él_végpontok_hossz(él(A,B,H), B, A, H).
```

```
| ?- hálózat(_Gráf), útvonala_4(2, _Gráf, 'Budapest', _, Út, H).
    H = 880, Út = ['Budapest', 'Bécs', 'Berlin'] ? ;
    H = 1510, Út = ['Budapest', 'Bécs', 'Párizs'] ? ;
    no
```

Bináris fákra vonatkozó példasor — fa levele

- Ismétlés: egészekből álló bináris fa:

```
:- type itree == {node(itree, itree)} \\/ {leaf(int)}.
:- type itree ----> node(itree, itree) | leaf(int).
```

- Írjunk egy predikátumot annak eldöntésére, hogy egy adott érték szerepel-e egy fa levelében (vö. member/2)!

```
% 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 a bal részében van, akkor az egészben is
fa_levele(node(_,R) , V) :-
    fa_levele(R, V). % ha a jobb részében van, akkor az egészben is
```

- Az aláhúzásjel egy ún. semmis (void) változó, ennek minden előfordulása különböző változó!
- Példák: ellenőrzés (1), adott fa leveleinek felsorolása (2), adott levelű fák felsorolása, (3) (∞ keresési tér).

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

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
```

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 megghiúsul, pl. ún. visszalépéses ciklus szervezésére jó.

Levél beszúrása bináris fába

- Írjunk egy predikátumot arra, hogy egy adott értékű levelet egy fába minden lehetséges módon beszúrjon!

- Nem kell írunk, 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).
...
```
- 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?`