Peano aritmetika — összeadás

- A természetes számok halmazán az összeadást definiálhatjuk a Peano axiómákkal ha a számokat az `s(X)` „rákövetkező” függvény segítségével ábrázoljuk:
  `1 = s(0), 2 = s(s(0)), 3 = s(s(s(0))), ...` (Peano ábrázolás).

```
% plus(X, Y, Z): X és Y összege Z (X, Y, Z Peano ábrázolású).
plus(0, X, X).                    % 0+X = X.
plus(s(X), Y, s(Z)) :-
        plus(X, Y, Z).            % s(X)+Y = s(X+Y).
```

- A `plus` predikátum több irányban is használható:

```
| ?- plus(s(0), s(s(0)), Z).        Z = s(s(s(0))) ? ; no     %  1+2 = 3

| ?- plus(s(0), Y, s(s(s(0)))).     Y = s(s(0)) ? ; no        %  3-1 = 2

| ?- plus(X, Y, s(s(0))).           X = 0, Y = s(s(0)) ? ;    %  2 = 0+2
                                    X = s(0), Y = s(0) ? ;    %  2 = 1+1
                                    X = s(s(0)), Y = 0 ? ;    %  2 = 2+0
                                    no
| ?-
```


Adott összegű fák építése

- Adott összegű fát építő eljárás Peano aritmetikával:

```
sum_tree(leaf(Value), Value).
sum_tree(node(Left, Right), S) :-
        plus(S1, S2, S),
        S1 \= 0, S2 \= 0,      % X \= Y beépített eljárás, jelentése: X ≠ Y.
                               % A 0-t kizárjuk, mert különben ∞ megoldás van.
        sum_tree(Left, S1),
        sum_tree(Right, S2).
```

- Az eljárás futása:

```
| ?- sum_tree(Tree, s(s(s(0)))).
Tree = leaf(s(s(s(0)))) ? ;                                    % 3
Tree = node(leaf(s(0)),leaf(s(s(0)))) ? ;                      % (1+2)
Tree = node(leaf(s(0)),node(leaf(s(0)),leaf(s(0)))) ? ;        % (1+(1+1))
Tree = node(leaf(s(s(0))),leaf(s(0))) ? ;                      % (2+1)
Tree = node(node(leaf(s(0)),leaf(s(0))),leaf(s(0))) ? ;        % ((1+1)+1)
no
```


A Prolog adatfogalma, a Prolog kifejezés

- konstans (*atomic*)
  - számkonstans (*number*) — egész vagy lebegőpontos, pl. `1, -2.3, 3.0e10`
  - névkonstans (*atom*), pl. `'István', ispow, +, -, <, sum_tree`
- összetett- vagy struktúra-kifejezés (*compound*)
  - ún. kanonikus alak: ⟨ struktúranév ⟩`(`⟨ arg_1 ⟩, ...`)`
    - a ⟨ struktúranév ⟩ egy névkonstans, az ⟨ arg_i ⟩ argumentumok tetszőleges Prolog kifejezések
    - példák: `leaf(1), person(william,smith,2003,1,22), <(X,Y), is(X, +(Y,1))`
  - szintaktikus „édesítőszerek”, pl. operátorok: `X is Y+1` ≡ `is(X, +(Y,1))`
- változó (*variable*)
  - pl. `X, Szulo, X2, _valt, _, _123`
  - a változó alaphelyzetben behelyettesítetlen, értékkel nem bír, az egyesítés (mintaillesztés) művelete során egy tetszőleges Prolog kifejezést vehet fel értékül (akár egy másik változót)

A PROLOG LOGIKAI ALAPJAI

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 függvénynév, a konstansnév (azaz 0-argumentumú függvénynév) és X változónév.
  - Elemi állítás: egy relációjel, megfelelő számú argumentummal ellátva, ahol az argumentumok kifejezések, pl. $osztja(X, X * Y)$.
  - 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 aposztrofok közé tesszük. Speciális jelek ill. jelsorozatok 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:

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

  - az implikáció bal (következmény) oldala a klóz **feje**
  - az implikáció feltétele a klóz **törzse**, a törzsbeli konjunkció elemeit (rész)**cél**oknak is hívjuk
  - F_i és T_j elemi állítások, $n, m \geq 0$, azaz a fej és a törzs is lehet üres.
  - $X_1 \ldots X_j$: a klózban szereplő összes változó.
- A fentivel ekvivalens logikai alak (vö. $A \leftarrow B \equiv A \vee \neg B$):

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

- Klózok egyszerűsített írásmódja: $F_1, \ldots, F_n$: $-T_1, \ldots, T_m$. 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).               ≡ ∀ X ¬ (ferfi(X) ∧ 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.
```


Definit klózok — a Prolog program építőelemei

- Általános klóz (ismétlés): $F_1, \ldots, F_n$: $-T_1, \ldots, T_m$. $\forall \overline{X}(F_1 \vee \ldots \vee F_n \vee \neg T_1 \vee \ldots \vee \neg T_m)$
- 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 **szabály**nak hívjuk, pl.
    `nagyszuloje(U,N):-szuloje(U,Sz),szuloje(Sz,N).`
    logikai alak: $\forall UNSz(\texttt{nagyszuloje}(U, N) \leftarrow \texttt{szuloje}(U, Sz) \wedge \texttt{szuloje}(Sz, N))$
    ekvivalens alak: $\forall UN \ (\texttt{nagyszuloje}(U, N) \leftarrow \exists Sz(\texttt{szuloje}(U, Sz) \wedge \texttt{szuloje}(Sz, N)))$
  - $n = 1, m = 0$ 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: $\forall X \neg \texttt{nagyszuloje}(\texttt{'Imre'}, X)$, azaz $\neg \exists X \texttt{nagyszuloje}(\texttt{'Imre'}, X)$
  - Ha $n = 0, m = 0$, akkor **üres klóz**ról beszélünk, jele: $\square$. Logikailag üres diszjunkció, azaz azonosan hamis.

A Prolog mint logikai nyelv

- Szintaxis:
  - Prolog program: szabályok és tényállítások halmaza. Példa:

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

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

  - Egy klóz fejének nevét és argumentumszámát együtt a klóz **funktor**ának hívjuk és `Név/Argszám` alakban írjuk.
  - Az azonos funktorú klózok alkotják egy **predikátum** (vagy eljárás) definícióját. A fenti példa a `szuloje/2` és `nagyszuloje/2` predikátumokat definiálja.
  - Egy program futtatásához megadandó egy célsorozat. Példa:
    `:- nagyszuloje('Imre', N).`

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: $f(x_1, \ldots, x_n) = z \Leftrightarrow (z = f(y_1, \ldots, y_n) \wedge x_1 = y_1) \wedge \ldots \wedge (x_n = y_n))$
  - 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 mint logikai nyelv

- Deklaratív szemantika
  - Segédfogalom: egy kifejezés/állítás **példánya**: belőle változók behelyettesítésével előálló kifejezés/állítás.
  - Egy célsorozat lefutása **sikeres**, 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. **fejkomment**ben, 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 állítás**ként 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 ciklus**ba 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. Különféle megadási módok:
  - Az ún. SLD rezolúciós tételbizonyítási módszer (nagyon tömören lásd alább)
  - egy cél-redukción alapuló tételbizonyítási módszer (lásd a következő fóliákon)
  - mintaillesztésen alapuló visszalépéses eljárásszervezés (részletesen lásd később).
- A Prologban alkalmazott rezolúciós tételbizonyítási módszerről:
  - SLD resolution: **L**inear resolution with a **S**election function for **D**efinite clauses.
  - A célsorozat **tagadja** a keresett dolgok létezését, pl. `'Imre'`-nek nincs nagyszülője:
    `:- nagyszuloje('Imre', N).` $\equiv \neg\exists N$ `nagyszuloje('Imre',`N`)`
  - A célsorozat és egy programklóz ún. rezolvenseként kapunk egy újabb célsorozatot.
  - A rezolúciós lépéseket addig ismételjük, 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 $\square$.
  - 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` argumentumából olvashatjuk ki.

A redukciós lépés

- A példa érintett klózai 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 megkiséreljük:
  - 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. $\forall X_1 \ldots X_j (F \leftarrow T)$).
  - A redukciós lépés előtt a klózt le kell másolni, a változókat szisztematikusan újakra cserélve (vö. rekurzió).
- **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.

`sum_tree(leaf(X), X)` `sum_tree(T, V)`	közös alakja	`sum_tree(leaf(X), X)` és nem pl. `sum_tree(leaf(0), 0)`

  - Az egyesítés eredménye a legáltalánosabb közös alakot előállító behelyettesítés. Ez változó-átnevezéstől eltekintve egyértelmű. A példában: `T=leaf(X), V=X`.
  - Példák:

Hívás:	*Fej:*	*Behelyettesítés:*
`nagyszuloje('Imre', N)`	`nagyszuloje(Gy, NSz)`	`Gy = 'Imre', NSz = N`
`szuloje('Imre', Sz)`	`szuloje('Imre', 'István')`	`Sz = 'István'`
`szuloje('Imre', Sz)`	`szuloje('István', 'Géza')`	*nem egyesíthető*
`szereti('István', Kit)`	`szereti(X, X)`	`X = 'István', Kit = 'István'`
`szereti(Ki, Kit)`	`szereti(X, X)`	`X = Ki, Kit = Ki`

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.

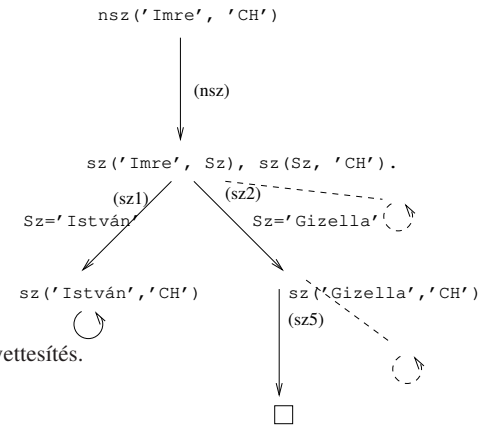
```
:- nagyszuloje('Imre', 'Civakodó Henrik').                (nsz)
:- szuloje('Imre', Sz), szuloje(Sz, 'Civakodó Henrik').   (sz1): szuloje('Imre', 'István')
:- szuloje('István', 'Civakodó Henrik').                  ???
```

- 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 klózkeresésre utalnak, az ún. első argumentum szerinti indexelés a felsőt kiküszöböli.

A keresési tér bejárásának nyomkövetése

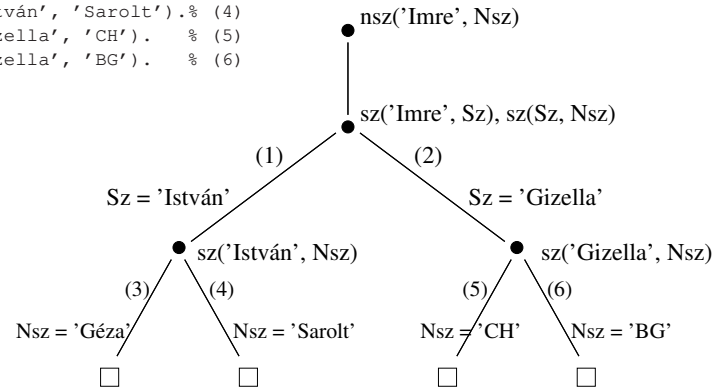
- Egy (szerkesztett) párbeszéd a redukciós nyomkövetővel, a meghiúsuló egyesítéseket elhagytuk.

```
|| ?- 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') ?
(...)                                      <--- G3-G8 6 sikertelen klózillesztés
|     |<<<< Failing back to goal G1         <--- 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 fólián alsó szaggatott
|
|<<<< No more choices                        <--- az előző fólián 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)
```

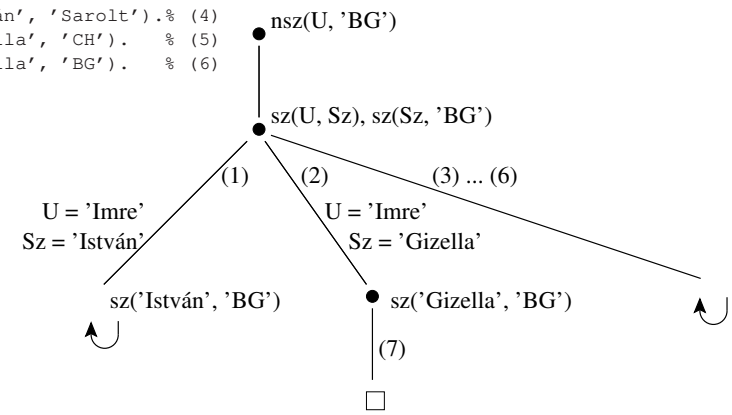
```
nsz(Gy, N) :-
    sz(Gy, Sz), sz(Sz, N).
```



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



A PROLOG ELJÁRÁSOS MODELLJEI

A Prolog végrehajtás eljárásos modelljei

- Az azonos funktorú klózok 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.
    - A modell előnyei: pontosan definiálható, a keresési tér szemléltethető
  - Eljárás-doboz modell
    - Az alapgondolat: egymásba skatulyázott eljárás-dobozokba lépünk be és ki.
    - Az alaplépések: belépés, sikeres kilépés, sikertelen kilépés.
    - Visszalépés: új megoldást kérünk egy már lefutott eljárástól.
    - A modell előnyei: közel van a hagyományos rekurzív eljárásmodellhez, a Prolog beépített nyomkövetője is ezen alapul.

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 → új célsorozat
    - zsákutca esetén visszalépés: visszatérés a legutolsó választási ponthoz
  - Választási pont:
    - egy olyan redukciós lépés amely nem a legutolsó klózzal illesztett
    - visszalépéskor visszatérünk a korábbi célsorozathoz és a **további** klózok 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)

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 **törzs**én 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 eljáráshívást 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 algoritmusa

1. *(Kezdeti beállítások:)* A verem üres, `CS := célsorozat`
2. *(Beépített eljárások:)* Ha `CS` első célja beépített akkor hajtsuk végre,
   a. Ha sikertelen ⇒ 6. lépés.
   b. Ha sikeres, `CS :=` a redukciós lépés eredménye ⇒ 5. lépés.
3. *(Klózszámláló kezdőértékezése:)* `I = 1`.
4. *(Redukciós lépés:)* Tekintsük `CS` első hívásához illeszthető klózok listáját. Ez lehet a predikátum összes klóza, vagy (indexelés esetén) ennek egy részsorozata. Tegyük fel, hogy ez a lista `N` elemű.
   a. Ha `I > N` ⇒ 6. lépés.
   b. Redukciós lépés a lista `I`-edik klóza és a `CS` célsorozat között.
   c. Ha sikertelen, akkor `I := I+1` ⇒ 4. lépés.
   d. Ha `I < N` (nem utolsó), akkor vermeljük `<CS,I>`-t.
   e. `CS :=` a redukciós lépés eredménye
5. *(Siker:)* Ha `CS` üres, akkor sikeres vég, egyébként ⇒ 2. lépés.
6. *(Sikertelenség:)* Ha a verem üres, akkor sikertelen vég.
7. *(Visszalépés:)* Ha a verem nem üres, akkor leemeljük a veremből `<CS, I>`-t, `I := I+1`, és ⇒ 4. lépés.

Indexelés (előzetes)

- Mi az indexelés?
  - egy hívásra illeszthető klózok gyors kiválasztása,
  - egy eljárás klózainak **fordítási idejű** csoportosításával.
- A legtöbb Prolog rendszer, így a SICStus Prolog is, az első fej-argumentum alapján indexel (first argument indexing).
- Az indexelés alapja az első fejargumentum külső funktora:
  - `C` szám vagy névkonstans esetén `C/0`;
  - `R` nevű és `N` argumentumú struktúra esetén `R/N`;
  - változó esetén nem értelmezett (minden funktorhoz besoroltatik).
- Az indexelés megvalósítása:
  - Fordítási időben a funktorokhoz elkészítjük az illeszthető klózok listáját
  - Futáskor lényegében konstans idő alatt választunk a részhalmazok közül.
  - *Fontos:* ha egyelemű a részhalmaz, nem hozunk létre választási pontot!
- Például `szuloje('István', X)` kételemű klózlistára szűkít, de `szuloje(X, 'István')` mind a 6 klózt megtartja (mert a SICStus Prolog csak az első argumentum szerint indexel)

Redukciós modell — előnyök és hátrányok

- Előnyök
  - (viszonylag) egyszerű és (viszonylag) precíz definíció
  - a keresési tér megjeleníthető, grafikusan 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 ?       ⇐ q-ból 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 épül

Az eljárás-doboz modell

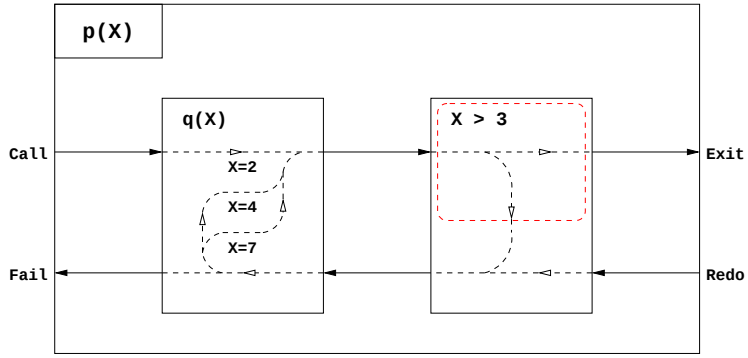
- A Prolog eljárás-végrehajtás két fázisa
  - előre menő végrehajtás: egymásba skatulyázott eljárás-belépések és -kilépések
  - visszafelé menő végrehajtás: újabb megoldás kérése egy már lefutott eljárástól
- Egy egyszerű példa

```
q(2).  q(4).  q(7).                    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` eljáráshívással 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.
```



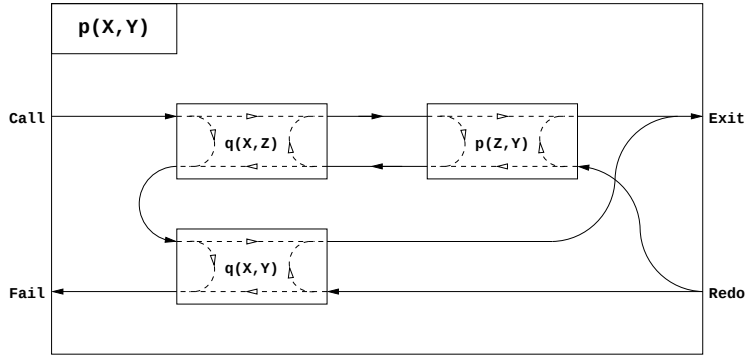
Eljárás-doboz modell — egyszerű nyomkövetési példa

- 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) ?          % ? ≡ nemdeterminisztikus 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 részeljárás Hívás kapuhoz érkezünk, **létrehozunk** egy példányt a meghívandó eljárásból, majd
- meghívjuk az eljáráspéldány „következő megoldás” metódusát (*)
  - Ha ez sikerül, akkor a vezérlés átkerül a következő hívás Hívás kapujára, vagy a szülő Kilépési kapujára
  - Ha ez meghiúsul, akkor **megszüntetjük** az eljáráspéldányt majd ugrunk az előző hívás Újra kapujára, vagy a következő klóz elejére, stb.
- Amikor egy Újra kapuhoz érkezünk, a (*) lépésnél folytatjuk.
- A szülő Újra kapuja (a „következő megoldás” nem első hívása) a tárolt klózsorszámnak megfelelő klózban az utolsó Újra kapura adja a vezérlést.

OO szemléletű dobozok: `p/2` „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).
    qaptr = new q(x, &z);     // create a new instance of subgoal q(X,Z)
  redo11:
    if(!qaptr->next()) {      // if q(X,Z) fails
      delete qaptr;           // 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 redo11;            // 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
  } }
```

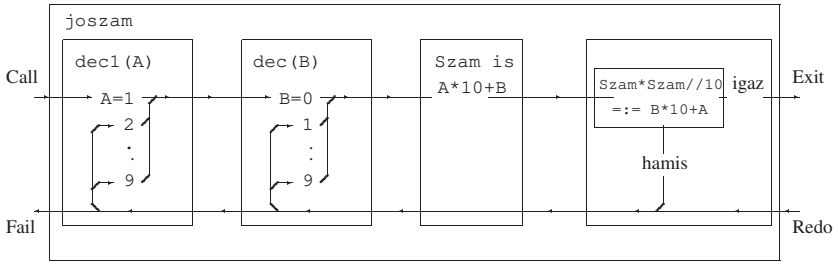

Visszalépéses keresés — egy aritmetikai példa

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

```
% dec1(J): J egy pozitív decimális számjegy.
dec1(1). dec1(2). dec1(3). dec1(4).
dec1(5). dec1(6). dec1(7). dec1(8). dec1(9).

% dec(J): J egy decimális számjegy.
dec(0).
dec(J) :- dec1(J).

% Szam négyzete háromjegyű és a Szam fordítottjával kezdődik.
joszam(Szam):-
        dec1(A), dec(B),
        Szam is A * 10 + B, Szam * Szam // 10 =:= B * 10 + A.
```

Prolog végrehajtás — a 4-kapus doboz modell

```
joszam(Szam):-
        dec1(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

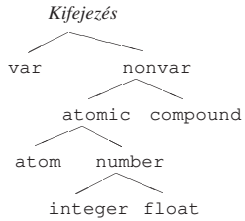
- Alapvető nyomkövetési parancsok
  - `h` <RET> (help) — parancsok listázása
  - `c` <RET> (creep) vagy <RET> — továbblépés minden kapunál megálló nyomkövetéssel
  - `l` <RET> (leap) — csak töréspontnál áll meg, de a dobozokat építi
  - `z` <RET> (zip) — csak töréspontnál áll meg, dobozokat nem épít
  - `+` <RET> ill. `-` <RET> — töréspont rakása/eltávolítása a kurrens predikátumra
  - `s` <RET> (skip) — eljárástörzs átlépése (Call/Redo ⇒ 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) — újrakezdi 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

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: $\langle$ struktúranév $\rangle$ ($\langle$ arg_1 $\rangle$, ... , $\langle$ arg_n $\rangle$)
  - $\langle$ struktúranév $\rangle$ egy tetszőleges névkonstans
  - $\langle$ 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*: $\langle$ struktúranév $\rangle$ / n

A Prolog kifejezések

- Prolog kifejezések osztályozása — osztályozó beépített predikátumok



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

- Egy osztályozó predikátum az argumentuma **pillanatnyi** állapotát **ellenőrzi**, logikailag nem tiszta:

```
| ?- 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 klózfej) 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: `szuloje('Imre', Sz),`
    fej: `szuloje('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 rendeli. 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 $y \in Dom(\theta) \backslash Dom(\sigma)$ változóhoz $y\theta$-t rendeli ($Dom(\sigma \otimes \theta) = Dom(\sigma) \cup Dom(\theta)$):

$$\sigma \otimes \theta = \{\, x \leftarrow (x\sigma)\theta \mid x \in Dom(\sigma) \} \bigcup \{\, y \leftarrow y\theta \mid y \in Dom(\theta) \backslash Dom(\sigma) \,\}$$

    - 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 egyesíthetőek 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 ($mgu(A, B)$) előállítása.
- 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 $A_1,\ldots,A_N$ ill. $B_1,\ldots,B_N$, 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 $\sigma = \sigma_1 \otimes \sigma_2 \otimes \sigma_3 \otimes \ldots$.
  5. Minden más esetben a A és B nem egyesíthető.

Egyesítési példák

- A = `sum_tree(leaf(V), V)`, B = `sum_tree(leaf(5), S)`
  - (4.) A és B neve és argumentumszáma megegyezik
    - (a.) `mgu(leaf(V), leaf(5))` (4., majd 2. szerint) = $\{$`V`$\leftarrow$`5`$\} = \sigma_1$
    - (b.) `mgu(V`σ_1`, S)` = `mgu(5, S)` (3. szerint) = $\{$`S`$\leftarrow$`5`$\} = \sigma_2$
  - tehát $mgu(A, B) = \sigma_1 \otimes \sigma_2 = \{$`V`$\leftarrow$`5`, `S`$\leftarrow$`5`$\}$
- A = `node(leaf(X), T)`, B = `node(T, leaf(3))`
  - (4.) A és B neve és argumentumszáma megegyezik
    - (a.) `mgu(leaf(X), T)` (3. szerint) = $\{$`T`$\leftarrow$`leaf(X)`$\} = \sigma_1$
    - (b.) `mgu(T`σ_1`, leaf(3))` = `mgu(leaf(X), leaf(3))` (4, majd 2. szerint) = $\{$`X`$\leftarrow$`3`$\} = \sigma_2$
  - tehát $mgu(A, B) = \sigma_1 \otimes \sigma_2 = \{$`T`$\leftarrow$`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 \= 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.
  - 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(...))))) ?
```