A PROLOG NYELV KÖZELÍTŐ SZINTAXISA

Predikátumok, klózok

- Példa:

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

- Szintaxis:

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

Prolog programok formázása

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

Prolog kifejezések

- Példa — egy klózfej mint kifejezés:

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

- Szintaxis:

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

Lexikai elemek

- Példák:

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

- Szintaxis:

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

Szintaktikus édesítőszer: operátorok

- Példa:

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

- Operátoros kifejezések

  $\langle$ összetett kifejezés $\rangle$ ::=

$\langle$ struktúranév $\rangle$ ($\langle$ argumentum $\rangle$, ...)	{eddig csak ez volt}
| $\langle$ argumentum $\rangle$ $\langle$ operátornév $\rangle$ $\langle$ argumentum $\rangle$	{infix kifejezés}
| $\langle$ operátornév $\rangle$ $\langle$ argumentum $\rangle$	{prefix kifejezés}
| $\langle$ argumentum $\rangle$ $\langle$ operátornév $\rangle$	{posztfix kifejezés}

  $\langle$ operátornév $\rangle$ ::= $\langle$ struktúranév $\rangle$ {ha operátorként lett definiálva}

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

Szabványos, beépített operátorok

Szabványos operátorok

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

Egyéb beépített operátorok SICStus Prologban

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

Operátorok jellemzői

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

Fajta			Osztály	Értelmezés
bal-asszoc.	jobb-asszoc.	nem-asszoc.		
`yfx`	`xfy`	`xfx`	infix	`X f Y` ≡ `f(X, Y)`
	`fy`	`fx`	prefix	`f X` ≡ `f(X)`
`yf`		`xf`	posztfix	`X f` ≡ `f(X)`

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

Operátorok: zárójelezés

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

Operátorok —kiegészít ő megjegyzések

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

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

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

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

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

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

Operátorok felhasználása

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

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

    Gy nagyszülője N :-  Gy szülője Sz, Sz szülője N.
    ```
  - adatstruktúrák olvashatóbbá tételére, pl.

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

    sav(kén, h.2-s-o.4).
    ```
- Miért rosszak az operátorok?
  - egyetlen globális erőforrás, ez nagyobb projektben gondot okozhat.

Aritmetika Prologban

- Az operátorok teszik lehetővé azt is, hogy a matematikában ill. más programozási nyelvekben megszokott módon írhassunk 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 csakkor sikerül, ha az értékek megegyeznek.
- Példák:

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

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

Klasszikus szimbolikus kifejezés-feldolgozás: deriválás

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

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

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

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

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

| ?- deriv(I, 0).
  ⟹  no
```

Operátoros példa: polinom behelyettesítési értéke

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

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

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

PROLOG PROGRAMOK JELENTÉSE, VÉGREHAJTÁSA

Deklaratív szemantika – klózok logikai alakja

- A matematikai logikában bevezetik az általános klóz fogalmát:
  $F_1, \ldots, F_n \text{:-} T_1, \ldots, T_m.$ $\quad\quad \forall\overline{X}(F_1 \vee \ldots \vee F_n \vee \neg T_1 \vee \ldots \vee \neg T_m)$
- 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: $\Box$. 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: $f(x_1, \ldots, x_n) = z \Leftrightarrow (z = f(y_1, \ldots, y_n) \wedge x_1 = y_1) \wedge \ldots \wedge (x_n = y_n))$
  - 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 deklaratív szemantikája

- 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 $\Box$.
  - 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 ⇒ 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').                   ???
```

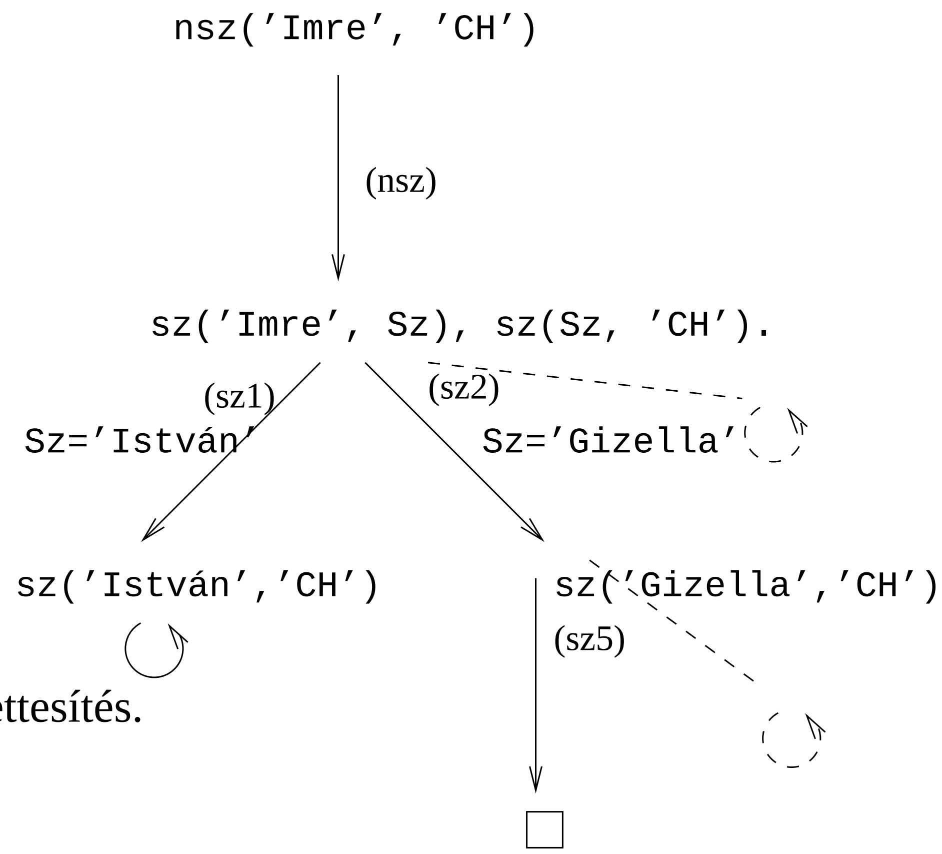
- A 2. célsorozatot az `(sz1)` klózzal redukáltuk, de a megoldáshoz az `(sz2): szuloje('Imre', 'Gizella')` vezet — nem csak az első egyesíthető klózfejet kell kezelnünk, hanem az összeset!
- Ha nem az utolsó klózzal redukálunk, akkor létrehozunk egy **választási pontot**, ebben elmentjük a célsorozatot 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

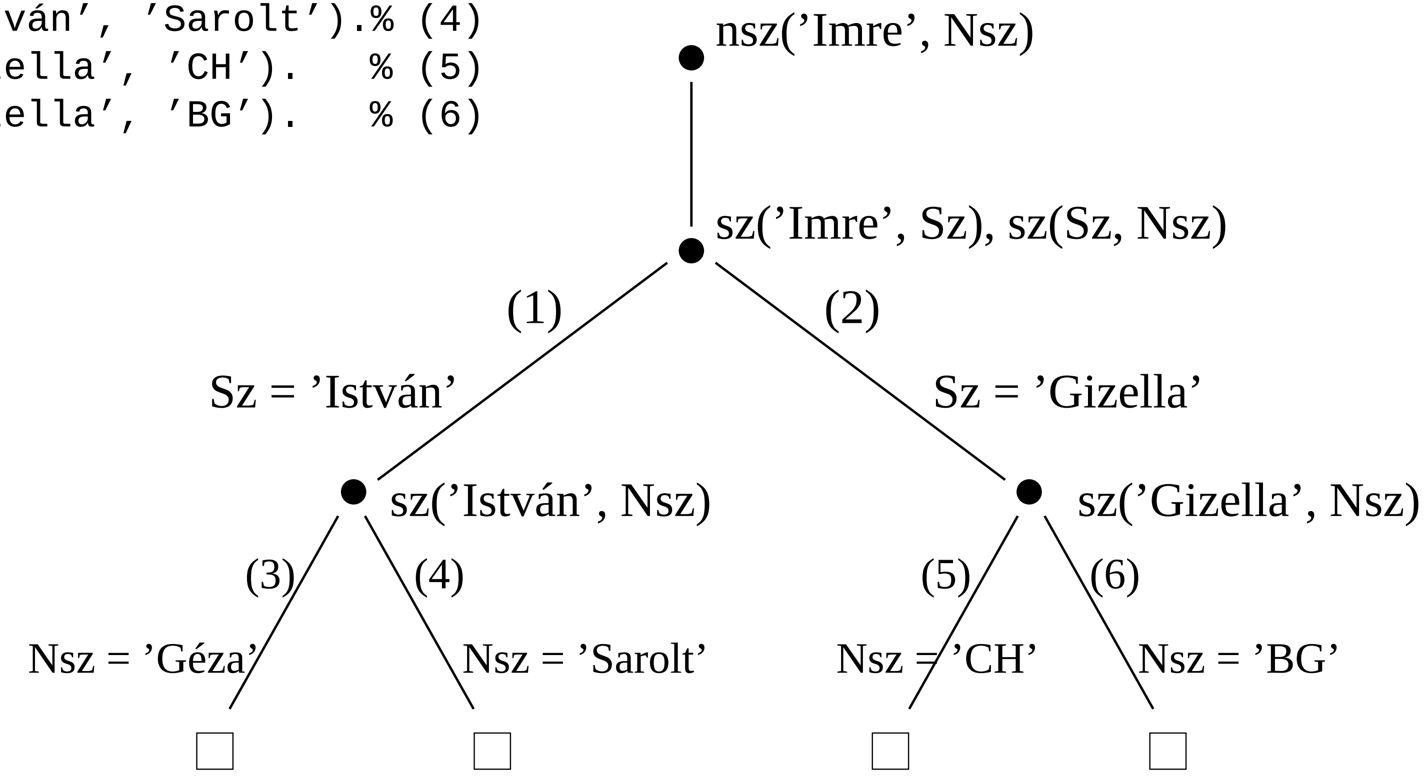
- Egy (szerkesztett) párbeszéd a redukciós nyomkövetővel, a meghiúsuló egyesítéseket elhagytuk.

```
|| ?- 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)
```

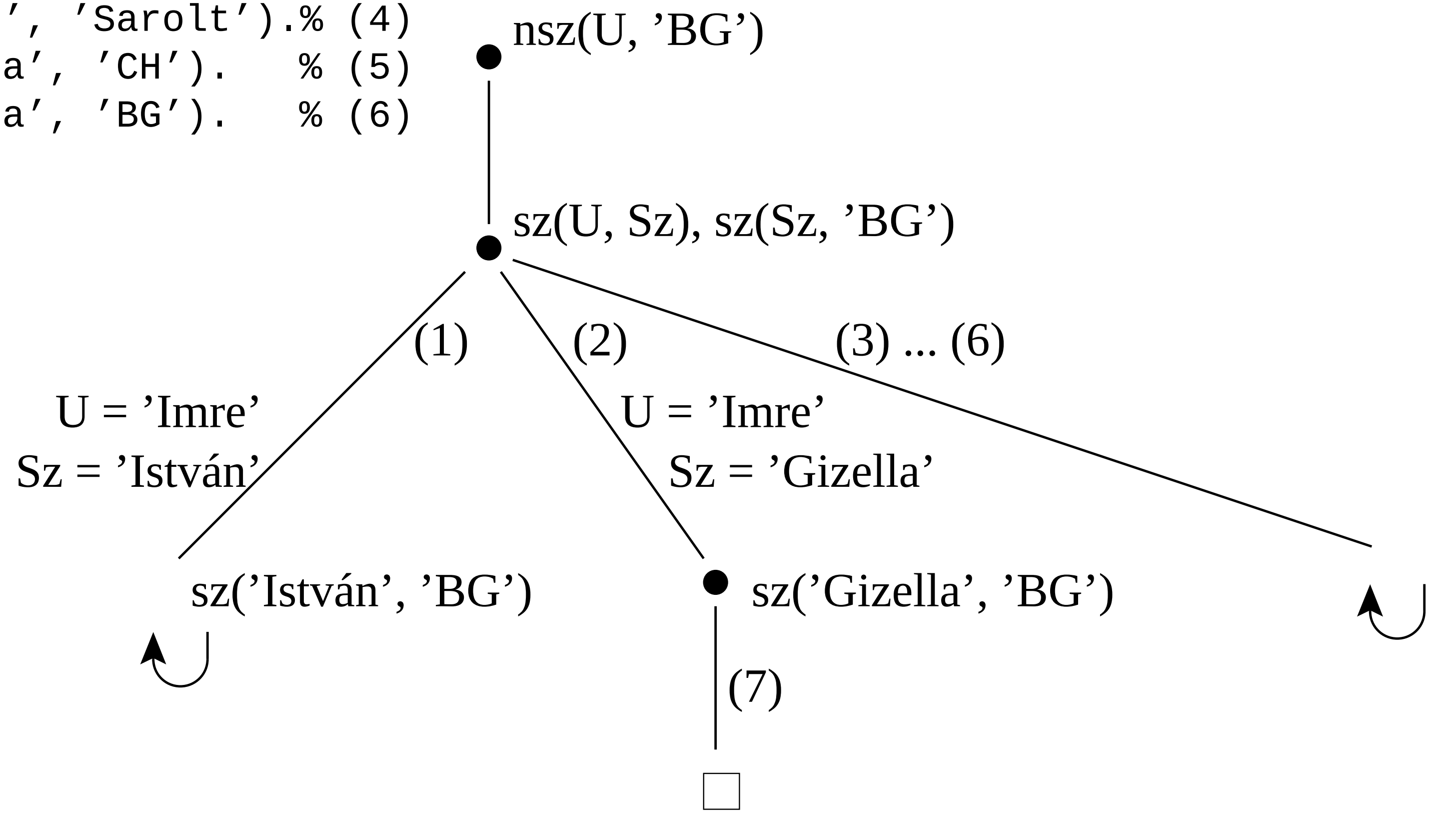
```
nsz(Gy, N) :-
    sz(Gy, Sz), sz(Sz, N).
```



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, 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.

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 $\longrightarrow$ 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** 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 hívása 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 hívására vonatkoztatható klózok listáját. Ez indexelés nélkül a predikátum összes klóza lesz, indexelés esetén ennek egy megszűrt 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 vonatkoztatható (potenciálisan 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 minden funktorhoz elkészítjük az alkalmazható klózok listáját
  - Futáskor lényegében konstans idő alatt elő tudjuk vennie a megfelelő klózlistát
  - *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

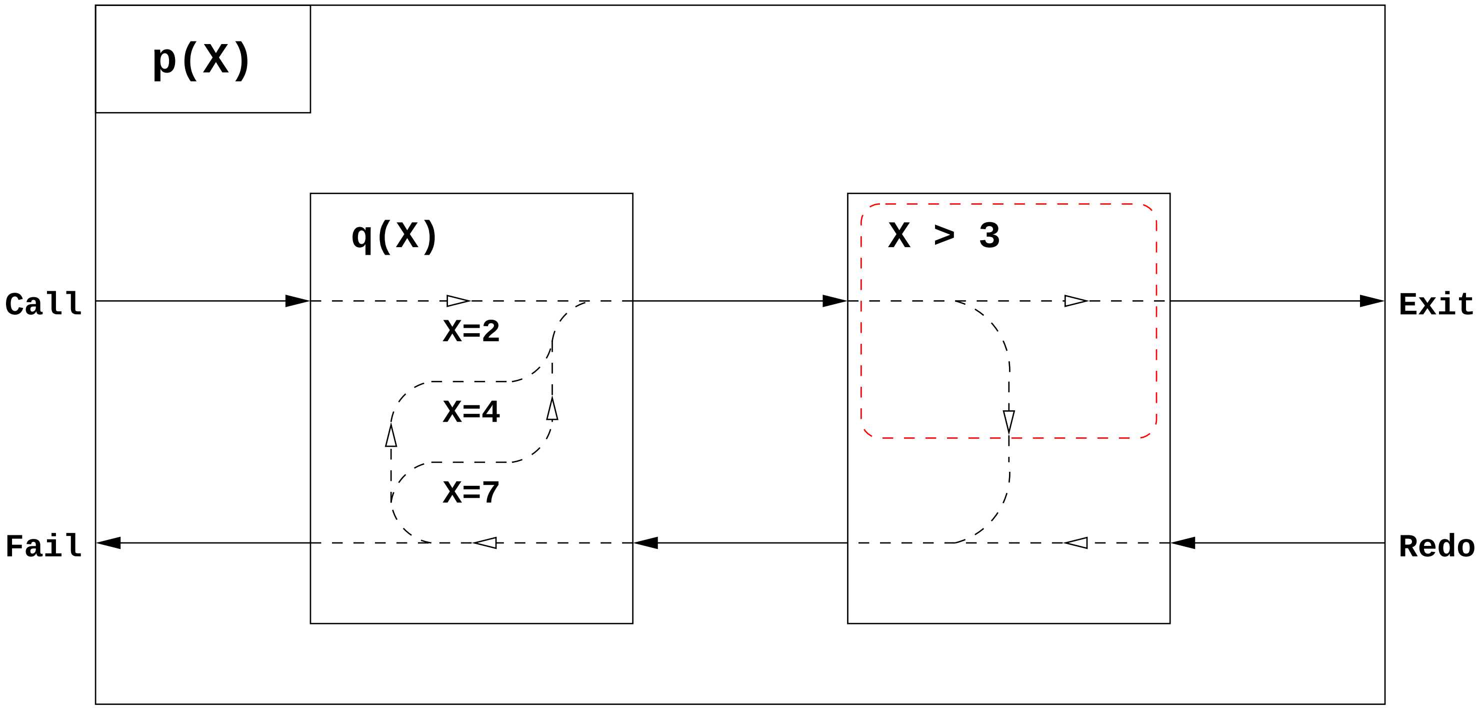
- A Prolog eljárás-végrehajtás két fázisa
  - előre menő végrehajtás: egymásba skatulyázott eljárás-belépések és - kilépések
  - visszafelé menő végrehajtás: újabb megoldás kérése egy már lefutott eljárástól
- Egy egyszerű példa

```
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 —grafi kus 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).

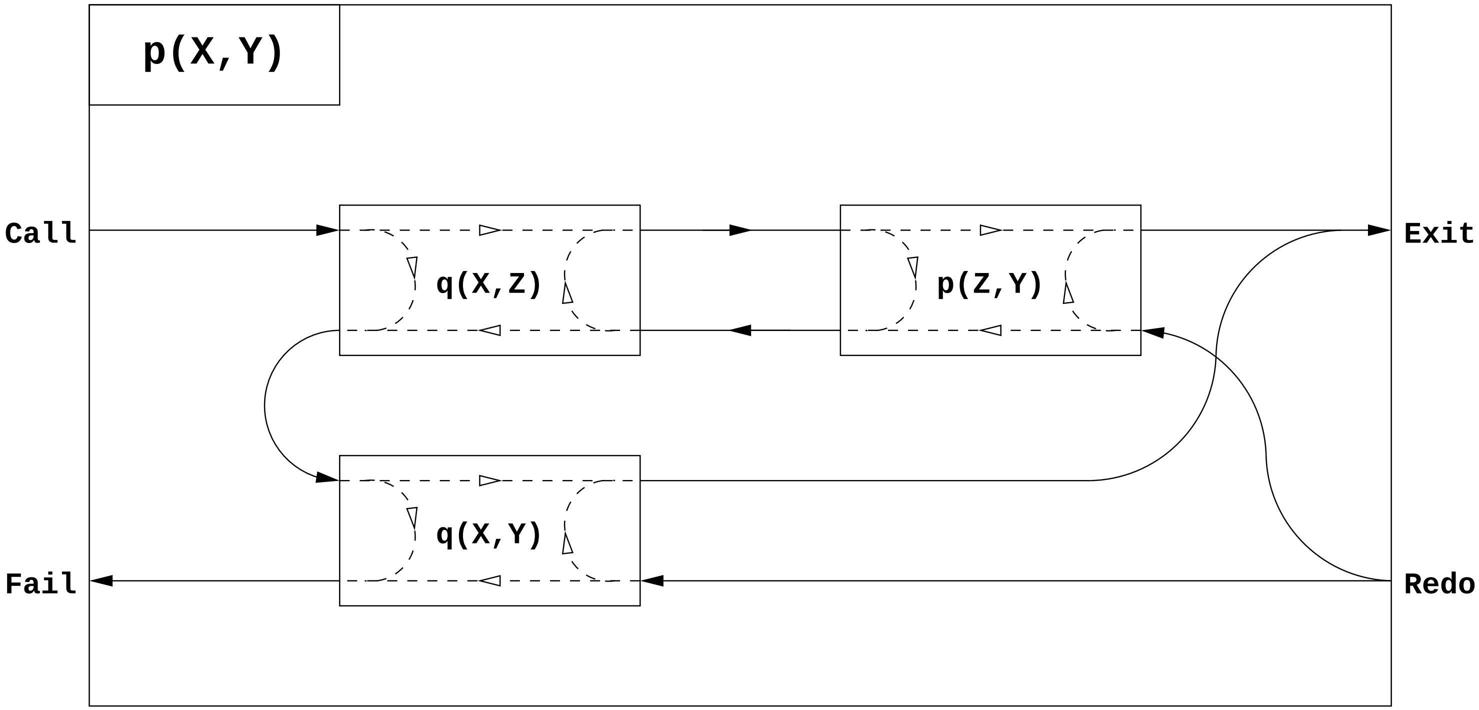
p(X) :- q(X), X > 3.


| ?- 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:

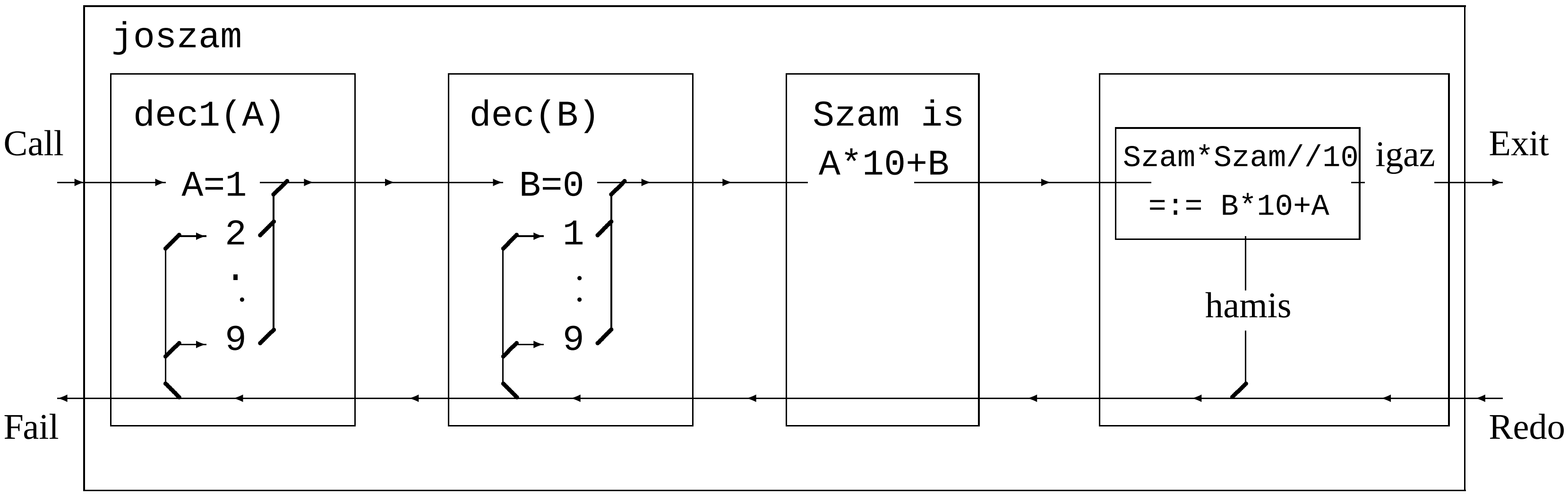
```
% dec1(J): J egy pozitív decimális számjegy.
dec1(1). dec1(2). dec1(3). dec1(4).
dec1(5). dec1(6). dec1(7). dec1(8). dec1(9).

% 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

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 meghiúsul, (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 = U.` $\equiv$ `=(U, U)`, vö. `(1)`.

A diszjunkció mint szintaktikus édesítőszer

- A diszjunkció akárhány tagú lehet. A ‘;’ művelet gyengébben köt mint a ‘,’, ezért a diszjunkciót mindig zárójelbe tesszük, mig az ágait nem kell zárójelezni. 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 egyklózossá alakítható:
  - a klózokat átalakítjuk azonos fejűvé, 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:

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

- Egy ekvivalens megoldás:

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

- Mi történik ha a két hívást megcseréljük?

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

A meghiúsulásos negáció (NF —Negation by Failure)

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

```
| ?- \+ szuloje('Imre', X).                ----> no
| ?- \+ szuloje('Géza', X).                ----> true ?
```

  - $\backslash + H$ 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: együttható 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 együtthatója 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).
E = 8 ? ;
no
```

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

Együttható 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
;   felt2 -> akkor2
;   ...
)
```

```
(   felt1 -> akkor1
;  (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

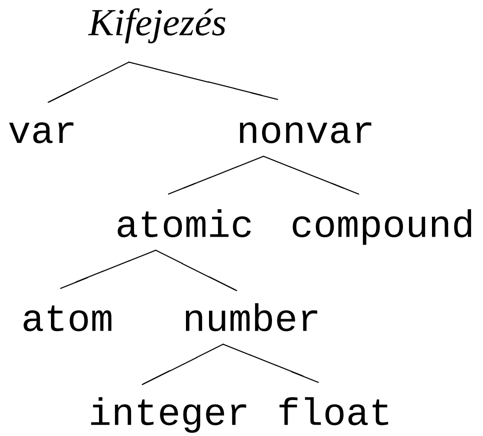
```
% Sign = sign(Num)
sign(Num, Sign) :-
        (   Num > 0 -> Sign = 1
        ;   Num < 0 -> Sign = -1
        ;   Sign = 0
        ).
```

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: $\langle$ 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: $\sigma = \{\texttt{X}\leftarrow\texttt{a}, \texttt{Y}\leftarrow\texttt{s(b,B)}, \texttt{Z}\leftarrow\texttt{C}\}$. 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: $\texttt{f(g(Z,h),A,Y)}\sigma = \texttt{f(g(C,h),A,s(b,B))}$
  - 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. $\theta = \{\texttt{X}\leftarrow\texttt{b}, \texttt{B}\leftarrow\texttt{d}\}$ esetén $\sigma \otimes \theta = \{\texttt{X}\leftarrow\texttt{a}, \texttt{Y}\leftarrow\texttt{s(b,d)}, \texttt{Z}\leftarrow\texttt{C}, \texttt{B}\leftarrow\texttt{d}\}$
- 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 $\rho = \{\texttt{A}\leftarrow 1, \texttt{Y}\leftarrow\texttt{s(Z)}\}$ 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. $\ldots$

     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`←`5`} $= \sigma_1$
    - (b.) `mgu(V`σ_1`, S)` $=$ `mgu(5, S)` (3. szerint) = {`S`←`5`} $= \sigma_2$
  - tehát $mgu(A, B) = \sigma_1 \otimes \sigma_2 =$ {`V`←`5`, `S`←`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`←`leaf(X)`} $= \sigma_1$
    - (b.) `mgu(T`σ_1`, leaf(3))` $=$ `mgu(leaf(X), leaf(3))` (4, majd 2. szerint) = {`X`←`3`} $= \sigma_2$
  - tehát $mgu(A, B) = \sigma_1 \otimes \sigma_2 =$ {`T`←`leaf(3)`, `X`←`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(...))))) ?
```