A PROLOG NYELV BEMUTATÁSA

A Prolog/LP rövid történeti áttekintése

1960-as évek	Tételbizonyító programok
1970-72	A logikai programozás elméleti alapjai (R A Kowalski)
1972	Az első Prolog interpreter (A Colmerauer)
1975	A második Prolog interpreter (Szeredi P)
1977	Az első Prolog fordítóprogram (D H D Warren)
1977–79	Számos kísérleti Prolog alkalmazás Magyarországon
1981	A japán 5. generációs projekt a logikai programozást választja
1982	A magyar MProlog az egyik első kereskedelmi forgalomba kerülő Prolog megvalósítás
1983	Egy új fordítási modell és absztrakt Prolog gép (WAM) megjelenése (D H D Warren)
1986	Prolog szabványosítás kezdete
1987–89	Új logikai programozási nyelvek (CLP, Gödel stb.)
1990–. . .	Prolog megjelenése párhuzamos számítógépeken Nagyhatékonyságú Prolog fordítóprogramok .....

Információk a logikai programozásról

- Prolog megvalósítások:
  - SWI Prolog: `http://www.swi-prolog.org/`
  - SICStus Prolog: `http://www.sics.se/sicstus`
  - GNU Prolog: `http://pauillac.inria.fr/~diaz/gnu-prolog/`
- Hálózati információforrások:
  - The WWW Virtual Library: Logic Programming:
    `http://www.afm.sbu.ac.uk/logic-prog`
  - CMU Prolog Repository:
    (a `http://www.cs.cmu.edu/afs/cs/project/ai-repository/ai/lang/prolog/` címen belül)
    - Főlap: `0.html`
    - Prolog FAQ: `faq/prolog.faq`
    - Prolog Resource Guide: `faq/prg_1.faq`, `faq/prg_2.faq`

Magyar nyelvű Prolog irodalom

Farkas Zsuzsa, Futó Iván, Langer Tamás, Szeredi Péter:
Az MProlog programozási nyelv.
Műszaki Könyvkiadó, 1989
jó bevezetés, sajnos az MProlog beépített eljárásai nem szabványosak.

Márkusz Zsuzsa: Prologban programozni könnyű.
Novotrade, 1988
mint fent

Futó Iván (szerk.): Mesterséges intelligencia. (9.2 fejezet, Szeredi Péter)
Aula Kiadó, 1999
csak egy rövid fejezet a Prologról

Peter Flach: Logikai Programozás. Az intelligens következtetés példákon keresztül.
Panem — John Wiley & Sons, 2001
jó áttekintés, inkább elméleti érdeklődésű olvasók számára

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:

$\langle$ Prolog program $\rangle$	::=	$\langle$ predikátum $\rangle$ . . .	
$\langle$ predikátum $\rangle$	::=	$\langle$ klóz $\rangle$ . . .	{azonos funktorú}
$\langle$ klóz $\rangle$	::=	$\langle$ tényállítás $\rangle$.␣ |	
		$\langle$ szabály $\rangle$.␣	{klóz funktora = fej funktora}
$\langle$ tényállítás $\rangle$	::=	$\langle$ fej $\rangle$	
$\langle$ szabály $\rangle$	::=	$\langle$ fej $\rangle$:- $\langle$ törzs $\rangle$	
$\langle$ törzs $\rangle$	::=	$\langle$ cél $\rangle$, . . .	
$\langle$ cél $\rangle$	::=	$\langle$ kifejezés $\rangle$	
$\langle$ fej $\rangle$	::=	$\langle$ kifejezés $\rangle$	

Prolog kifejezések

- Példa — egy klózfej mint kifejezés:

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

- Szintaxis:

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

Lexikai elemek

- Példák:

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

- Szintaxis:

⟨ változó ⟩ ::= ⟨ nagybetű ⟩⟨ alfanumerikus jel ⟩... |
_ ⟨ alfanumerikus jel ⟩... |

⟨ névkonstans ⟩ ::= ' ⟨ idézett karakter kar ⟩... ' |
⟨ kisbetű ⟩⟨ alfanumerikus jel ⟩... |
⟨ tapadó jel ⟩... | ! | ; | [] | {}

⟨ egész szám ⟩ ::= {előjeles vagy előjeltelen számjegysorozat}

⟨ lebegőpontos szám ⟩ ::= {belsejében tizedespontot tartalmazó számjegysorozat esetleges exponenssel}

⟨ idézett karakter ⟩ ::= {tetszőleges nem ' és nem \ karakter} | \ ⟨ escape szekvencia ⟩

⟨ alfanumerikus jel ⟩ ::= ⟨ kisbetű ⟩ | ⟨ nagybetű ⟩ | ⟨ számjegy ⟩ | _

⟨ tapadó jel ⟩ ::= + | - | * | / | \ | $ | ^ | < | > | = | ` | ~ | : | . | ? | @ | # | &

Szintaktikus édesítőszer: operátorok

- Példa:

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

- Operátoros kifejezések

  ⟨ összetett kifejezés ⟩ ::=

⟨ struktúranév ⟩ (⟨ argumentum ⟩, . . .)	{eddig csak ez volt}
| ⟨ argumentum ⟩ ⟨ operátornév ⟩ ⟨ argumentum ⟩	{infix kifejezés}
| ⟨ operátornév ⟩ ⟨ argumentum ⟩	{prefix kifejezés}
| ⟨ argumentum ⟩ ⟨ operátornév ⟩	{posztfix kifejezés}
⟨ operátornév ⟩ ::= ⟨ struktúranév ⟩	{ha operátorként lett definiálva}

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

Szabványos, beépített operátorok

Szabványos operátorok

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

Egyéb beépített operátorok

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

Operátorok jellemzői

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

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

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

Operátorok: zárójelezés

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

Operátorok — kiegészítő megjegyzések

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

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

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

- A „vessző” kettős szerepe
  - struktúra-kifejezés argumentumait választja el
  - 1000 prioritású `xfy` operátorként működik: pl. `(p :- a,b,c) = :-(p,','(a,','(b,c)))`
  - a „pucér” vessző (`,`) nem atom, 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 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.

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
```

A PROLOG VÉGREHAJTÁSI MECHANIZMUSA

A Prolog szemléletmódjai

- A Prolog nyelv terminológiája többféle szemléletből, értelmezésből származik.

Logikai (tételbizonyítási)	Célvezérelt keresési	Procedurális (eljárásszervezési)
predikátum		eljárás
klóz	szabály, tényállítás	(eljárás-változat)
(implikáció következménye) (pozitív literál)	(eljárás/klóz)fej	
(implikáció előfeltételei)	(klóz)törzs	(eljárás)törzs
(negatív literálok)	célsorozat	(eljárás)hívások
(negatív literál)	cél	(eljárás)hívás

A Prolog végrehajtás alapelemei

- (Ismétlés:) A Prolog végrehajtási mechanizmusának különböző szemléletei:
  - SLD rezolúciós tételbizonyítási folyamat
  - Cél-redukciós következtetési módszer
  - Mintaillesztésen és visszalépéses eljárásszervezésen alapuló program-végrehajtás
- A Prolog eljárásos szemlélete
  - Prolog program: eljárások gyűjteménye
  - Prolog eljárás: egy vagy több eljárás-változatból (azonos funktorú klózból) áll
  - Prolog klóz (eljárás-változat): a klózfej tartalmazza az eljárás nevét és a „formális paramétereket”, a klóztörzsben az eljáráshívásokban a meghívandó eljárások neve és az „aktuális paraméterek” szerepel.
- Prolog eljárásvégrehajtási modellek
  - Redukciós modell: makró-szerűen végrehajtandó eljáráshívási lépések (= redukciós lépések) sorozata, zsákutca esetén visszalépéssel — ez a korábbi cél redukciós modell pontosítása
  - 4-kapus doboz modell: többszörös eredményt szolgáltató eljárások meghívása, sikeres lefutása, új eredmény kérése, eljárás meghiúsulása — ez a beépített nyomkövető modellje.
  - Mindkét modellben az eljáráshívási lépés mintaillesztésen (egyesítésen) alapul

A végrehajtási modellek közös eleme: az egyesítés

- Két kifejezés (pl. egy eljáráshívás és egy klózfej) azonos alakra hozása, változók 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: `fa_levele(leaf(5), Sum)`
    fej: `fa_levele(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 ($Dom(\sigma \otimes \theta) = Dom(\sigma) \cup Dom(\theta)$):

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

    - 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. $\ldots$

   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 = `fa_levele(leaf(V), V)`, B = `fa_levele(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

- (Ismétlés:) Az `= /2` beépített eljárás egyesíti a két argumentumát
- 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.
        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(...))))) ?
```

A redukciós végrehajtási modell

- A redukciós végrehajtási modell alapgondolata
  - A végrehajtás egy állapota: egy célsorozat
  - A végrehajtás kétféle lépésből áll:
    - redukciós lépés: egy célsorozat + klóz $\rightarrow$ új célsorozat
    - zsákutca esetén visszalépés: visszatérés a legutolsó választási ponthoz
  - Választási pont:
    - 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)

Prolog végrehajtási példa — redukciós nyomkövető

```
fa_levele(leaf(V), V).                                        % (1)
fa_levele(node(L,R), V) :- fa_levele(L, V).                   % (2)
fa_levele(node(L,R), V) :- fa_levele(R, V).                   % (3)

proba(X) :-
  fa_levele(node(leaf(3),node(leaf(5),leaf(2))), X), X > 4.   % (1)

% Kezdeti célsorozat:
G0:   proba(X) ?
|     % Redukciós lépés a proba/1 eljárás egyetlen klózával
|                                        (1) X_1 = X
G1:   fa_levele(node(leaf(3),node(leaf(5),leaf(2))),X), X>4 ?
|                                        (2) L_2 = leaf(3), R_2 = node(leaf(5),leaf(2)), V_2 = X
|-----G2:   fa_levele(leaf(3),X), X>4 ?
|     |                                  (1) X = 3, V_3 = 3
|     G3:   3>4 ?
|     % Meghiúsulás esetén visszatérünk egy korábbi állapotba (itt G1)
|     |<<<< Failing back to goal G1
|                                        (3) L_5 = leaf(3), R_5 = node(leaf(5),leaf(2)), V_5 = X
G5:   fa_levele(node(leaf(5),leaf(2)),X), X>4 ?
|                                           (2) L_6 = leaf(5), R_6 = leaf(2), V_6 = X
|-----G6:   fa_levele(leaf(5),X), X>4 ?
|     |                                  (1) X = 5, V_7 = 5
|     G7:   5>4 ?
|     % Redukciós lépés beépített eljárással (BIP = built-in predicate)
|     | BIP
|     G8:   [] ?
|     |++++ Solution: X = 5 ?
```

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

- 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

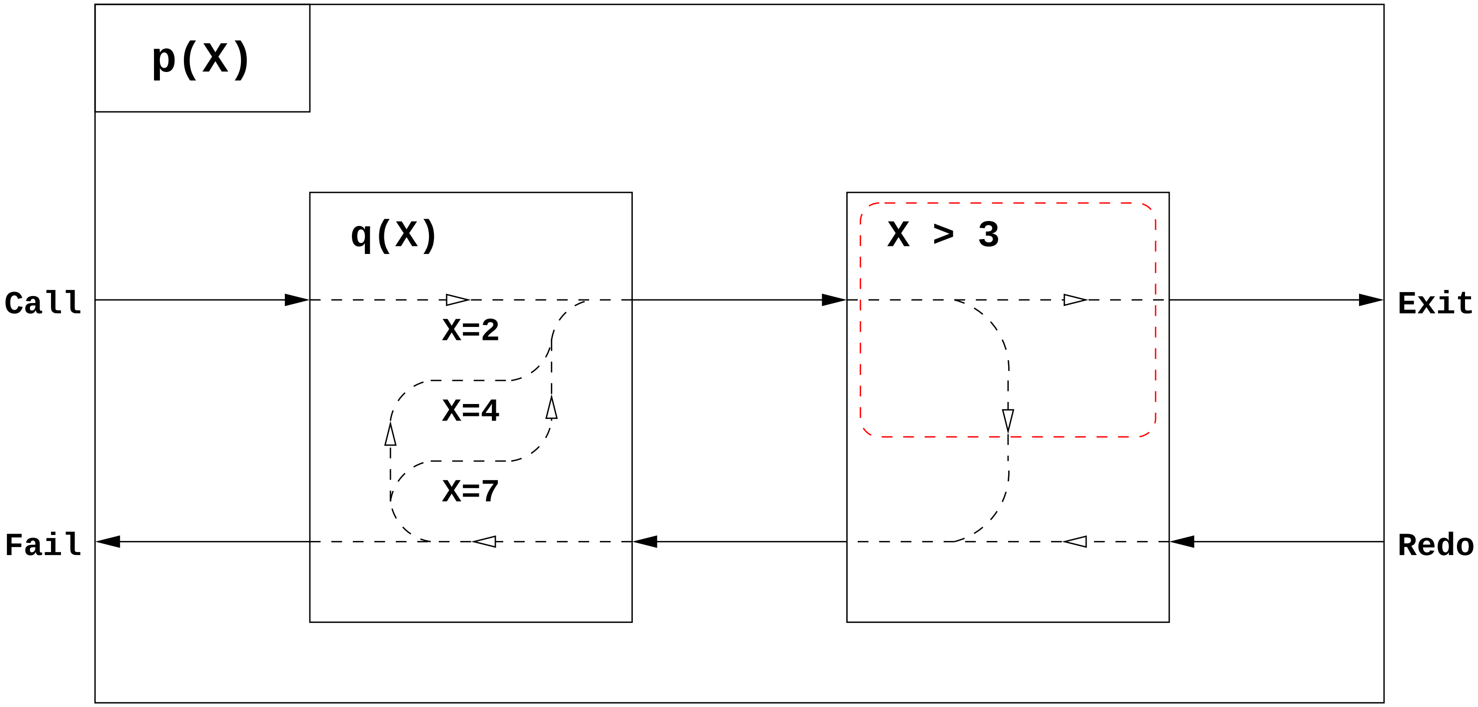
- A Prolog eljárás-végrehajtás két fázisa
  - előre menő végrehajtás: egymásba skatulyázott eljárás-belépések és -kilépések
  - visszafelé menő végrehajtás: újabb megoldás kérése egy már lefutott eljárástól
- Egy egyszerű példa

```
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.
```



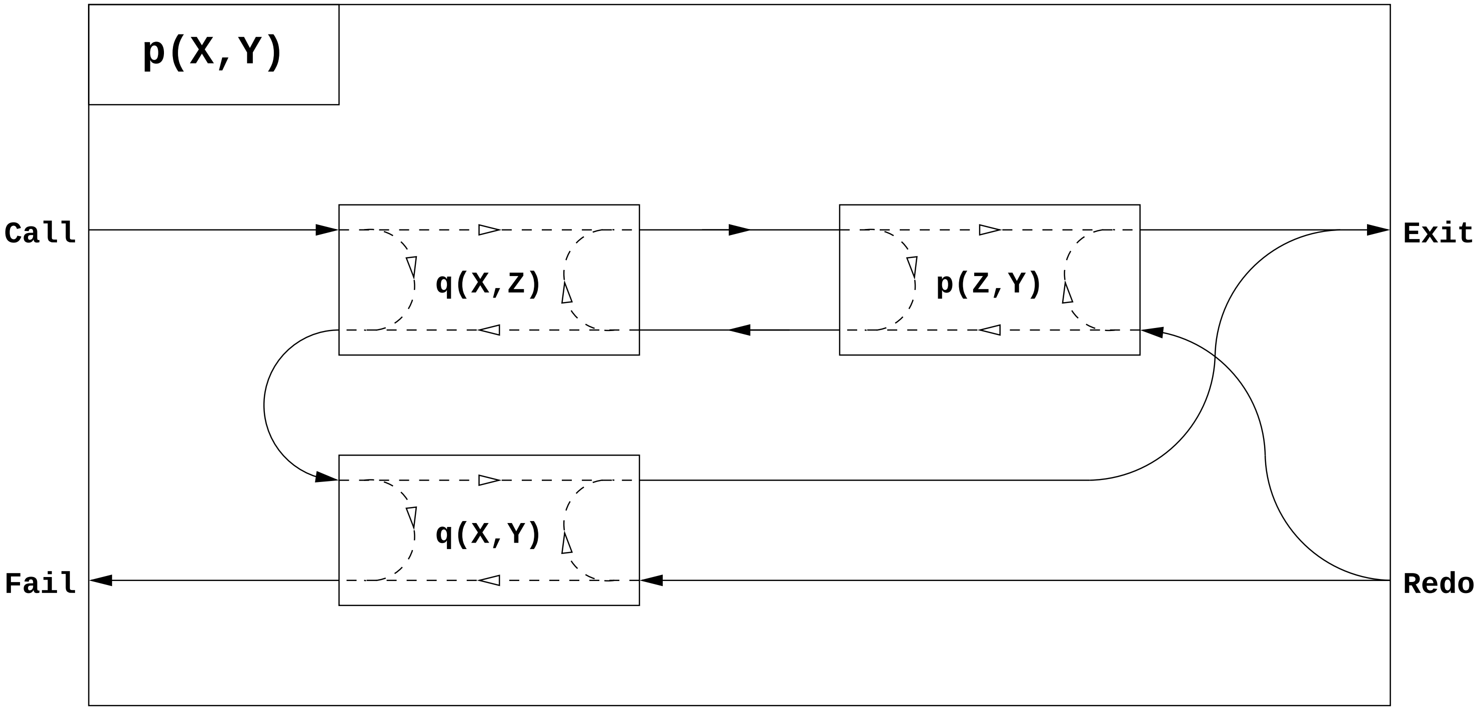
Eljárás-doboz modell — egyszerű nyomkövetési példa

- 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: `pp/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.
```