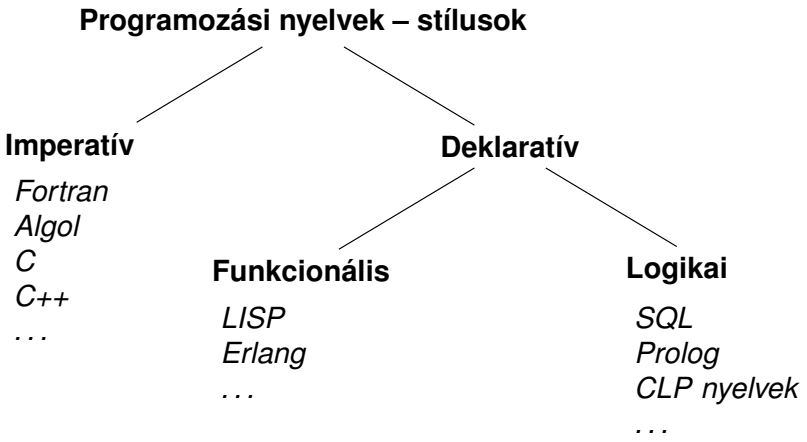
III. rész

Prolog alapok

Programozási nyelvek osztályozása



Deklaratív programozási nyelvek

- A funkcionális nyelvek alapja a matematika függvényfogalma
- A logikai nyelvek alapja a matematika relációfogalma
- Közös tulajdonságaik
  - Deklaratív szemantika – a program jelentése egy matematikai állításként olvasható ki.
  - Deklaratív változó ≡ matematikai változó – *egy* ismeretlen értéket jelöl, vö. egyszeres értékadás
- Jelmondat
  - MIT és nem HOGYAN (WHAT rather than HOW): a *megoldás módja* helyett inkább a megoldandó *feladat leírását* kell megadni
  - A gyakorlatban mindkét szemponttal foglalkozni kell – kettős szemantika:
    - deklaratív szemantika – MIT (milyen feladatot) old meg a program;
    - procedurális szemantika – HOGYAN oldja meg a program a feladatot.

A logikai programozás alapgondolata

- Logikai programozás (LP):
  - Programozás a matematikai logika segítségével
    - egy logikai program nem más mint **logikai állítások halmaza**
    - egy logikai **program futása** nem más mint **következtetési folyamat**
  - De: a logikai következtetés óriási keresési tér bejárását jelenti
    - szorítsuk meg a logika nyelvét
    - válasszunk egyszerű, ember által is követhető következtetési algoritmusokat
  - Az LP máig legelterjedtebb megvalósítása a **Prolog** = **Pro**gramozás **log**ikában (**Pro**gramming in **log**ic)
    - az elsőrendű logika egy erősen megszorított résznyelve az ún. **definit-** vagy **Horn-klóz**ok nyelve,
    - végrehajtási mechanizmusa: **mintaillesztés**es eljáráshíváson alapuló **visszalépés**es keresés.

Az előadás LP részének áttekintése

- **1. blokk:** A Prolog nyelv alapjai
  - Szintaxis
  - Deklaratív szemantika
  - Procedurális szemantika (végrehajtási mechanizmus)
- **2. blokk:** Prolog programozási módszerek
  - A legfontosabb beépített eljárások
  - Fejlettebb nyelvi és rendszerelemek
- Kitekintés: Új irányzatok a logikai programozásban

A Prolog/LP rövid történeti áttekintése

1960-as évek	Első 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 .....

Tartalom

Néhány alapvető példafeladat

- Tényállítások, szabályok: *családi kapcsolatok*
- Adatstruktúrák: *bináris fák*
- Aritmetika: *faktoriális*
- Szimbolikus feldolgozás: *deriválás*

A Prolog alapelemei: a családi kapcsolatok példája

- Adatok

  Adottak gyerek–szülő kapcsolatra vonatkozó állítások, pl.

gyerek	*szülő*
Imre	*István*
Imre	*Gizella*
István	*Géza*
István	*Sarolta*
Gizella	*Civakodó Henrik*
Gizella	*Burgundi Gizella*

- A feladat:
  - Definiálandó az unoka–nagyszülő kapcsolat, pl. keressük egy adott személy nagyszüleit.

A nagyszülő feladat — Prolog megoldás

```
% szuloje(Gy, Sz):Gy szülője Sz.
% Tényállításokból álló predikátum
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)

%  Gyerek nagyszülője Nagyszulo.
% Egyetlen szabályból álló predikátum
nagyszuloje(Gyerek, Nagyszulo) :-
    szuloje(Gyerek, Szulo),
    szuloje(Szulo, Nagyszulo). % (nsz1)
```

```
% Kik Imre nagyszülei?
| ?- nagyszuloje('Imre', NSz).
NSz = 'Géza' ? ;
NSz = 'Sarolt' ? ;
NSz = 'Civakodó Henrik' ? ;
NSz = 'Burgundi Gizella' ? ;
no
% Kik Géza unokái?
| ?- nagyszuloje(U, 'Géza').
U = 'Imre' ? ;
no
```


Deklaratív szemantika – klózok logikai alakja

- A **szabály** jelentése implikáció: a törzsbeli célok **konjunkciójából** következik a fej.
  - Példa: `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:
    $\forall UN \quad (\texttt{nagyszuloje}(U,N) \leftarrow \exists Sz(\texttt{szuloje}(U,Sz) \wedge \texttt{szuloje}(Sz,N)))$
- A **tényállítás** feltétel nélküli állítás, pl.
  - Példa: `szuloje('Imre', 'István').`
  - Logikai alakja változatlan
  - A tényállításban is lehetnek változók, ezeket is univerzálisan kell kvantálni

A Prolog deklaratív szemantikája

- **Példány**: Egy kifejezésből változók behelyettesítésével előálló kifejezés
- Egy célsorozat lefutása **sikeres**, ha a célsorozat 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.

A Prolog végrehajtási mechanizmusa (procedurális szemantika)

- Egy eljárás: azon klózok összesége, amelyek fejének neve és argumentumszáma azonos.
- Egy klóz: `Fej :- Törzs`, ahol `Törzs` egy célsorozat
- Egy célsorozat: C_1, ..., C_n, célok (eljáráshívások) sorozata, $n \geq 0$
- Végrehajtás: adott egy program és egy futtatandó célsorozat
- Redukciós lépés:
  - a célsorozat *első* tagjához keresünk egy vele *egyesíthető* klózfejet,
  - az egyesítéshez szükséges *változó-behelyettesítéseket* elvégezzük,
  - az első célt helyettesítjük az adott klóz törzsével
- Egyesítés: két Prolog kifejezés azonos alakra hozása változók behelyettesítésével, a lehető legáltalánosabb módon
- Keresés:
  - a redukciós lépésben a klózokat a felírás sorrendjében (felülről lefele) nézzük végig,
  - ha egy cél több klózfejjel is egyesíthető, akkor a Prolog *minden* lehetséges redukciós lépést megpróbál (meghiúsulás, visszalépés esetén)

Adatstruktúrák Prologban – a bináris fák példája

- A bináris fa adatstruktúra
  - vagy egy csomópont (`node`), amelynek két részfája van (`left`,`right`)
  - vagy egy levél (`leaf`), amely egy egészt tartalmaz
- Binárisfa-struktúrák különböző nyelveken

```
% Struktúra deklarációk C-ben
enum treetype {Node, Leaf};
struct tree {
  enum treetype type;
  union {
    struct { struct tree *left;
             struct tree *right;
           } node;
    struct { int value;
           } leaf;
  } u;
};
```

```
% Adattípus-leírás Prologban
% (ún. Mercury jelölés):

% :- type tree --->
%        node(tree, tree)
%      | leaf(int).
```


Bináris fák összegzése

- Egy bináris fa levélösszegének kiszámítása:
  - csomópont esetén a két részfa levélösszegének összege
  - levél esetén a levélben tárolt egész

```
% C nyelvű (deklaratív) függvény
int tree_sum(struct tree *tree)
{
  switch(tree->type) {
  case Leaf:
   return tree->u.leaf.value;
  case Node:
   return
    tree_sum(tree->u.node.left) +
    tree_sum(tree->u.node.right);
    }
}
```

```
% Prolog eljárás (predikátum)
% tree_sum(Tree, S): A Tree fa
% leveleiben levő számok
% összege S
tree_sum(leaf(Value), Value).
tree_sum(node(Left,Right), S) :-
        tree_sum(Left, S1),
        tree_sum(Right, S2),
        S is S1+S2.
```


Aritmetika Prologban – faktoriális

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

Klasszikus szimbolikuskifejezés-feldolgozás: deriválás

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

```
% deriv(Kif, D):  Kif-nek az x szerinti deriváltja D.
deriv(x, 1).
deriv(C, 0) :-                    number(C).
deriv(U+V, DU+DV) :-              deriv(U, DU), deriv(V, DV).
deriv(U-V, DU-DV) :-              deriv(U, DU), deriv(V, DV).
deriv(U*V, DU*V + U*DV) :-        deriv(U, DU), deriv(V, DV).
| ?- deriv(x*x+x, D).
  ⟹  D = 1*x+x*1+1 ? ; no

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

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

| ?- deriv(I, 0).
  ⟹  no
```


A Prolog adatfogalma, a Prolog kifejezés

- konstans (`atomic`)
  - számkonstans (`number`) – egész vagy lebegőp, pl. `1`, `-2.3`, `3.0e10`
  - névkonstans (`atom`), pl. `'István'`, `szuloje`, `+`, `-`, `<`, `tree_sum`
- összetett- vagy struktúra-kifejezés (`compound`)
  - ún. kanonikus alak: ⟨ struktúranév ⟩(⟨ arg_1 ⟩, . . . , ⟨ arg_n ⟩)
    - a ⟨ struktúranév ⟩ egy névkonstans, az ⟨ arg_i ⟩ argumentumok tetszőleges Prolog kifejezések
    - példák: `leaf(1)`, `person(william,smith,2003)`, `<(X,Y)`, `is(X, +(Y,1))`
  - szintaktikus „édesítőszerek”, pl. operátorok: `X is Y+1` ≡ `is(X, +(Y,1))`
- változó (`var`)
  - 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)

Néhány beépített predikátum

- Kifejezések egyesítése
  - `X = Y`: az `X` és `Y` **szimbolikus** kifejezések változók behelyettesítésével azonos alakra hozhatók
  - `X \= Y`: az `X` és `Y` kifejezések nem hozhatók azonos alakra
- Aritmetikai predikátumok
  - `X is Kif`: A `Kif` **aritmetikai** kifejezés **értékét** egyesíti `X`-szel.
  - `Kif1<Kif2`: `Kif1` **aritmetikai értéke** kisebb `Kif2` értékénél.
  - Hasonlóan: `Kif1=<Kif2`, `Kif1>Kif2`, `Kif1>=Kif2`, `Kif1=:=Kif2` (aritmetikailag egyenlő), `Kif1=\=Kif2` (aritmetikailag nem egyenlő)
  - Fontos aritmetikai operátorok: `+, -, *, /, rem, //` (egész-osztás)
- További hasznos predikátumok
  - `true, fail`: Mindig sikerül ill. mindig meghiúsul.
  - `write(X)`: Az `X` Prolog kifejezést kiírja, `nl`: kiír egy újsort.
  - `trace, notrace`: A (teljes) nyomkövetést be- ill. kikapcsolja.

Programfejlesztési beépített predikátumok

- `consult(File)`: A `File` állományban levő programot beolvassa és értelmezendő alakban eltárolja. (`File = user` ⇒ terminálról olvas.)
- `listing` vagy `listing(Predikátum)`: Az értelmezendő alakban eltárolt összes ill. adott nevű predikátumokat kilistázza.
- `compile(File)` vagy `[File]`: A `File` állományban levő programot beolvassa, lefordítja.
- A lefordított alak gyorsabb, de nem listázható, kevésbé nyomkövethető.
- `halt`: A Prolog rendszer befejezi működését.

```
> sicstus
SICStus 4.1.2 (x86-linux-glibc2.7): Wed Apr 28 22:42:37 CEST 2010
| ?- consult(deriv).
% consulted /home/user/szulok.pl in module user, 0 msec 376 bytes
yes
| ?- deriv(x*x+x, D).
D = 1*x+x*1+1 ? ;
no
| ?- listing(deriv).
(...)
yes
| ?- halt.
>
```

Tartalom

Az eljárás-redukciós végrehajtási modell

- A Prolog végrehajtás:
  - egy adott célsorozat futtatása egy adott programra vonatkozóan,
  - eredménye lehet:
    - siker – változó-behelyettesítésekkel
    - meghiúsulás (változó-behelyettesítések nélkül)
- 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 és a **további** (eddig nem próbált) klózokkal próbálunk redukciós lépést
- A végrehajtási algoritmus leírásában használt adatstruktúrák:
  - `CS` – célsorozat
  - egy verem, melynek elemei `<CS,I>` alakú párok – ahol `CS` egy célsorozat, `I` a célsorozat első céljának redukálásához használt legutolsó klóz sorszáma.
  - a verem a visszalépést szolgálja: minden választási pontnál létrehozunk egy `<CS,I>` párt

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 ⇒meghiúsulás
  - egy beépített eljárás segítségével (az első cél beépített eljárást hív):
    - A célsorozatot szétbontjuk az első hívásra és a maradékra.
    - A beépített eljáráshívást végrehajtjuk
    - Siker esetén a behelyettesítéseket elvégezzük a célsorozat maradékán ez lesz az új célsorozat
    - Ha a beépített eljárás hívása sikertelen⇒meghiúsulás

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 alkalmazható 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` ⇒ 4a. 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 alkalmazható (illeszthető fejű) 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 besoroljuk).
- Az indexelés megvalósítása:
  - Fordításkor minden funktor ⇒az alkalmazható klózok listája
  - Futáskor konstans idő alatt elő tudjuk vennie a megfelelő klózlistát
  - *Fontos:* ha egyelemű a lista, 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)

A faösszegző program változatai

- A bináris fákat összegző predikátum adott fa levélösszegét állítja elő
- Egy `tree_sum(T, 3)` hívás hibát jelez a `3 is S1+S2` hívásnál.
- Az `is` beépített eljárás helyett egy saját `plus` eljárást használva egészek korlátos tartományon megoldható a kétirányú működés.

```
% plus(A, B, C): A+B=C, ahol 0 < A,B,C =< 4 egész számok,
plus(1, 1, 2).     plus(1, 2, 3).     plus(2, 1, 3).

% tree_sum(Tree, S): A Tree fa leveleiben levő számok összege S.
```

```
% tree_sum(+Tree, ?S):
tree_sum(leaf(Value), Value).
tree_sum(node(Left,Right), S) :-
        tree_sum(Left, SL),
        tree_sum(Right, SR),
        S is SL+SR.
```

```
% tree_sum(?Tree, ?S):
tree_sum(leaf(Value), Value).
tree_sum(node(Left,Right), S) :-
        plus(SL, SR, S),
        tree_sum(Left, SL),
        tree_sum(Right, SR).
```

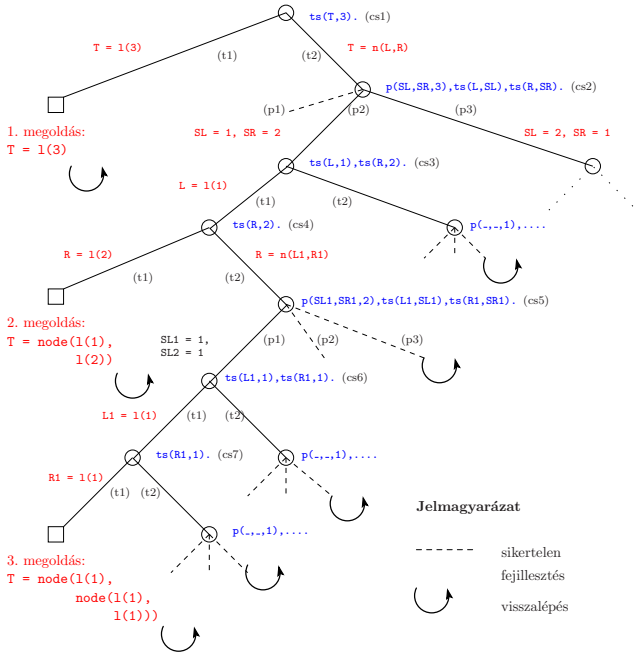
- Az argumentumok input-output módjának jelölése:
  `+` – bemenő (nem változó), `-` – kimenő (változó), `?` – tetszőleges

A faösszegző program keresési tere

A program:

```
p(1, 1, 2).    (p1)
p(1, 2, 3).    (p2)
p(2, 1, 3).    (p3)

ts(l(V), V).   (t1)
ts(n(L,R), S) :-
   p(SL,SR,S),
   ts(L, SL),
   ts(R, SR).  (t2)
```



A verem a 3. megoldásnál:

⟨(cs2), 2⟩
⟨(cs3), 1⟩
⟨(cs5), 1⟩
⟨(cs6), 1⟩
⟨(cs7), 1⟩

Tartalom

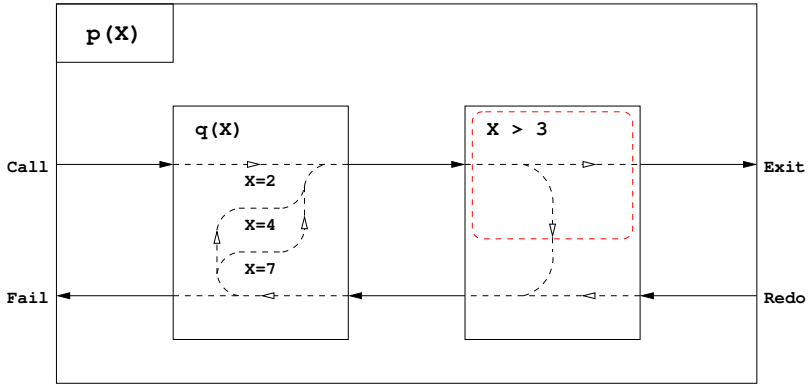
A Prolog nyomkövető által használt eljárás-doboz modell

- A Prolog eljárás-végrehajtás két fázisa
  - előre menő: egymásba skatulyázott eljárás-belépések és -kilépések
  - visszafelé menő: újabb megoldás kérése egy már lefutott eljárástól
- Egy egyszerű példaprogram, hívása `| ?- p(X).`

  `q(2). q(4). q(7).` `p(X) :- q(X), X > 3.`
- Példafutás: belépünk a `p/1` eljárásba (Hívási kapu, Call port)
  - Belépünk a `q/1` eljárásba (Call port)
  - `q/1` sikeresen lefut, `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 újra sikeresen lefut a `q(4)` eredménnyel (Exit)
    - A `4>3` hívással a `>/2`-be belépünk majd 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

- `?...Exit` jelzi, hogy maradt választási pont a lefutott eljárásban
- Ha nincs `?` az Exit kapunál, akkor a doboz törlődik (lásd a szaggatott piros téglalapot az előző dián)

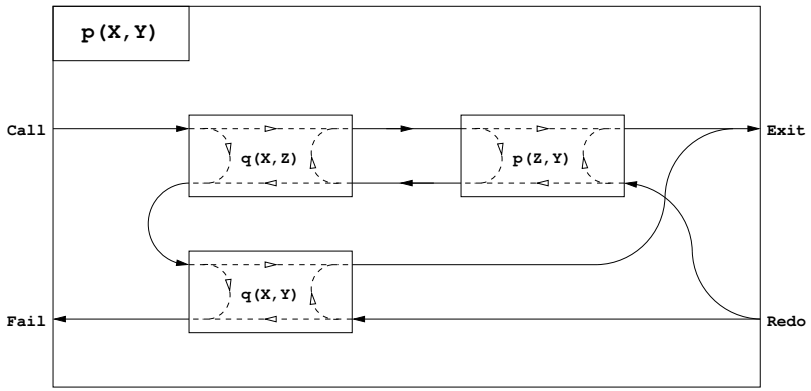
```
q(2). q(4). q(7).                p(X) :- q(X), X > 3.

| ?- trace, p(X).
        1      1 Call: p(_463) ?
        2      2 Call: q(_463) ?
?       2      2 Exit: q(2) ?      % ? ≡ maradt választási pont q-ban
        3      2 Call: 2>3 ?
        3      2 Fail: 2>3 ?                % visszafelé menő végrehajtás
        2      2 Redo: q(2) ?               % visszafelé menő végrehajtás
?       2      2 Exit: q(4) ?
        4      2 Call: 4>3 ?
        4      2 Exit: 4>3 ?       % nincs ? ⇒ a doboz törlődik (*)
?       1      1 Exit: p(4) ?
X = 4 ? ;                                   % ; ⇒ "mesterséges" meghiúsulás
        1      1 Redo: p(4) ?               % visszafelé menő végrehajtás
                                    % (*) miatt nem látjuk a Redo-Fail kapukat a 4>3 hívásra
        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

- A „szülő” eljárásdoboz és a „belső” eljárások dobozainak összekapcsolása
- Előfeldolgozás: a klózfejekben csak változók legyenek, a fej-egyesítéseket alakítsuk hívásokká, pl.
  `fakt(0,1).` ⇒`fakt(X,Y) :- X=0, Y=1.`
- Előre menő végrehajtás (balról-jobbra menő nyilak):
  - A szülő Call kapuját az 1. klóz első hívásának Call kapujára kötjük.
  - Egy belső eljárás Exit kapuját
    - a következő hívás Call kapujára, vagy,
    - ha nincs következő hívás, akkor a szülő Exit kapujára kötjük
- Visszafelé menő végrehajtás (jobbról-balra menő nyilak):
  - Egy belső eljárás Fail kapuját
    - az előző hívás Redo kapujára, vagy, ha nincs előző hívás, akkor
    - a következő klóz első hívásának Call kapujára, vagy
    - ha nincs következő klóz, akkor a szülő Fail kapujára kötjük
  - A szülő Redo kapuját mindegyik klóz utolsó hívásának Redo kapujára kötjük
    - mindig abba a klózra térünk vissza, amelyben legutoljára voltunk

SICStus nyomkövetés – legfontosabb parancsok

- Beépített eljárások
  - `trace`, `debug`, `zip` – a `c`, `l`, `z` paranccsal indítja a nyomkövetést
  - `notrace`, `nodebug`, `nozip` – kikapcsolja a nyomkövetést
  - `spy(P)`, `nospy(P)`, `nospyall` – töréspont be/ki a `P` eljárásra, ∀ ki.
- Alapvető nyomkövetési parancsok, ujsorral (<RET>) kell lezárni
  - `h` (help) – parancsok listázása
  - `c` (creep) vagy csak <RET> – lassú futás (minden kapunál megáll)
  - `l` (leap) – csak töréspontnál áll meg, de a dobozokat építi
  - `z` (zip) – csak töréspontnál áll meg, dobozokat nem épít
  - `+` ill. `-` – töréspont be/ki a kurrens eljárásra
  - `s` (skip) – eljárástörzs átlépése (Call/Redo ⇒ Exit/Fail)
  - `o` (out) – kilépés az eljárástörzsből (⇒szülő Exit/Fail kapu)
- A Prolog végrehajtást megváltoztató parancsok
  - `u` (unify) – a kurrens hívást helyettesíti egy egyesítéssel
  - `r` (retry) – újrakezdi a kurrens hívás végrehajtását (⇒Call)
- Információ-megjelenítő és egyéb parancsok
  - `<` n – a kiírási mélységet n-re állítja ($n = 0$ ⇒∞ mélység)
  - `n` (notrace) – nyomkövető kikapcsolása
  - `a` (abort) – a kurrens futás abbahagyása

Eljárás-doboz modell – OO szemléletben (kiegészítő anyag)

- Minden eljáráshoz tartozik egy osztály, amelynek van egy konstruktor függvénye (amely megkapja a hívási paramétereket) és egy `next` „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 kapujához 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, **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` C++ kódrészlet (kieg. anyag)

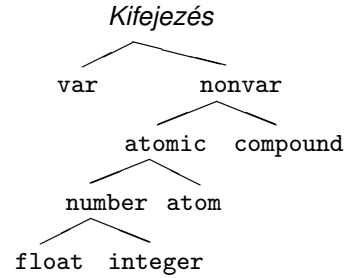
A `p/2`-nek megfeleltetett C++ osztály „következő megoldás” metódusa:

```
boolean p::next()           { // Return next solution for p/2
  switch(clno)           {
  case 0:                     // first call of the method
    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==2)
    /* 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
                         } }
```

Tartalom

A Prolog adatfogalma – ismétlés

- Prolog kifejezések osztályozása – kanonikus alak (belső ábrázolás)



`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

A Prolog alapvető adatkezelő művelete: az egyesítés

- Egyesítés (*unification*): két Prolog kifejezés (pl. egy eljáráshívás és egy klózfej) azonos alakra hozása, változók esetleges behelyettesítésével, a lehető legáltalánosabban (a legkevesebb behelyettesítéssel)
- Az egyesítés **szimmetrikus**: mindkét oldalon lehet – és behelyettesítődhet – változó
- 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'`
  - Kétirányú paraméterátadás – fej- és hívásváltozókat is behelyettesít:
    hívás: `tree_sum(leaf(5), Sum)`
    fej: `tree_sum(leaf(V), V)`
    behelyettesítés: `V = 5, Sum = 5`

Az egyesítési algoritmus feladata

- Az egyesítési algoritmus
  - bemenete: két Prolog kifejezés: *A* és *B*
    (általában egy klóz feje és egy célsorozat első tagja)
  - feladata: a két kifejezés egyesíthetőségének eldöntése
  - matematikailag az eredménye: meghiúsulás, vagy siker és a legáltalánosabb egyesítő – *most general unifier*, *mgu*(*A*, *B*) – előállítása
  - praktikusan nem az *mgu* egyesítő előállítása szükséges, hanem az egyesítő behelyettesítés végrehajtása (a szóbanforgó klóz törzsén és a célsorozat maradékán)
- A legáltalánosabb egyesítő az, amelyik nem helyettesít be „feleslegesen”
  - példa: `tree_sum(leaf(V), V) = tree_sum(T, S)`
    - egyesítő behelyettesítés: `V←1`, `T←leaf(1)`, `S←1`
    - legáltalánosabb egyesítő behelyettesítés: `T←leaf(V)`, `S←V`, vagy `T←leaf(S)`, `V←S`
  - az *mgu* – változó-átnevezéstől (pl. `V←S`) eltekintve – **egyértelmű**
  - minden egyesítő előállítható a legáltalánosabból további behelyettesítéssel, pl. `V←1` ill. `S←1`

A „praktikus” egyesítési algoritmus

1. Ha A és B azonos változók vagy konstansok, akkor kilép sikerrel, behelyettesítés nélkül
2. Egyébként, ha A változó, akkor a $\sigma = \{A \leftarrow B\}$ behelyettesítést elvégzi, és kilép sikerrel
3. Egyébként, ha B változó, akkor a $\sigma = \{B \leftarrow A\}$ behelyettesítést elvégzi, és kilép sikerrel (a 2. és 3. lépések sorrendje változhat)
4. Egyébként, ha A és B azonos nevű és argumentumszámú összetett kifejezések és argumentum-listáik $A_1,\ldots,A_N$ ill. $B_1,\ldots,B_N$, akkor
   - A_1 és B_1 egyesítését elvégzi (azaz az ehhez szükséges behelyettesítéseket végrehajtja), ha ez sikertelen, akkor kilép meghiúsulással;
   - A_2 és B_2 egyesítését elvégzi, ha ez sikertelen, akkor kilép meghiúsulással;
   - ...
   - A_N és B_N egyesítését elvégzi, ha ez sikertelen, akkor kilép meghiúsulással

   Kilép sikerrel
5. Minden más esetben kilép meghiúsulással (A és B nem egyesíthető)

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, így ciklikus kifejezések keletkezhetnek.
  - Szabványos eljárásként rendelkezésre áll: `unify_with_occurs_check/2`
  - Kiterjesztés (pl. SICStus): az előfordulás-ellenőrzés elhagyása miatt keletkező ciklikus kifejezések tisztességes kezelése.
- Példák:

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


Az egyesítési algoritmus matematikai megfogalmazása

- A *behelyettesítés* egy olyan σ függvény, amely a $Dom(\sigma)$-beli változókhoz kifejezéseket rendel. Általában posztfix jelölést használunk, pl. $X\sigma$
  - Példa: $\sigma = \{X \leftarrow a, Y \leftarrow s(b,B), Z \leftarrow C\}$, $Dom(\sigma) = \{X, Y, Z\}$, $X\sigma = a$
- A behelyettesítés-függvény természetes módon kiterjeszthető:
  - $K\sigma$: σ alkalmazása egy *tetszőleges* K kifejezésre: σ behelyettesítéseit *egyidejűleg* elvégezzük K-ban.
  - Példa: $f(g(Z,h),A,Y)\sigma = f(g(C,h),A,s(b,B))$
- Kompozíció: $\sigma \otimes \theta = \sigma$ és θ egymás utáni alkalmazása: $X(\sigma \otimes \theta) = X\sigma\theta$
  - A $\sigma \otimes \theta$ behelyettesítés az $x \in Dom(\sigma)$ változókhoz az $(x\sigma)\theta$ kifejezést, a többi $y \in Dom(\theta) \setminus Dom(\sigma)$ változóhoz $y\theta$-t rendeli ($Dom(\sigma \otimes \theta) = Dom(\sigma) \bigcup Dom(\theta)$):

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

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

Egyesítés: a legáltalánosabb egyesítő előállítása

- 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. Pl. σ_2 és σ_3 a fenti A és B legáltalánosabb egyesítője.
- **Tétel**: A legáltalánosabb egyesítő, változó-átnevezéstől eltekintve egyértelmű.

A „matematikai” egyesítési algoritmus

- Az egyesítési algoritmus
  - bemenete: két Prolog kifejezés: A és B
  - feladata: a két kifejezés egyesíthetőségének eldöntése
  - eredménye: siker esetén az $mgu(A, B)$ legáltalánosabb egyesítő
- A rekurzív egyesítési algoritmus $\sigma = mgu(A, B)$ előállítására
  1. Ha A és B azonos változók vagy konstansok, akkor $\sigma = \{\}$ (üres).
  2. Egyébként, ha A változó, akkor $\sigma = \{A \leftarrow B\}$.
  3. Egyébként, ha B változó, akkor $\sigma = \{B \leftarrow A\}$.
     (A (2) és (3) lépések sorrendje felcserélődhet.)
  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 = `tree_sum(leaf(V), V)`, B = `tree_sum(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`$\}$

Tartalom

Predikátumok, klózok

- Példa:

```
% két klózból álló predikátum definíciója, funktora: tree_sum/2
tree_sum(leaf(Val), Val).        %                  1. klóz, tényáll.
tree_sum(node(Left,Right), S) :- %          fej    \
        tree_sum(Left, S1),      % cél     \       |
        tree_sum(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

- Megjegyzések (comment)
  - A % százalékjeltől a sor végéig
  - A /* jelpártól a legközelebbi */ jelpárig.
- Formázó elemek (komment, szóköz, újsor, tabulátor stb.) szabadon használhatók
  - kivétel: struktúrakifejezés neve után tilos formázó elemet tenni;
  - prefix operátor (ld. később) és „(” között kötelező a formázó elem;
  - klózt lezáró pont (␣): önmagában álló pont (előtte nem tapadó jel áll) amit egy formázó elem követ
- Programok javasolt formázása:
  - Az egy predikátumhoz tartozó klózok legyenek egymás mellett a programban, közéjük ne tegyünk üres sort.
  - A predikátum elé tegyünk egy üres sort és egy fejkommentet:

```
% predikátumnév(A1, ..., An): A1, ..., An közötti
% összefüggést leíró kijelentő mondat.
```

  - A klózfejet írjuk sor elejére, minden célt lehetőleg külön sorba, néhány szóközzel beljebb kezdve

Prolog kifejezések

- Példa – egy klózfej mint kifejezés:

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

- Szintaxis:

⟨ kifejezés ⟩ ::= ⟨ változó ⟩ | {Nincs funktora}
⟨ konstans ⟩ | {Funktora: ⟨ konstans ⟩/0}
⟨ összetett kif. ⟩ | {Funktor: ⟨ struktúranév ⟩/⟨ arg.sz. ⟩}
⟨ egyéb kifejezés ⟩ {Operátoros, lista, stb.}
⟨ konstans ⟩ ::= ⟨ névkonstans ⟩ |
⟨ számkonstans ⟩
⟨ számkonstans ⟩ ::= ⟨ egész szám ⟩ |
⟨ lebegőp. szám ⟩
⟨ összetett kif. ⟩ ::= ⟨ struktúranév ⟩ (⟨ argumentum ⟩, ...)
⟨ struktúranév ⟩ ::= ⟨ névkonstans ⟩
⟨ argumentum ⟩ ::= ⟨ kifejezés ⟩

Lexikai elemek: példák és szintaxis

```
% 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
```

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

Tartalom

Operátor-kifejezések

- Példa: `S is -S1+S2` ekvivalens az `is(S, +(-(S1),S2))` kifejezéssel
- Operátoros kifejezések

  ⟨ összetett kif. ⟩ ::=

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

  ⟨ operátornév ⟩ ::= ⟨ struktúranév ⟩ {ha operátorként lett definiálva}
- Operátort az alábbi beépített predikátummal definiálhatunk:
  - `op(Prioritás, Fajta, OpNév)`, `op(Prioritás, Fajta, [OpNév`$_1$`,...])`:
    - `Prioritás`: 1–1200 közötti egész
    - `Fajta`: az `yfx`, `xfy`, `xfx`, `fy`, `fx`, `yf`, `xf` névkonstansok egyike
    - `OpNév`: tetszőleges névkonstans
  - Az `op/3` beépített predikátum meghívását általában a programot tartalmazó file elején, *direktívában* helyezzük el:

    `:- op(800, xfx, [szuloje]).` `'Imre' szuloje 'István'.`
- A direktívák a programfile *betöltésekor* azonnal végrehajtódnak.

Operátorok jellemzői

- Egy operátort jellemez a fajtája és prioritása
- A fajta meghatározza az irásmódot és az asszociatívitás irányát:

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

- A zárójelezést a prioritás és az asszociatívitás együtt határozza meg, pl.
  - `a/b+c*d` ≡ `(a/b)+(c*d)` mert `/` és `*` prioritása 400 < 500 (`+` prioritása) (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

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	- \

További beépített operátorok SICStus Prologban

1150	fx	mode public dynamic volatile discontiguous initialization multifile meta_predicate block
1100	xfy	do
900	fy	spy nospy
550	xfy	:
500	yfx	\
200	fy	+

Operátorok implicit zárójelezése – általános szabályok

- Egy `X` op_1 `Y` op_2 `Z` zárójelezése, ahol op_1 és op_2 prioritása n_1 és n_2:
  - ha $n_1 > n_2$ akkor `X` op_1 `(Y` op_2 `Z)`;
  - ha $n_1 < n_2$ akkor `(X` op_1 `Y)` op_2 `Z`;
  - ha $n_1 = n_2$ és op_1 jobb-asszociatív (`xfy`), akkor `X` op_1 `(Y` op_2 `Z)`;
  - egyébként, ha $n_1 = n_2$ és op_2 bal-assz. (`yfx`), akkor `(X` op_1 `Y)` op_2 `Z`;
  - egyébként szintaktikus hiba
- Érdekes példa: `:- 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”.

- Alapszabály: egy n prioritású operátor zárójelezetlen operandusaként
  - legfeljebb $n-1$ prioritású operátort fogad el az `x` oldalon
  - legfeljebb n prioritású operátort fogad el az `y` oldalon
- Az alapszabály segítségével a prefix és posztfix operátorok zárójelezése is meghatározható
- Explicit zárójelekkel az implicit zárójelezés felülbírálható

Operátorok – kiegészítő megjegyzések

- A „vessző” kettős szerepe
  - a struktúra-kifejezés argumentumait választja el
  - 1000 prioritású `xfy` op. pl.: `(p:-a,b,c)`≡`:-(p,',',(a,',',(b,c)))`
- Egyértelműsítés: egy struktúra-kifejezés argumentumaként szereplő, 999-nél nagyobb prioritású kifejezést zárójelezni kell:

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

- Megjegyzés: a vessző (`,`) névkonstansként csak `','` alakban használható, de operátorként önmagában is állhat.
- Használható-e ugyanaz a név többféle fajtájú operátorként?
  - Nyilván nem lehet egy operátor egyszerre `xfy` és `xfx` is, stb.
  - De pl. a `+` és `-` operátorok `yfx` és `fy` fajtával is használhatók
- A könnyebb elemezhetőség miatt a Prolog szabvány kiköti, hogy
  - operátort operandusként zárójelbe kell tenni, pl. `Comp=(>)`
  - egy operátor nem lehet egyszerre infix és posztfix.

Sok Prolog rendszer (pl. a SICStus) nem követeli meg ezek betartását

Operátorok törlése, lekérdezése

- Egy vagy több operátor törlésére az `op/3` beépített eljárást használhatjuk, ha első argumentumként (prioritásként) 0-t adunk meg.

```
| ?- X = a+b, op(0, yfx, +). ⇒   X = +(a,b) ? ; no
| ?- X = a+b.                ⇒   ! Syntax error
                               ! operator expected after expression
                               ! X = a <<here>> + b .
| ?- op(500, yfx, +).        ⇒   yes
| ?- X = +(a,b).             ⇒   X = a+b ? ; no
```

- Az adott pillanatban érvényes operátorok lekérdezése:

```
current_op(Prioritás, Fajta, OpNév)
```

```
| ?- current_op(P, F, +).
F = fy, P = 200 ? ;
F = yfx, P = 500 ? ;
no
| ?- current_op(_P, xfy, Op), write_canonical(Op), write(' '), fail.
; do -> ',' : ^
no
```


Operátorok felhasználása

- Mire jók az operátorok?
  - aritmetikai eljárások kényelmes irására, pl. `X is (Y+3) mod 4`
  - szimbolikus kifejezések kezelésére (pl. szimbolikus deriválás)
  - klózok leírására (`:-` és `','` is operátor), és meta-eljárásoknak való átadására, 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.

```
sav(kén, h*2-s-o*4).
```

- Miért rossz a Prolog operátorfogalma?
  - A modularitás hiánya miatt:
  - Az operátorok egy globális erőforrást képeznek, ez nagyobb projektben gondot okozhat.

Aritmetika Prologban

- Az operátorok teszik lehetővé azt is, hogy a matematikában megszokott módon írhassunk le aritmetikai kifejezéseket.
- Például (ismétlés): az `is` beépített predikátum egy aritmetikai kifejezést vár a jobboldalán (2. argumentumában), azt kiértékeli, és az eredményt egyesíti a baloldali argumentummal
- (Az első Prolog rendszerekben is voltak operátorok, de `X is Y+Z` helyett a `plus(Y, Z, X)` hívást kellett használni.)
- 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 struktúra-kifejezések. Csak az aritmetikai beépített predikátumok értékelik ki ezeket!
- A Prolog kifejezések szimbolikusak, az aritmetikai kiértékelés a „kivétel”.

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 x=X helyen vett értéke E.
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
```

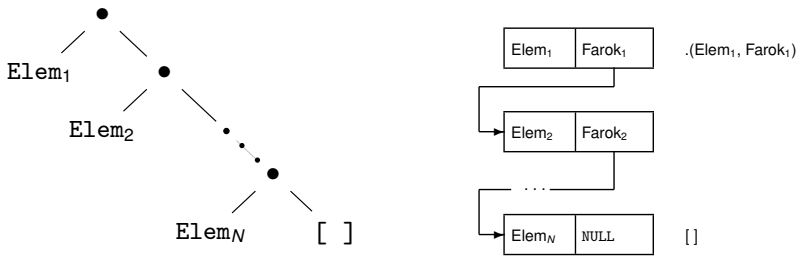

A Prolog lista-fogalma

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

```
| ?- write(.(1,.(2,[]))).      ⇒   [1,2]
| ?- write_canonical([1,2]).   ⇒   '.'(1,'.'(2,[]))
| ?- [1|[2,3]] = [1,2,3].      ⇒   yes
| ?- [1,2|[3]] = [1,2,3].      ⇒   yes
```


Listák írásmódjai

- Egy N elemű lista lehetséges írásmódjai:
  - alapstruktúra-alak: $.(Elem_1,.(Elem_2,...,.(Elem_N,[])...))$
  - ekvivalens lista-alak: $[Elem_1,Elem_2,...,Elem_N]$
  - kevésbe kényelmes ekvivalens alak: $[Elem_1|[Elem_2|...|[\ Elem_N|[]\ ]\ ...]]$
- A listák fastruktúra alakja és megvalósítása



Listák jelölése – szintaktikus édesítőszerek

- Az alapvető édesítés: `[Fej|Farok]` ≡ `.(Fej, Farok)`
- Kiterjesztés N „fej”-elemre:
  $[Elem_1,\ldots,Elem_N|Farok] \equiv [Elem_1|\ldots|[Elem_N|Farok]\ldots]$
- Ha a farok `[]`: $[Elem_1,\ldots,Elem_N] \equiv [Elem_1,\ldots,Elem_N|[]]$

```
| ?- [1,2] = [X|Y].              ⇒  X = 1, Y = [2] ?
| ?- [1,2] = [X,Y].              ⇒  X = 1, Y = 2 ?
| ?- [1,2,3] = [X|Y].            ⇒  X = 1, Y = [2,3] ?
| ?- [1,2,3] = [X,Y].            ⇒  no
| ?- [1,2,3,4] = [X,Y|Z].        ⇒  X = 1, Y = 2, Z = [3,4] ?
| ?- L = [1|_], L = [_,2|_].     ⇒  L = [1,2|_A] ? % nyílt végű
| ?- L = .(1,[2,3|[]]).          ⇒  L = [1,2,3] ?
| ?- L = [1,2|.(3,[])].          ⇒  L = [1,2,3] ?
| ?- [X|[3-Y/X|Y]]=
                .(A,[A-B,6]).    ⇒  A=3, B=[6]/3, X=3, Y=[6] ?
```


Tartalom

Diszjunkció

- Ismétlés: klóztörzsben a vessző (‘,’) jelentése „és”, azaz konjunkció
- A ‘;’ operátor jelentése „vagy”, azaz diszjunkció

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

```
fakt(N, F) :-
    (   N = 0, F = 1

    ;   N > 0, N1 is N-1,
        fakt(N1, F1), F is F1*N
    ).
```

- A diszjunkciót nyitó zárójel elérésekor választási pont jön létre
  - először a diszjunkciót az első ágára redukáljuk
  - visszalépés esetén a diszjunkciót a második ágára redukáljuk
- Tehát az első ág sikeres lefutása után kilépünk a diszjunkcióból, és az utána jövő célokkal folytatjuk a redukálást
  - azaz a ‘;’ elérésekor a ‘)’-nél folytatjuk a futást
- A ‘;’ skatulyázható (jobbról-balra) és gyengébben köt mint a ‘,’
- Konvenció: a diszjunkciót mindig zárójelbe tesszük, a skatulyázott diszjunkciót és az ágakat feleslegesen nem zárójelezzük. Pl. (a felesleges zárójelek pirossal jelölve): `(p;(q;r)), (a;(b,c);d)`

A diszjunkció mint szintaktikus édesítőszer

- A diszjunkció egy segéd-predikátummal kiküszöbölhető, pl.:

```
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).
```

- Kigyűjtjük a diszjunkcióban és azon kívül is előforduló változókat
- 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).
```

Diszjunkció – megjegyzések

- Az egyes klózok ‘ÉS’ vagy ‘VAGY’ kapcsolatban vannak?
  - A program klózai **ÉS** kapcsolatban vannak, pl.

```
szuloje('Imre', 'István').     szuloje('Imre', 'Gizella').        % (1)
```

    azt állítja: Imre szülője István **ÉS** Imre szülője Gizella.
  - Az *(1)* klózok alternatív (VAGY kapcsolatú) válaszokhoz vezetnek:

```
:- szuloje('Imre' Ki). ⇒ Ki = 'István' ? ; Ki = 'Gizella' ? ; no
```

    „`Ki Imre` szülője?” akkor és csak akkor ha `Ki` = `István` vagy `Ki` = `Gizella`.
- Az *(1)* predikátum átalakítható egyetlen, diszjunkciós klózzá:

```
szuloje('Imre', Sz) :-          (   Sz = 'István'
                                ;   Sz = 'Gizella'
                                ).                                  % (2)
```

  Vö. De Morgan azonosságok: $(A \leftarrow B) \wedge (A \leftarrow C) \equiv (A \leftarrow (B \vee C))$
- Általánosan: tetszőleges predikátum egyklózossá alakítható:
  - a klózokat azonos fejűvé alakítjuk, új változók és =-ek bevezetésével:

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

  - a klóztörzseket egy diszjunkcióvá fogjuk össze, lásd *(2)*.

A meghiúsulásos negáció (NF – Negation by Failure)

- A `\+ Hívás` vezérlési szerkezet (vö. $\nvdash$ – nem bizonyítható) procedurális szemantikája
  - 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
- Példa: Keressünk (adatbázisunkban) olyan gyermeket, aki **nem** szülő!
- Ehhez negációra van szükségünk, egy megoldás:

```
| ?- sz(X, _Sz), \+ sz(Gy, X). % negált cél ≡ ¬(∃Gy.sz(Gy,X))
X = 'Imre' ? ; no
```

- Mi történik ha a két hívást megcseréljük?

```
| ?- \+ sz(Gy, X), sz(X, _Sz).% negált cél ≡ ¬(∃Gy,X.sz(Gy,X))
no
```

- $\backslash + H$ deklaratív szemantikája: $\neg\exists\vec{X}(H)$, ahol $\vec{X}$ a H-ban a hívás pillanatában behelyettesítetlen változók felsorolását jelöli.

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


Gondok a meghiúsulásos negációval

- A negált cél jelentése függ attól, hogy mely változók bírnak értékkel
- Mikor nincs gond?
  - Ha a negált cél tömör (nincs benne behelyettesítetlen változó)
  - Ha nyilvánvaló, hogy mely változók behelyettesítetlenek (pl. _), és a többi tömör.

```
% nem_szulo(+Sz): adott Sz nem szulo
nem_szulo(Sz) :- \+ szuloje(_, Sz).
```

- További gond: „zárt világ feltételezése” (Closed World assumption – CWA): ami nem bizonyítható, az nem igaz.

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

  - A klasszikus matematikai logika következményfogalma monoton: ha a premisszák halmaza bővül, a következmények halmaza nem szűkülhet.
  - A CWA alapú logika nem monoton, példa: bővítsük a programot egy `szuloje('Géza', xxx).` alakú állítással $\Rightarrow$`(*)` meghiúsul.

Példa: együttható meghatározása lineáris kifejezésben

- Formula: számokból és az ‘`x`’ atomból ‘`+`’ és ‘`*`’ operátorokkal épül fel.
- 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.
```

- Van amikor többszörös megoldást kapunk:

```
| ?- egyhat(((x+1)*3)+x+2*(x+x+3), E).
E = 8 ? ;
no
```

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

Többszörös megoldások kiküszöbölése

- A többszörös megoldás oka: az `egyhat(2*3, E)` hívás esetén a 4. és 5. klóz is sikeresen lefut.
- A többszörös megoldás 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
;   ...
))
```

  A piros zárójelek feleslegesek (a ‘;’ egy `xfy` írásmódú op.).
- Az `egyébként` rész elhagyható, alapértelmezése: `fail`.
- `\+ felt` ekvivalens alakja: `( felt -> fail ; true )`

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
        ).
```

Tartalom

Listák összefűzése: az append/3 eljárás

- append(L1, L2, L3): Az L3 lista az L1 és L2 listák elemeinek egymás után fűzése (L3 = L1⊕L2) – Cékla megoldás, és Prolog fordítása:

```
list append(list L0, list L2) {
  if (L0 == nil) return L2;
  int X = hd(L0);
  list L1 = tl(L0);
  list L3 = append(L1, L2);
  return cons(X, L3);          }
```

```
append__(L0, L2, R) :-
        (   L0=[] -> R=L2
        ;   hd(L0, X),
            tl(L0, L1),
            append__(L1, L2, L3),
            cons(X, L3, R)       ).
```

- Itt 'hd(L0,X), tl(L0,L1)' ≡ 'L0 = [X|L1]', 'cons(X, L3, R)' ≡ 'R = [X|L3]'; ez a feltételes szerk. ≡ diszjunkció (' -> ' ⇒', ') ≡ app0 $\overset{?}{\equiv}$ app1 pred.

```
app0([], L2, R) :- R = L2.
app0([X|L1], L2, R) :-
    app0(L1, L2, L3), R = [X|L3].
```

```
app1([], L, L).
app1([X|L1], L2, R) :-
    R = [X|L3], app1(L1, L2, L3).
```

- Az app$i$(L1, ...) komplexitása: a futási idő arányos L1 hosszával
- Miért jobb az app1/3 mint az app0/3?
  - app1/3 **jobbrekurzív**, ciklussal ekvivalens (nem fogyaszt vermet)
  - app1([1,...,1000],[0],[2,...]) 1, app0(...) 1000 lépésben hiúsul meg.
  - app1/3 használható szétszedésre is (lásd később), míg app0/3 nem.

Lista építése *elölről* – nyílt végű listákkal

- Egy X Prolog kif. nyílt végű lista, ha X változó, vagy X = [_|Farok] ahol Farok nyílt végű lista.

  | ?- L = [1|_], L = [_,2|_]. ⇒ L = [1,2|_A] ?

- Az app1/3 eljárás ekvivalens a beépített append/3-mal:

```
app1([], L, L).
app1([X|L1], L2, R) :-
    R = [X|L3], app1(L1, L2, L3).
```

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

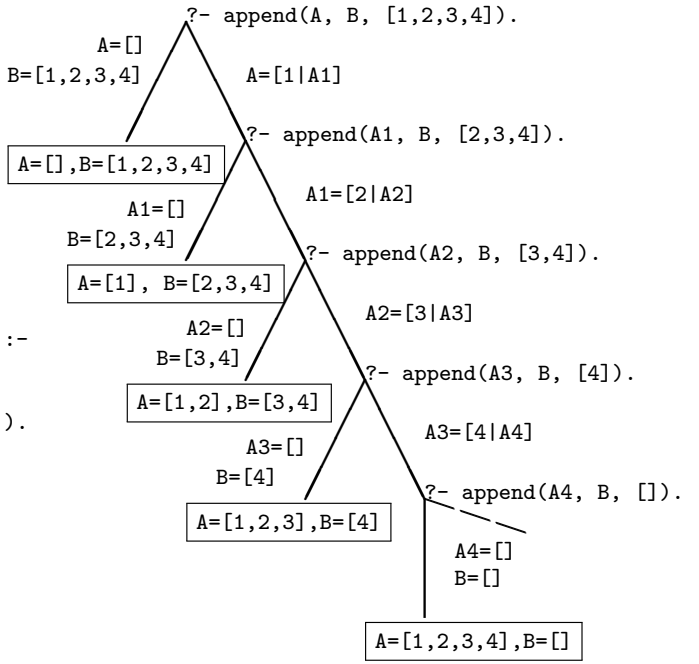
- Az append eljárás már az első redukciónál felépíti az eredmény fejét!
  - Célok (pl.): append([1,2,3], [4], Ered), write(Ered).
  - Fej: append([X|L1], L2, [X|L3])
  - Behelyettesítés: X = 1, L1 = [2,3], L2 = [4], Ered = [1|L3]
  - Új célsorozat: append([2,3], [4], L3), write([1|L3]). (Ered nyílt végű lista, farka még behelyettesítetlen.)
  - A további redukciós lépések behelyettesítése és eredménye:

```
L3 =  [2|L3a]     append([3], [4], L3a),   write([1|[2|L3a]]).
L3a = [3|L3b]     append([], [4], L3b),    write([1,2|[3|L3b]]).
L3b = [4]                                  write([1,2,3|[4]]).
```


Listák szétbontása az append/3 segítségével

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

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



Milyen módon használhatók az append változatok?

- app0: Logikai változó (nyílt végű lista) használata: app0: nem append: igen.

```
app0([], L, L).
app0([X|L1], L2, R) :-
    app0(L1, L2, L3), R = [X|L3].
```

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

- app0(L1,_,_) korlátos futású, ha L1 zárt végű: max. *len*(L1) mélység.

```
| ?- app0([1,2], L2, L3).
L3 = [1,2|L2] ? ; no
```

- app0(_,L2,_) esetén (még ha L2 zárt végű is) ∞ sok megoldás van:

```
| ?- app0(L1, [1,2], L3).
L1 = [], L3 = [1,2] ? ;            L1 = [A], L3 = [A,1,2] ? ;      (*)
L1 = [A,B], L3 = [A,B,1,2] ? ; L1 = [A,B,C], L3 = [A,B,C,1,2] ? ; ...
```

- app0(_,L2,L3) esetén (L2, L3 zárt végű) ∞ ciklust kapunk:

app0(L, [1,2], []) redukálva a 2. klózzal app0(L1, [1,2], L3), [X|L3] = [].

- append(L1, L2, L3) keresési tere véges, ha L1 vagy L3 zárt végű!
  Ha L1 és L3 nyílt végű, akkor ∞ sok megoldás lehet, lásd (*).

Variációk appendre 1. – Három lista összefűzése

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

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

- Nem hatékony, pl.: append([1,...,100],[1,2,3],[1], L) 103 helyett 203 lépés!
- Szétszedésre nem alkalmas – végtelen választási pontot hoz létre
- Szétszedésre is alkalmas, hatékony változat

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

  - Az első append/3 hívás nyílt végű listát állít elő:
    | ?- append([1,2], L23, L). ⇒ L = [1,2|L23] ?
  - Az L3 argumentum behelyettesítettsége (nyílt vagy zárt végű lista-e) nem számít.

Mintakeresés append/3-mal

- Párban előforduló elemek

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

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

- Dadogó részek

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

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


Listák megfordítása

- Naív (négyzetes lépésszámú) megoldás

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

- Lineáris lépésszámú megoldás

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

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

- A lists könyvtár tartalmazza a reverse/2 eljárás definícióját.
- A könyvtár betöltése:

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

Listák gyűjtési iránya – append és revapp Prolog és C++ nyelven

- Prolog

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

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

- C++

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

```
list append(list L1, list L2)
{ list L3, *lp = &L3;
  for (list p=L1; p; p=p->next)
  { list newl = new link(p->elem);
    *lp = newl; lp = &newl->next;
  }
  *lp = L2;  return L3;
}
```

```
list revapp(list L1, list L2)
{ list l = L2;
  for (list p=L1; p; p=p->next)
  { list newl = new link(p->elem);
    newl->next = l; l = newl;
  }
  return l;
}
```


Keresés listában – a member/2 beépített eljárás

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

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

```
member(Elem, [Fej|Farok]) :-
    (   Elem = Fej
    ;   member(Elem, Farok)
    ).
```

- Eldöntendő (igen-nem) kérdés:

```
| ?- member(2, [1,2,3,2]).      ⇒  yes                  DE
| ?- member(2, [1,2,3,2]),X=X.  ⇒  true ? ; true ? ; no
```

- Lista elemeinek felsorolása:

```
| ?- member(X, [1,2,3]).        ⇒  X = 1 ? ; X = 2 ? ; X = 3 ?  ; no
| ?- member(X, [1,2,1]).        ⇒  X = 1 ? ; X = 2 ? ; X = 1 ?  ; no
```

- Listák közös elemeinek felsorolása – az előző két hivásformát kombinálja:

```
| ?- member(X, [1,2,3]),
     member(X, [5,4,3,2,3]).    ⇒  X = 2 ? ; X = 3 ? ; X = 3 ? ; no
```

- Egy értéket egy (nyílt végű) lista elemévé tesz, végtelen választás!

```
| ?- member(1, L).              ⇒  L = [1|_A] ? ; L = [_A,1|_B] ? ;
                                   L = [_A,_B,1|_C] ? ; ...
```

- A member/2 keresési tere **véges**, ha 2. argumentuma zárt végű lista.

A member/2 predikátum általánosítása: select/3

- select(E, Lista, Marad): E-t a Listaból elhagyva marad Marad.

```
select(E, [E|Marad], Marad).     % Elhagyjuk a fejet, marad a farok.
select(E, [X|Farok], [X|M0]) :- % Marad a fej,
        select(E, Farok, M0).    %    a farokból hagyunk el elemet.
```

- Felhasználási lehetőségek:

```
| ?- select(1, [2,1,3,1], L).    % Adott elem elhagyása
        L = [2,3,1] ? ; L = [2,1,3] ? ; no
| ?- select(X, [1,2,3], L).      % Akármelyik elem elhagyása
        L=[2,3], X=1 ? ;  L=[1,3], X=2 ? ;   L=[1,2], X=3 ? ; no
| ?- select(3, L, [1,2]).        % Adott elem beszúrása!
        L = [3,1,2] ? ; L = [1,3,2] ? ; L = [1,2,3] ? ; no
| ?- select(3, [2|L], [1,2,7,3,2,1,8,9,4]).
                                 % Beszúrható-e 3 az [1,...]-ba
        no                       % úgy, hogy [2,...]-t kapjunk?
| ?- select(1, [X,2,X,3], L).
        L = [2,1,3], X = 1 ? ; L = [1,2,3], X = 1 ? ; no
```

- A lists könyvtárban a fenti módon definiált select/3 eljárás keresési tere **véges**, ha vagy a 2., vagy a 3. argumentuma zárt végű lista.

Listák permutációja

- perm(Lista, Perm): Lista permutációja a Perm lista.

```
perm([], []).
perm(Lista, [Elso|Perm]) :-
        select(Elso, Lista, Maradek),
        perm(Maradek, Perm).
```

- Felhasználási példák:

```
| ?- perm([1,2], L).
        L = [1,2] ? ; L = [2,1] ? ; no

| ?- perm([a,b,c], L).
        L = [a,b,c] ? ; L = [a,c,b] ? ; L = [b,a,c] ? ;
        L = [b,c,a] ? ; L = [c,a,b] ? ; L = [c,b,a] ? ; no

| ?- perm(L, [1,2]).
        L = [1,2] ? ;  végtelen keresési tér
```

- Ha perm/2-ben az első argumentum ismeretlen, akkor a select hívás keresési tere végtelen!
- A lists könyvtár tartalmaz egy kétirányban is működő permutation/2 eljárást.