

BEVEZETÉS A LOGIKAI PROGRAMOZÁSBA



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 = Programozás logikában (Programming in logic)**
 - az elsőrendű logika egy erősen megszorított résznyelve az ún. **definit-** vagy **Horn-klózik** nyelve,
 - végrehajtási mechanizmusa: **mintaillesztéses** eljáráshíváson alapuló **visszalépéses** keresés.

Az előadás LP részének áttekintése

- **1. blokk:** A Prolog nyelv alapjai (6 előadás)
 - Logikai háttér
 - Szintaxis
 - Végrehajtási mechanizmus
- **2. blokk:** Prolog programozási módszerek (6 előadás)
 - A legfontosabb beépített eljárások
 - Fejlettebb nyelvi és rendszerelemek
- Kitekintés: Új irányzatok a logikai programozásban (1 előadás)

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

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.

Márkus 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

English Textbooks on Prolog

- Logic, Programming and Prolog, 2nd Ed., by Ulf Nilsson and Jan Maluszynski, Previously published by John Wiley & Sons Ltd. (1995)
Downloadable as a pdf file from <http://www.ida.liu.se/~ulfni/lpp>
- Prolog Programming for Artificial Intelligence, 3rd Ed., Ivan Bratko, Longman, Paperback - March 2000
- The Art of PROLOG: Advanced Programming Techniques, Leon Sterling, Ehud Shapiro, The MIT Press, Paperback - April 1994
- Programming in PROLOG: Using the ISO Standard, C.S. Mellish, W.F. Clocksin, Springer-Verlag Berlin, Paperback - July 2003

Első Prolog programunk: hatványellenőrzés

- Egy egyszerű példa

```
/* ispow(a,b) = 1 <=> exists i, such that bi = a. Precondition: a,b > 0 */
```

```
int ispow(int num, int base) {
    if (num == 1)
        return 1;
    else if (num%base == 0)
        return ispow(num/base, base);
    else
        return 0;
}

ispow(Num, Base) :-
    (    Num == 1
    -> true
    ;    Num rem Base == 0,
        Num1 is Num//Base,
        ispow(Num1, Base)
    ).
```

- ispow egy Prolog **predikátum**, azaz Boole értékű eljárás.
- Az eljárás egyetlen klózból áll, amely *Fej:-Törzs* alakú.
- Az eljárásfejben a paraméterek a Num és Base változók (**nagybetűsek!**)
- A törzs egyetlen célt tartalmaz, amely egy **feltételes szerkezet**:
if Felt then Akkor else Egyébként \equiv (Felt -> Akkor ; Egyébként)
- A „true”, „A == B” és „A is B” szerkezetek beépített predikátumok hívásai.

Néhány beépített predikátum

- Egyesítés: $x = y$: az x és y **szimbolikus** kifejezések változók behelyettesítésével azonos alakra hozhatók (és el is végzi a behelyettesítéseket).
- Aritmetikai predikátumok
 - $x \text{ is } Kif$: A Kif **aritmetikai** kifejezést kiértékeli és **értékét** egyesíti x -szel.
 - $Kif1 < Kif2$, $Kif1 = < Kif2$, $Kif1 > Kif2$, $Kif1 >= Kif2$, $Kif1 =: = Kif2$, $Kif1 = \backslash = Kif2$:
A $Kif1$ és $Kif2$ aritmetikai kifejezések értéke a megadott relációban van egymással
($=: =$ jelentése: aritmetikai egyenlőség, $= \backslash =$ jelentése aritmetikai nem-egyenlőség).
 - Ha Kif , $Kif1$, $Kif2$ valamelyike nem **tömör** (változómentes) aritmetikai kifejezés \Rightarrow hiba.
 - Legfontosabb aritmetikai operátorok: $+$, $-$, $*$, $/$, rem , $//$ (egész-osztás)
- Kiíró predikátumok
 - $write(x)$: Az x Prolog kifejezést kiírja.
 - nl : Kiír egy újsort.
- Egyéb predikátumok
 - $true$, $fail$: Mindig sikerül ill. mindig megghiúsul.
 - $trace$, $notrace$: A (teljes) nyomkövetést be- ill. kikapcsolja.

Programfejlesztési beépített predikátumok

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

```
> sicstus
SICStus 3.11.0 (x86-linux-glibc2.3): Mon Oct 20 15:59:37 CEST 2003
| ?- consult(ispow).
% consulted /home/user/ispow.pl in module user, 0 msec 376 bytes
yes
| ?- ispow(8, 3).
no
| ?- ispow(8, 2).
yes
| ?- listing(ispow).
(...)
yes
| ?- halt.
>
```

Általános (nem Boole-értékű) függvények Prologban

- Példa: Természetes számok hatványozása

```
/* powd(a,n) = a^n */
int powd(int a, int n) {
    if (n > 0)
        return a*powd(a,n-1);

    else
        return 1;
}
```

```
/* powd(A, N, P): A^N = P. */
powd(A, N, P) :-
    ( N > 0
    -> N1 is N-1,
        powd(A, N1, P1),
        P is A*P1
    ; P = 1
    ).
```

```
| ?- powd(2, 8, P).
P = 256 ?
```

- A 2-argumentumú `powd` függvénynek a 3-argumentumú `powd` predikátum felel meg.
- A függvény két argumentumának a predikátum első két, **bemenő**, azaz behelyettesített argumentuma felel meg.
- A függvény eredménye a predikátum utolsó, kimenő argumentuma, amely általában behelyettesítetlen változó.

Predikátum definiálása több klózzal

- A feltételes szerkezet nem alap-építőeleme a Prolog nyelvnek (az első Prologokban nem is volt)
- Helyette több **egymást kizáró** klózt is lehet alkalmazni:

```

/* powd(A, N, P): A^N = P. */
powd(A, N, P) :-
    ( N > 0
    -> N1 is N-1,
        powd(A, N1, P1),
        P is A*P1
    ; P = 1
    ).

powd2(A, N, P) :-
    N > 0,
    N1 is N-1,
    powd2(A, N1, P1),
    P is A*P1.
powd2(A, N, 1) :- N =< 0.

```

- Ha egy predikátum több klózból áll, a Prolog **mindegyiket** megkísérli felhasználni a futásban:
 - ha `powd2` második paramétere (`N`) pozitív, akkor az első klózt használja,
 - egyébként (azaz ha `N =< 0`) a második klózt.
- Ha `powd2` második klózaként `powd(A, 0, 1)` állna akkor negatív kitevő esetén a hívás megghiúsulna (egyik klóz sem lenne alkalmazható).
- Általában nem kell kizáró feltételeket alkalmazni, így egy kérdésre több választ is kaphatunk!

```

gyöke(A, B, C, X) :- X is (-B + sqrt(B*B-4*A*C))/(2*A).
gyöke(A, B, C, X) :- X is (-B - sqrt(B*B-4*A*C))/(2*A).

```

Több választ adó predikátumok — a családi kapcsolatok példája

- Adatok

Egy gyerek–szülő kapcsolat, pl.

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

- A feladat:

Definiálandó az unoka–nagyözülő kapcsolat, pl. keressük egy adott személy nagyözüleit.

A nagyszülő feladat — Prolog megoldás

```

% szuloje(Gy, Sz):Gy szülője Sz.
szuloje('Imre', 'István').
szuloje('Imre', 'Gizella').
szuloje('István', 'Géza').
szuloje('István', 'Sarolt').
szuloje('Gizella',
        'Civakodó Henrik').
szuloje('Gizella',
        'Burgundi Gizella').

% Gyerek nagyszülője Nagyszulo.
nagyszuloje(Gyerek, Nagyszulo) :-
    szuloje(Gyerek, Szulo),
    szuloje(Szulo, Nagyszulo).

```

```

% 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

```

Adatstruktúrák deklaratív nyelvekben — példa

- A bináris fa adatstruktúra
 - vagy egy csomópont (`node`), amelynek két részfája van mutat (`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-deklaráció SML-ben

datatype Tree =
    Node of Tree * Tree
    | Leaf of int

% Adattípus-leírás Prologban

:- 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 sum_tree(struct tree *tree)
{
  switch(tree->type) {
    case Leaf:
      return tree->u.leaf.value;
    case Node:
      return
        sum_tree(tree->u.node.left) +
        sum_tree(tree->u.node.right);
  }
}
```

```
% SML függvény
fun sum_tree( Node(Left,Right) )
  = sum_tree Left +
    sum_tree Right
| sum_tree( Leaf(Val) ) = Val

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


Bináris fák összegzése

- Prolog példafutás

```
% sicstus -f
SICStus 3.10.0 (x86-linux-glibc2.1): Tue Dec 17 15:12:52 CET 2002
Licensed to BUTE DP course
| ?- consult(tree).
% consulting /home/szeredi/peldak/tree.pl...
% consulted /home/szeredi/peldak/tree.pl in module user, 0 msec 704 bytes
yes
| ?- sum_tree(node(leaf(5),
                node(leaf(3), leaf(2))), Sum).
Sum = 10 ? ;
no
| ?- sum_tree(Tree, 10).
Tree = leaf(10) ? ;
! Instantiation error in argument 2 of is/2
! goal: 10 is _73+_74
| ?- halt.
%
```

- A hiba oka: a beépített aritmetika egyirányú: a `10 is S1+S2` hívás hibát jelez!

Peano aritmetika — összeadás

- A természetes számok halmazán az összeadást definiálhatjuk a Peano axiómákkal ha a számokat az $s(x)$ „rákövetkező” függvény segítségével ábrázoljuk:

$1 = s(0)$, $2 = s(s(0))$, $3 = s(s(s(0)))$, ... (Peano ábrázolás).

`% plus(X, Y, Z): X és Y összege Z (X, Y, Z Peano ábrázolású).`

`plus(0, X, X).`

`% 0+X = X.`

`plus(s(X), Y, s(Z)) :-`

`plus(X, Y, Z).`

`% s(X)+Y = s(X+Y).`

- A `plus` predikátum több irányban is használható:

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

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

| `?- plus(X, Y, s(s(0))).` `X = 0, Y = s(s(0)) ? ;` `% 2 = 0+2`

`X = s(0), Y = s(0) ? ;` `% 2 = 1+1`

`X = s(s(0)), Y = 0 ? ;` `% 2 = 2+0`

`no`

| `?-`

Adott összegű fák építése

- Adott összegű fát építő eljárás Peano aritmetikával:

```

sum_tree(leaf(Value), Value).
sum_tree(node(Left, Right), S) :-
    plus(S1, S2, S),
    S1 \= 0, S2 \= 0,           % X \= Y beépített eljárás, jelentése:
                                % X és Y nem egyesíthető
                                % A 0-t kizárjuk, mert különben  $\infty$  sok megoldás van.

    sum_tree(Left, S1),
    sum_tree(Right, S2).

```

- Az eljárás futása:

```

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

```

A Prolog adatfogalma, a Prolog kifejezés

- konstans (*atomic*)
 - számkonstans (*number*) — egész vagy lebegőpontos, pl. `1`, `-2.3`, `3.0e10`
 - névkonstans (*atom*), pl. `'István'`, `ispow`, `+`, `-`, `<`, `sum_tree`
- összetett- vagy struktúra-kifejezés (*compound*)
 - ún. kanonikus alak: $\langle \text{struktúranév} \rangle (\langle \text{arg}_1 \rangle, \dots)$
 - a $\langle \text{struktúranév} \rangle$ egy névkonstans, az $\langle \text{arg}_i \rangle$ argumentumok tetszőleges Prolog kifejezések
 - példák: `leaf(1)`, `person(william,smith,2003,1,22)`, `<(X,Y)`, `is(X, +(Y,1))`
 - szintaktikus „édesítőszerek”, pl. operátorok: `X is Y+1` \equiv `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)

A PROLOG NYELV KÖZELÍTŐ SZINTAXISA



Predikátumok, klózek

● Példa:

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

● Szintaxis:

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

Prolog programok formázása

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

Prolog kifejezések

● Példa — egy klózfej mint kifejezés:

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

● Szintaxis:

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

Lexikai elemek

● Példák:

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

● Szintaxis:

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

Szintaktikus édesítőszerek: operátorok

- Példa:

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

- Operátoros kifejezések

$\langle \text{összetett kifejezés} \rangle ::=$

$\langle \text{struktúranév} \rangle (\langle \text{argumentum} \rangle, \dots)$	{eddig csak ez volt}
$\langle \text{argumentum} \rangle \langle \text{operátornév} \rangle \langle \text{argumentum} \rangle$	{infix kifejezés}
$\langle \text{operátornév} \rangle \langle \text{argumentum} \rangle$	{prefix kifejezés}
$\langle \text{argumentum} \rangle \langle \text{operátornév} \rangle$	{posztfix kifejezés}
$\langle \text{operátornév} \rangle ::= \langle \text{struktúranév} \rangle$	{ha operátorként lett definiálva}

- Operátor-kezelő beépített predikátumok:

- `op(Prioritás, Fajta, OpNév)` vagy `op(Prioritás, Fajta, [OpNév1, OpNév2, ...])`:
 - Prioritás: 0–1200 közötti egész
 - Fajta: az `yfx`, `xfy`, `xfx`, `fy`, `fx`, `yf`, `xf` névkonstansok egyike
 - OpNév: tetszőleges névkonstans
 - pozitív prioritás esetén definiálja az operátor(oka)t, 0 prioritás esetén megszünteti azokat.
- `current_op(Prioritás, Fajta, OpNév)`: felsorolja a definiált operátorokat.

Szabványos, beépített operátorok

Szabványos operátorok

```

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

```

Egyéb beépített operátorok SICStus Prologban

```

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

```

Operátorok jellemzői

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

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

- 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 fajtanévben az y betű mutatja az asszociatívitás irányát)
 - $a^b^c \equiv a^(b^c)$ 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.: $(p \text{ :- } a, b, c) = \text{:-}(p, ', '(a, ', '(b, c)))$
- a „pucér” vessző $(,)$ nem névkonstans, de operátorként aposztrofok nélkül is írható.
- struktúra-argumentumban 999-nél nagyobb prioritású kifejezést zárójelezni kell:

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

- Az egyértelmű elemezhetőség érdekében a Prolog szabvány kiköti, hogy

- operandusként előforduló operátort zárójelbe kell tenni, pl. $\text{Comp} = (>)$
- nem létezhet azonos nevű infix és posztfix operátor.

- Sok Prolog rendszerben nem kötelező betartani ezeket a megszorításokat.

Operátorok felhasználása

- Mire jók az operátorok?

- aritmetikai eljárások kényelmes írására, pl. $X \text{ is } (Y+3) \bmod 4$
- aritmetikai kifejezések szimbolikus feldolgozására (pl. szimbolikus deriválás)
- klózok leírására (`:-` és `' , '` is operátor)
- klózok átadhatók meta-eljárásoknak, pl `asserta((p(X):-q(X),r(X)))`
- eljárásfejek, eljárás hívások olvashatóbbá tételére:

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

```
Gy nagyszülője N :- Gy szülője Sz, Sz szülője N.
```

- adatstruktúrák olvashatóbbá tételére, pl.

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

```
sav(kén, h.2-s-o.4).
```

- Miért rosszak az operátorok?

- egyetlen globális erőforrás, ez nagyobb projektben gondot okozhat.

Aritmetika Prologban

- Az operátorok teszik lehetővé azt is, hogy a matematikában ill. más programozási nyelvekben megszokott módon írassunk le aritmetikai kifejezéseket.
- Az `is` beépített predikátum egy aritmetikai kifejezést vár a jobboldalán (2. argumentumában), azt kiértékeli, és az eredményt egyesíti a baloldali argumentummal
- Az `==` beépített predikátum mindkét oldalán aritmetikai kifejezést vár, azokat kiértékeli, és csak akkor sikerül, ha az értékek megegyeznek.

- Példák:

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

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

Klasszikus szimbolikus kifejezés-feldolgozás: deriválás

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

```
% deriv(Kif, D): Kif-nek az x szerinti deriváltja D.
```

```
deriv(x, 1).
```

```
deriv(C, 0) :- number(C).
```

```
deriv(U+V, DU+DV) :- deriv(U, DU), deriv(V, DV).
```

```
deriv(U-V, DU-DV) :- deriv(U, DU), deriv(V, DV).
```

```
deriv(U*V, DU*V + U*DV) :- deriv(U, DU), deriv(V, DV).
```

```
| ?- deriv(x*x+x, D).
```

```
⇒ D = 1*x+x*1+1 ? ; no
```

```
| ?- deriv((x+1)*(x+1), D).
```

```
⇒ D = (1+0)*(x+1)+(x+1)*(1+0) ? ; no
```

```
| ?- deriv(I, 1*x+x*1+1).
```

```
⇒ I = x*x+x ? ; no
```

```
| ?- deriv(I, 0).
```

```
⇒ no
```

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

LISTÁK PROLOGBAN



A Prolog lista-fogalma

● A Prolog lista

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

● Példa

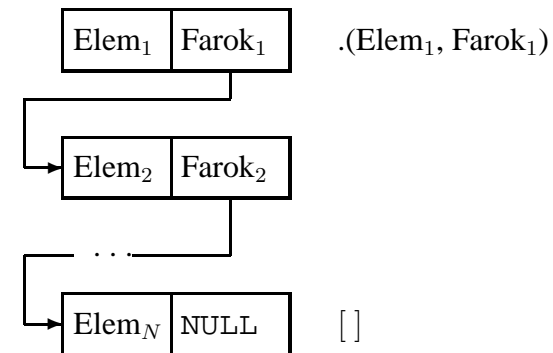
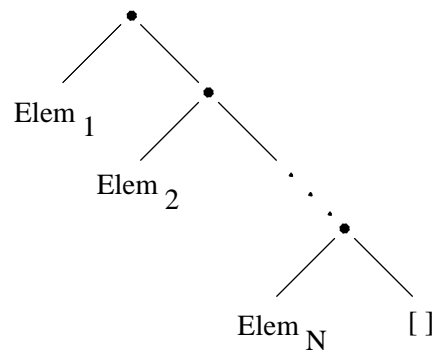
```
számlista.(E,L) :-
    number(E), számlista(L).
számlista([]).

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

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

Listák írásmódjai

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



Listák jelölése — szintaktikus édesítőszer

• az alapvető édesítés: $[Fej | Farok] \equiv .(Fej, Farok)$

• N -szeri alkalmazás kevesebb zárójellel:

$$[Elem_1, Elem_2, \dots, Elem_N | Farok] \equiv \\ [Elem_1 | [Elem_2 | \dots | [Elem_N | Farok] \dots]]$$

• Ha a farok $[]$: $[Elem_1, Elem_2, \dots, Elem_N] \equiv [Elem_1, Elem_2, \dots, Elem_N | []]$

| ?- $[1, 2] = [X | Y].$ $\Rightarrow X = 1, Y = [2] ?$

| ?- $[1, 2] = [X, Y].$ $\Rightarrow X = 1, Y = 2 ?$

| ?- $[1, 2, 3] = [X | Y].$ $\Rightarrow X = 1, Y = [2, 3] ?$

| ?- $[1, 2, 3] = [X, Y].$ \Rightarrow no

| ?- $[1, 2, 3, 4] = [X, Y | Z].$ $\Rightarrow X = 1, Y = 2, Z = [3, 4] ?$

| ?- $L = [1 | _], L = [_ , 2 | _].$ $\Rightarrow L = [1, 2 | _A] ?$ % nyílt végű

| ?- $L = .(1, [2, 3 | []]).$ $\Rightarrow L = [1, 2, 3] ?$

| ?- $L = [1, 2 | .(3, [])].$ $\Rightarrow L = [1, 2, 3] ?$

| ?- $[X | [3 - Y / X | Y]] = .(A, [A - B, 6]).$ $\Rightarrow A = 3, B = [6] / 3, X = 3, Y = [6] ?$

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

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

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

A logikai változó

- A logikai változó fogalma:
 - kifejezésként, kifejezésben egyaránt előfordulhat, vö. a változókat a (lista) mintákban.
 - két változó azonossá tehető (azaz egyesíthető): pl. két azonos változó egy kifejezésben.
 - a változó „teljes jogú” állampolgár a (rész)kifejezések világában
- SML-ben is van mintaillesztés, de a minta csak szétszedésre használható, összerakásra nem; a mintabeli változók mindig (tömör) értéket kapnak.
- (Egyes újabb funkcionális nyelvek, pl. az Oz nyelv, támogatják a logikai változókat.)
- Példa: Az alábbi célsorozat egy két **azonos** elemből álló listát épít fel az `L` változóban. Az elemek értéke **azonos** lesz a célsorozatbeli `x` változóval:

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

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

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

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


Listák összefűzése: az append / 3 eljárás

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

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

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

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

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

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

Lista építése *előlről* — nyílt végű listákkal

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

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

- Haladó nyomkövetési lehetőségek ennek demonstrálására
 - `library(debugger_examples)` —példák a nyomkövet ő programozására, új parancsokra
 - új parancs: ‘N <név>’ —fókuszált argumentum elnevezése
 - szabványos parancs: ‘^ <argszám>’ —adott argumentumra fókuszlás
 - új parancs: ‘P [<név>]’ —adott nevű (ill összes) kifejezés kiiratása

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

Listák megfordítása

- Naív (négyzetes lépésszámú) megoldás

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

- Lineáris lépésszámú megoldás

```
% reverse(R, L): Az R lista az L megfordítása.  
reverse(R, L) :- revapp(L, [], R).  
  
% revapp(L1, L2, R): L1 megfordítását L2 elé fűzve kapjuk R-t.  
revapp([], R, R).  
revapp([X|L1], L2, R) :-  
    revapp(L1, [X|L2], R).
```

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

- A könyvtár betöltése:

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

append és revapp — listák gyűjtési iránya

● Prolog megvalósítás

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

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

● C++ megvalósítás

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

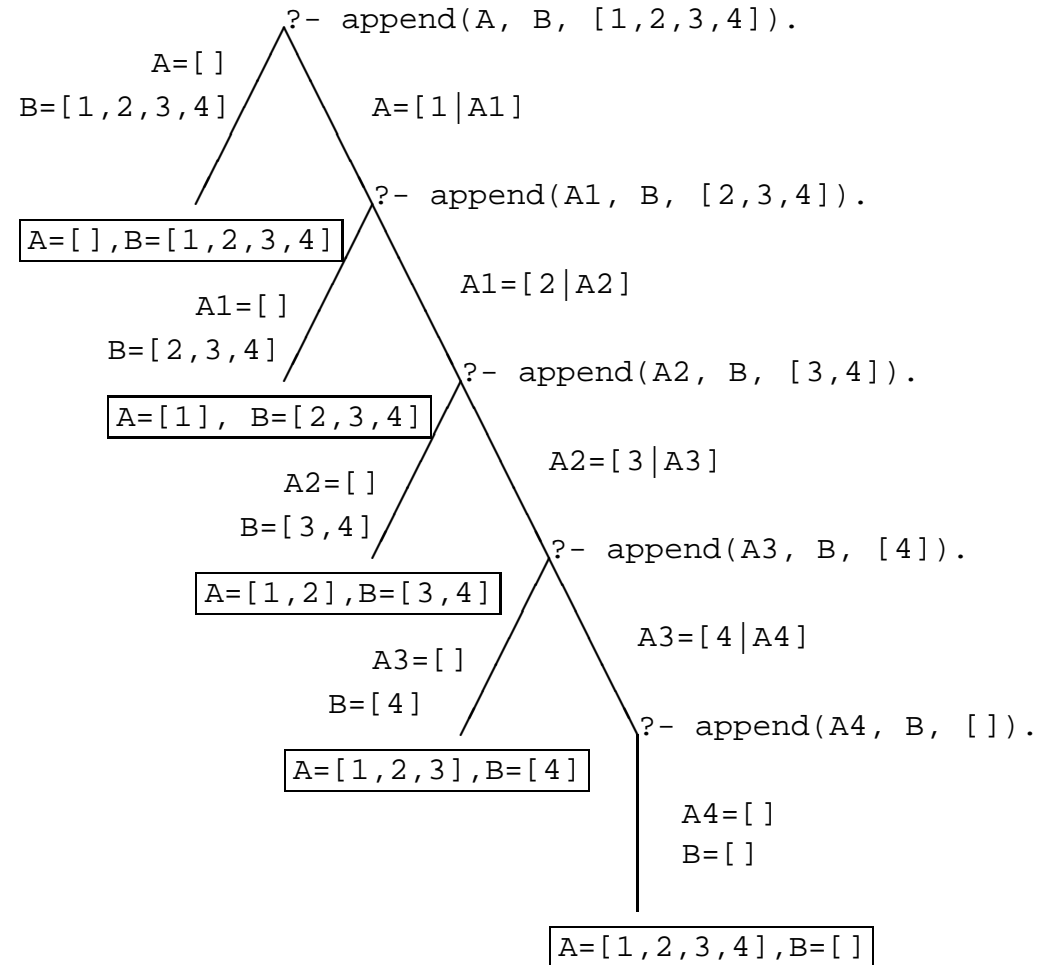
```
list append(list list1, list list2)
{ list list3, *lp = &list3;
  for (list p=list1; p; p=p->next)
  { list newl = new link(p->elem);
    *lp = newl; lp = &newl->next;
  }
  *lp = list2;
  return list3;
}
```

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

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
```



Variációk appendre 1. — Három lista összefűzése

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

- `append(L1, L2, L3, L123): L1 ⊕ L2 ⊕ 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
```

Keresés listában

- `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)
    ).
```

- A `member/2` felhasználási lehetőségei

- Eldöntendő (igen-nem) kérdés:

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

- Lista elemeinek felsorolása:

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

- Listák közös elemeinek felsorolása – mindkét fenti hívásmintát használja:

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

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

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

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

member/2 általánosítása: select/3

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

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

- Felhasználási lehetőségek:

```
| ?- select(1, [2,1,3], L).             % Adott elem elhagyása
    L = [2,3] ? ; no
| ?- select(X, [1,2,3], L).             % Akármelyik elem elhagyása
    L=[2,3], X=1 ? ; L=[1,3], X=2 ? ; L=[1,2], X=3 ? ; no
| ?- select(3, L, [1,2]).               % Adott elem 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` könyvtár tartalmazza a `member/2` és `select/3` eljárások definícióját is.
- A `select/3` keresési tere **véges**, ha 2. és 3. argumentuma közül legalább az egyik zárt végű.

Listák permutációja

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

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

- Felhasználási példák:

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

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

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

- Ha `permutation/2`-ben az első argumentum ismeretlen, akkor a `select` hívás keresési tere végtelen!

AZ EGYESÍTÉSI ALGORITMUS



A Prolog alapvető adatkezelő művelete: az egyesítés

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

Egyesítés: változók behelyettesítése

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

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

Egyesítés: legáltalánosabb egyesítő

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

Az egyesítési algoritmus

- Az egyesítési algoritmus
 - bemenete: két Prolog kifejezés: A és B
 - feladata: a két kifejezés egyesíthetőségének eldöntése
 - eredménye: sikeresség esetén a legáltalánosabb egyesítő ($mgu(A, B)$) 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, \dots, A_N ill. B_1, \dots, B_N , és
 - a. A_1 és B_1 legáltalánosabb egyesítője σ_1 ,
 - b. $A_2\sigma_1$ és $B_2\sigma_1$ legáltalánosabb egyesítője σ_2 ,
 - c. $A_3\sigma_1\sigma_2$ és $B_3\sigma_1\sigma_2$ legáltalánosabb egyesítője σ_3 ,
 - d. ...
 akkor $\sigma = \sigma_1 \otimes \sigma_2 \otimes \sigma_3 \otimes \dots$
 5. Minden más esetben a A és B nem egyesíthető.

Egyesítési példák

● $A = \text{sum_tree}(\text{leaf}(V), V), B = \text{sum_tree}(\text{leaf}(5), S)$

● (4.) A és B neve és argumentumszáma megegyezik

● (a.) $\text{mgu}(\text{leaf}(V), \text{leaf}(5))$ (4., majd 2. szerint) $= \{V \leftarrow 5\} = \sigma_1$

● (b.) $\text{mgu}(V\sigma_1, S) = \text{mgu}(5, S)$ (3. szerint) $= \{S \leftarrow 5\} = \sigma_2$

● tehát $\text{mgu}(A, B) = \sigma_1 \otimes \sigma_2 = \{V \leftarrow 5, S \leftarrow 5\}$

● $A = \text{node}(\text{leaf}(X), T), B = \text{node}(T, \text{leaf}(3))$

● (4.) A és B neve és argumentumszáma megegyezik

● (a.) $\text{mgu}(\text{leaf}(X), T)$ (3. szerint) $= \{T \leftarrow \text{leaf}(X)\} = \sigma_1$

● (b.) $\text{mgu}(T\sigma_1, \text{leaf}(3)) = \text{mgu}(\text{leaf}(X), \text{leaf}(3))$ (4, majd 2. szerint) $= \{X \leftarrow 3\} = \sigma_2$

● tehát $\text{mgu}(A, B) = \sigma_1 \otimes \sigma_2 = \{T \leftarrow \text{leaf}(3), X \leftarrow 3\}$

Egyesítési példák a gyakorlatban

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

- Példák:

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

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

- Kérdés: x és $s(x)$ egyesíthető-e?

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

- Példák:

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

A PROLOG VÉGREHAJTÁSI MECHANIZMUSA



A Prolog végrehajtás eljárásos modelljei

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

A eljárás-redukciós végrehajtási modell

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

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 **törzsén** 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 megghiúsul.
 - egy beépített eljárás segítségével (az első cél beépített eljárást hív):
 - A célsorozatot szétbontjuk az első hívásra és a maradékra.
 - A beépített eljárás hívást végrehajtjuk.
 - Ez lehet sikeres (változó-behelyettesítésekkel), vagy lehet sikertelen.
 - Siker esetén a behelyettesítéseket elvégezzük a célsorozat maradékán.
 - Az új célsorozat: az (első hívás elhagyása után fennmaradó) maradék célsorozat.
 - Ha a beépített eljárás hívása sikertelen, akkor a redukciós lépés megghiúsul.

A Prolog végrehajtási algoritmus

1. *(Kezdeti beállítások:)* A verem üres, $CS := \text{célsorozat}$
2. *(Beépített eljárások:)* Ha CS első hívása beépített akkor hajtsuk végre,
 - a. Ha sikertelen \Rightarrow 6. lépés.
 - b. Ha sikeres, $CS :=$ a redukciós lépés eredménye \Rightarrow 5. lépés.
3. *(Klózszámláló kezdőértékezése:)* $I = 1$.
4. *(Redukciós lépés:)* Tekintsük CS első hívására vonatkoztatható klózok listáját. Ez indexelés nélkül a predikátum összes klóza lesz, indexelés esetén ennek egy megszűrt részsorozata. Tegyük fel, hogy ez a lista N elemű.
 - a. Ha $I > N \Rightarrow$ 6. lépés.
 - b. Redukciós lépés a lista I -edik klóza és a CS célsorozat között.
 - c. Ha sikertelen, akkor $I := I+1 \Rightarrow$ 4. lépés.
 - d. Ha $I < N$ (nem utolsó), akkor vermeljük $\langle CS, I \rangle$ -t.
 - e. $CS :=$ a redukciós lépés eredménye
5. *(Siker:)* Ha CS üres, akkor sikeres vég, egyébként \Rightarrow 2. lépés.
6. *(Sikertelenség:)* Ha a verem üres, akkor sikertelen vég.
7. *(Visszalépés:)* Ha a verem nem üres, akkor leemeljük a veremből $\langle CS, I \rangle$ -t, $I := I+1$, és \Rightarrow 4. lépés.

Indexelés (előzetes)

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

Redukciós modell — előnyök és hátrányok

● Előnyök

- (viszonylag) egyszerű és (viszonylag) precíz definíció
- a keresési tér megjeleníthető, grafikusán szemléltethető

● Hátrányok

- az eljárásokból való kilépést elfedi, pl.

p :- q, r.
 q :- s, t.
 s.
 t.
 r.

G0: p ?
 G1: q, r ?
 G2: s, t, r ?
 G3: t, r ?
 G4: r ? \Leftarrow q-ből való kilépés
 G5: [] ?

- nem jól illeszkedik a Prolog megvalósítások tényleges végrehajtási mechanizmusához
- nem alkalmazható „igazi” Prolog programok nyomkövetésére (hosszú célsorozatok)
- Ezért van létjogosultsága egy másik modellnek:
 - eljárás-doboz (procedure box) modell
 - (szokás még 4-kapus doboz ill. Byrd doboz modellnek is nevezni)
 - a Prolog rendszerek nyomkövető szolgáltatása erre a modellre 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

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

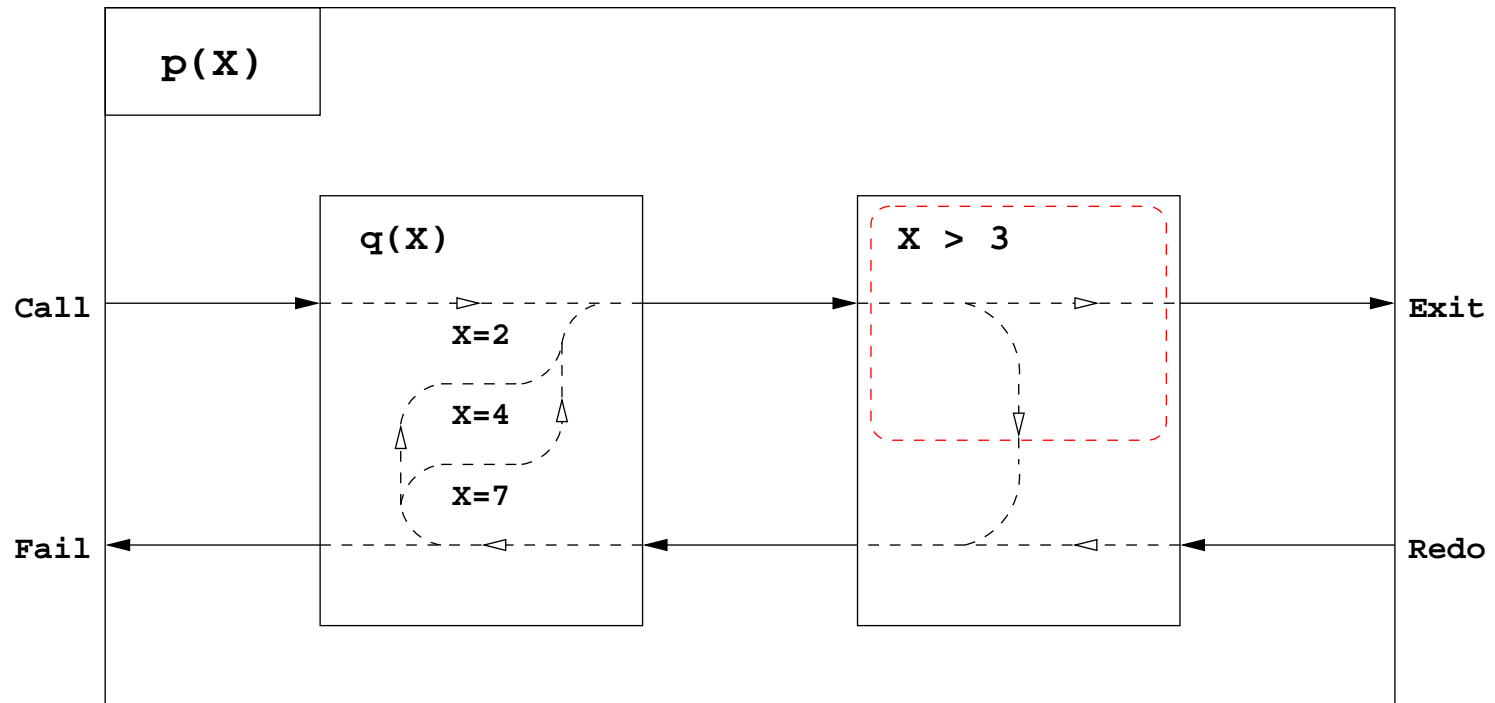
$p(X) :- q(X), X > 3.$

- 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.
```

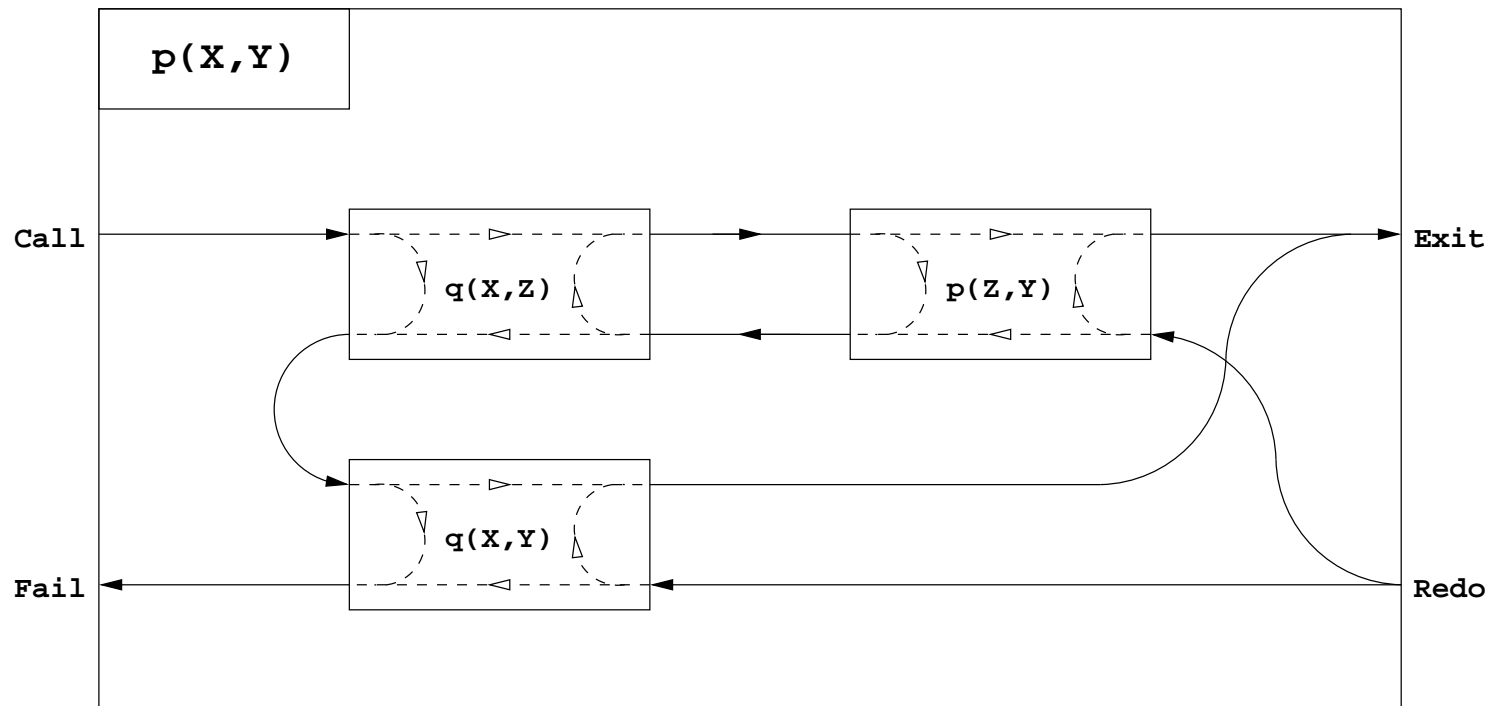
```
| ?- trace, p(X).
      1      1 Call: p(_463) ?
      2      2 Call: q(_463) ?
?      2      2 Exit: q(2) ?           % ? ≡ nemdeterminisztikus
kilépés
      3      2 Call: 2>3 ?
      3      2 Fail: 2>3 ?
      2      2 Redo: q(2) ?           % visszafelé menő végrehajtás
?      2      2 Exit: 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 részjeljárás 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 megghiúsul, akkor **megszüntetjük** az eljáráspéldányt majd ugrunk az előző hívás Újra kapujára, vagy a következő klóz elejére, stb.
- Amikor egy Újra kapuhoz érkezünk, a (*) lépésnél folytatjuk.
- A szülő Újra kapuja (a „következő megoldás” nem első hívása) a tárolt klózsorszámnak megfelelő klózban az utolsó Újra kapura adja a vezérlést.

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

```

boolean p::next()
{ switch(clno) {
  case 0:          // entry point for the Call port
    clno = 1;      // enter clause 1:
    qaptr = new q(x, &z); // create a new instance of subgoal q(X,Z)
  redo1:
    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 redo1;      // and continue at redo port of q(X,Z)
    }
    return TRUE;     // otherwise, exit via the Exit port
  cl2:
    clno = 2;      // enter clause 2:
    qbpPtr = 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(!qbpPtr->next()) { // if q(X,Y) fails
      delete qbpPtr;    // destroy it,
      return FALSE;     // and exit via the Fail port
    }
    return TRUE;     // otherwise, exit via the Exit port
  } }

```

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

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

Visszalépéses keresés — egy aritmetikai példa

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

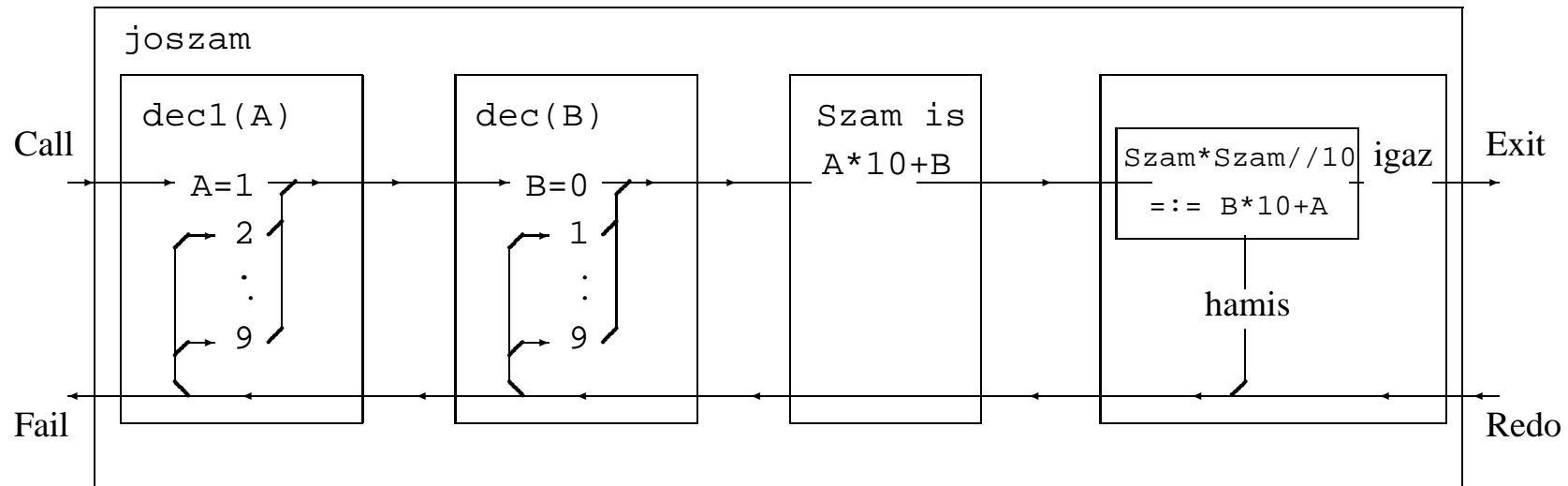
```
% decl(J): J egy pozitív decimális számjegy.  
decl(1). decl(2). decl(3). decl(4).  
decl(5). decl(6). decl(7). decl(8). decl(9).
```

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

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

Prolog végrehajtás — a 4-kapus doboz modell

```
joszam(Szam):-
    dec1(A), dec(B),
    Szam is A * 10 + B, Szam * Szam // 10 ::= B * 10 + A.
```



Visszalépéses keresés — számintervallum felsorolása

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

```
% between(M, N, I): M =< I =< N, I egész.
```

```
between(M, N, M) :-
```

```
    M =< N.
```

```
between(M, N, I) :-
```

```
    M < N,
```

```
    M1 is M+1,
```

```
    between(M1, N, I).
```

```
% dec(X): X egy decimális számjegy
```

```
dec(X) :- between(0, 9, X).
```

```
| ?- between(1, 2, _X), between(3, 4, _Y), Z is 10*_X+_Y.
```

```
Z = 13 ? ;
```

```
Z = 14 ? ;
```

```
Z = 23 ? ;
```

```
Z = 24 ? ;
```

```
no
```

A SICStus eljárás-doboz alapú nyomkövetése — legfontosabb parancsok

● Alapvető nyomkövetési parancsok

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

● A Prolog végrehajtást megváltoztató parancsok

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

● Információ-megjelenítő és egyéb parancsok

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

TOVÁBBI VEZÉRLÉSI SZERKEZETEK



Diszjunkció, példa: az „őse” predikátum

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

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

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

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

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

- A balrekurziót kiküszöbölve kapjuk:

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

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

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

A diszjunkció

- Az `ose1` predikátum hatékonyabbá tehető klózai összevonásával:

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

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

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

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

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

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

A diszjunkció mint szintaktikus édesítőszer

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

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

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

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

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

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

Diszjunkció — megjegyzések

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


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

jelentése: Imre szülője István **ÉS** Imre szülője Gizella.
 - Az **ÉS** kapcsolatban levő klózek alternatív (VAGY kapcsolatban levő) válaszokhoz vezetnek:


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

A „Ki Imre szülője?” kérdésre a válasz: István vagy Gizella.
- A fenti két klózos predikátum átalakítható egyetlen klózzá, diszjunkció segítségével:


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

A konjunkció ezáltal diszjunkcióvá alakult (vö. De Morgan azonosságok).
- Általánosan: tetszőleges predikátum egyklózossá alakítható:
 - a klózeket átalakítjuk azonos fejűvé, új változók és egyenlőségek bevezetésével:


```
szuloje('Imre', Sz) :- Sz = 'István'.
szuloje('Imre', Sz) :- Sz = 'Gizella'.
```
 - a klóztörzseket egy diszjunkcióvá fogjuk össze, amely az új predikátum törzse (lásd (*)).

Negáció

- Feladat: Keressünk (adatbázisunkban) egy olyan szülőt, aki **nem** nagyszülő!
- Ehhez negációra van szükségünk:
 - Meghiúsulásos negáció: a `\+` hívás szerkezet lefuttatja `Hívást`, és pontosan akkor sikerül, ha a `Hívás` meghiúsult.

- Egy megoldás:

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

- Egy ekvivalens megoldás:

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

- Mi történik ha a két hívást megcseréljük?

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

A megghiúsulások negáció (NF — Negation by Failure)

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

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

?- $\backslash +$ X = 1, X = 2.	-----> no
?- X = 2, $\backslash +$ X = 1.	-----> X = 2 ?

Példa: együttható meghatározása lineáris kifejezésben

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

```
% egyhat(Kif, E): A Kif lineáris formulában az x együtthatója E.
egyhat(x, 1).
egyhat(Kif, E) :-
    number(Kif), E = 0.
egyhat(K1+K2, E) :-
    egyhat(K1, E1),
    egyhat(K2, E2),
    E is E1+E2.
egyhat(K1*K2, E) :-
    number(K1),
    egyhat(K2, E0),
    E is K1*E0.
egyhat(K1*K2, E) :-
    number(K2),
    egyhat(K1, E0),
    E is K2*E0.
```

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

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

Együttható meghatározása: többszörös megoldások kiküszöbölése

- negáció alkalmazásával:

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

- hatékonyabban, feltételes kifejezéssel:

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

Feltételes kifejezések

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

```
(...) :-  
    (...),  
    ( felt -> akkor  
    ;   egyébként  
    ),  
    (...).
```

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

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

Feltételes kifejezések (folyt.)

- Procedurális szemantika

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

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

- Többszörös elágaztatás skatulyázott feltételes kifejezésekkel:

<pre>(felt1 -> akkor1 ; felt2 -> akkor2 ; ...)</pre>	<pre>(felt1 -> akkor1 ; (felt2 -> akkor2 ;))</pre>
--	---

- Az egyébként rész elhagyható, alapértelmezése: `fail`.
- A `\+ felt` negáció kiváltható a `(felt -> fail ; true)` feltételes kifejezéssel.

Feltételes kifejezés — példák

- Faktoriális

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

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

- Szám előjele

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


TÍPUSOK PROLOGBAN



Típusok leírása Prologban

- Típusleírás: (tömör) Prolog kifejezések egy halmazának megadása

- Alaptípusok leírása: `int`, `float`, `number`, `atom`, `any`

- Új típusok felépítése:

$$\{ \text{str}(T_1, \dots, T_n) \} \equiv \{ \text{str}(e_1, \dots, e_n) \mid e_1 \in T_1, \dots, e_n \in T_n \}, n \geq 0$$

Példa: `{személy(atom,atom,int)}` az olyan `személy/3` funktorú struktúrák halmaza, amelyben az első két argumentum `atom`, a harmadik egész.

- Típusok, mint halmazok úniója képezhető a `\/` operátorral.

$$\{ \text{személy}(\text{atom}, \text{atom}, \text{int}) \} \setminus / \{ \text{atom-atom} \} \setminus / \text{atom}$$

- Egy típusleírás elnevezhető (kommentben): `:- type tnév == tleírás.`

```
:- type t1 == {atom-atom} \/ atom.,
```

```
:- type ember == {ember-atom} \/ {semmi}.
```

- Megkülönböztetett únió: csupa különböző funktorú összetett típus úniója. Ha S_1, \dots, S_n mind különböző funktorú, alkalmazható az egyszerűsített (Mercury) jelölés:

```
:- type T == { S1 } \/ ... \/ { Sn }.  $\Rightarrow$  :- type T ---> S1 ; ... ; Sn. Példák:
```

```
:- type ember ---> ember-atom; semmi.
```

```
:- type fa ---> leaf(int) ; node(fa,fa).
```

Típusok leírása Prologban — folytatás

● Paraméteres típusok — példák

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

● Típusdeklarációk szintaxisa

```
⟨ típusdeklaráció ⟩ ::= ⟨ típuselnevezés ⟩ | ⟨ típuskonstrukció ⟩
⟨ típuselnevezés ⟩ ::= :- type ⟨ típusazonosító ⟩ == ⟨ típusleírás ⟩ .
⟨ típuskonstrukció ⟩ ::= :- type ⟨ típusazonosító ⟩ ---> ⟨ megkülönb. únió ⟩ .
⟨ megkülönb. únió ⟩ ::= ⟨ konstruktor ⟩ ; ...
⟨ konstruktor ⟩ ::= ⟨ névkonstans ⟩ | ⟨ struktúranév ⟩ ( ⟨ típusleírás ⟩ , ... )
⟨ típusleírás ⟩ ::= ⟨ típusazonosító ⟩ | ⟨ típusváltozó ⟩ | { ⟨ konstruktor ⟩ } |
                  ⟨ típusleírás ⟩ \ / ⟨ típusleírás ⟩
⟨ típusazonosító ⟩ ::= ⟨ típusnév ⟩ | ⟨ típusnév ⟩ ( ⟨ típusváltozó ⟩ , ... )
⟨ típusnév ⟩ ::= ⟨ névkonstans ⟩
⟨ típusváltozó ⟩ ::= ⟨ változó ⟩
```

Predikátumtípus-deklarációk

- Predikátumtípus-deklaráció

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

- Példa:

```
:- pred sum_tree(tree(int), int).
```

- Predikátummód-deklaráció (Nem kötelező, több is megadható.)

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

(Mercury-ban az inout módazonosító nem megengedett.)

- Példák:

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

- Vegyes típus- és móddeklaráció

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

- Példa:

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

Móddeklaráció: a SICStus kézikönyv által használt alak

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

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

- Mód-jelölő karakterek:

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

A PROLOG SZINTAXIS



A Prolog szintaxis összefoglalása

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

A Prolog nyelv-változatok

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

Szintaktikus édesítőszerek — összefoglalás, gyakorlati tanácsok

- Operátoros kifejezések alapstruktúra alakra hozása

- Zárójelezzük be a kifejezést, az operátorok prioritása és fajtája alapján, például $-a+b*2 \Rightarrow ((-a)+(b*2))$.

- Hozzuk az operátoros kifejezéseket alapstruktúra alakra:

$(A \text{ Inf } B) \Rightarrow \text{Inf}(A, B)$, $(\text{Pref } A) \Rightarrow \text{Pref}(A)$, $(A \text{ Postf}) \Rightarrow \text{Postf}(A)$

Példa: $((-a)+(b*2)) \Rightarrow (-a) + *(b, 2) \Rightarrow +(-a), *(b, 2)$.

- Trükkös esetek:

- A vesszőt névként idézni kell: pl. $(pp, (qq; rr)) \Rightarrow ', '(pp, i(qq, rr))$.

- $- \text{Szám} \Rightarrow$ negatív számkonstans, de $- \text{Egyéb} \Rightarrow$ prefix alak.

Példa: $-1+2 \Rightarrow +(-1, 2)$, de $-a+b \Rightarrow +(-a), b$.

- $\text{Név}(\dots) \Rightarrow$ struktúrakifejezés;

$\text{Név}(\dots) \Rightarrow$ prefix operátoros kifejezés. Példák:

$-(1, 2) \Rightarrow -(1, 2)$ (változatlan), de

$-(1, 2) \Rightarrow -(' , '(1, 2))$.

Szintaktikus édesítőszer — listák, egyebek

- Listák alapstruktúra alakra hozása

- Farok-megadás betoldása.

$[1,2] \Rightarrow [1,2|[]]$. $[[X|Y]] \Rightarrow [[X|Y]|[]]$

- Vessző (ismételt) kiküszöbölése $[Elem1,Elem2\dots] \Rightarrow [Elem1|[Elem2\dots]]$.

$[1,2|[]] \Rightarrow [1|[2|[]]]$

$[1,2,3|[]] \Rightarrow [1|[2,3|[]]] \Rightarrow [1|[2|[3|[]]]]$

- Strukturakifejezéssé alakítás: $[Fej|Farok] \Rightarrow .(Fej,Farok)$.

$[1|[2|[]]] \Rightarrow .(1,.(2,[]))$, $[[X|Y]|[]] \Rightarrow .(. (X,Y), [])$

- Egyéb szintaktikus édesítőszer:

- Karakterkód-jelölés: $0'Kar$.

$0'a \Rightarrow 97$, $0'b \Rightarrow 98$, $0'c \Rightarrow 99$, $0'd \Rightarrow 100$, $0'e \Rightarrow 101$

- Füzér (string): $"xyz\dots" \Rightarrow$ az $xyz\dots$ karakterek kódját tartalmazó lista

$"abc" \Rightarrow [97,98,99]$, $"" \Rightarrow []$, $"e" \Rightarrow [101]$

- Kapcsos zárójelezés: $\{Kif\} \Rightarrow \{\}(Kif)$ (egy $\{\}$ nevű, egyargumentumú struktúra — a $\{\}$ jelpár egy önálló lexikai elem, egy névkonstans).

- Bináris, hexa stb. alak (csak `iso` módban), pl. `0b101010`, `0x1a`.

Kifejezések szintaxisa — kétszintű nyelvtanok

- Egy részlet egy „hagyományos” nyelv kifejezés-szintaxisából:

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

- Ugyanez kétszintű nyelvtannal:

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

Prolog kifejezések szintaxisa

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

Kifejezések szintaxisa — folytatás

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

Kifejezések szintaxisa — megjegyzések

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

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

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

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

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

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

A Prolog lexikai elemei 1. (ismétlés)

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

A Prolog lexikai elemei 2.

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

Megjegyzések és formázó-karakterek

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

PROLOG PÉLDÁK



A régi jegyzet bevezető példája: útvonalkeresés

- A feladat:

- Tekintsük (autóbusz)járatok egy halmazát.
- Mindegyik járhoz a két végpont és az útvonal hossza van megadva.
- Írjunk Prolog eljárást, amellyel megállapítható, hogy két pont összeköthető-e pontosan N csatlakozó járattal!

- Átfogalmazás: egy súlyozott irányítatlan gráfban két pont közötti utat keresünk. Élek:

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

- Irányított élek:

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

Az útvonalkeresési feladat — folytatás

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

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

- Futási példa:

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

Körmentes út keresése

- Könyvtár betöltése, adott funktorú eljárások importálásával:

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

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

```
% útvonal_2(N, A, B, H): A és B között van (pontosan)
```

```
% N szakaszból álló körmentes útvonal, amelynek összhossza H.
```

```
útvonal_2(N, Honnan, Hová, H) :-
```

```
    útvonal_2(N, Honnan, Hová, [Honnan], H).
```

```
% útvonal_2(N, A, B, Kizártak, H): A és B között van pontosan
```

```
% N szakaszból álló körmentes, Kizártak elemein át nem menő H hosszú út.
```

```
útvonal_2(0, Hová, Hová, Kizártak, 0).
```

```
útvonal_2(N, Honnan, Hová, Kizártak, H) :-
```

```
    N > 0, N1 is N-1, útszakasz(Honnan, Közben, H1),
```

```
    \+ member(Közben, Kizártak),
```

```
    útvonal_2(N1, Közben, Hová, [Közben/Kizártak], H2), H is H1+H2.
```

- Példa-futás:

```
| ?- útvonal_2(2, 'Párizs', Hová, H).
```

```
    H = 1900, Hová = 'Berlin' ? ;
```

```
    H = 1510, Hová = 'Budapest' ? ; no
```

Továbbfejlesztés: körmentes út keresése, útvonal-gyűjtéssel

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

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

```
% útvonal_3(N, A, B, Út, H): A és B között van (pontosan)
```

```
% N szakaszból álló körmentes Út útvonal, amelynek összhossza H.
```

```
útvonal_3(N, Honnan, Hová, Út, H) :-
```

```
    útvonal_3(N, Honnan, Hová, [Honnan], FÚt, H),
    reverse(FÚt, Út).
```

```
% útvonal_3(N, A, B, FÚt0, FÚt, H): A és B között van pontosan
```

```
% N szakaszból álló körmentes, FÚt0 elemein át nem menő H hosszú út.
```

```
% FÚt = (az A → B útvonal megfordítása) ⊕ FÚt0.
```

```
útvonal_3(0, Hová, Hová, FordÚt, FordÚt, 0).
```

```
útvonal_3(N, Honnan, Hová, FordÚt0, FordÚt, H) :-
```

```
    N > 0, N1 is N-1, útszakasz(Honnan, Közben, H1),
```

```
    \+ member(Közben, FordÚt0),
```

```
    útvonal_3(N1, Közben, Hová, [Közben/FordÚt0], FordÚt, H2), H is H1+H2.
```

```
| ?- útvonal_3(2, 'Párizs', _, Út, H).
```

```
    H = 1900, Út = ['Párizs', 'Bécs', 'Berlin'] ? ;
```

```
    H = 1510, Út = ['Párizs', 'Bécs', 'Budapest'] ? ; no
```

Súlyozott gráf ábrázolása éllistával

- A gráf ábrázolása

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

- Típus-definíció

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

- Példa

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

Ismétlődésmentes útvonal keresése listával ábrázolt gráfban

```

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

% útvonal_4(N, G, A, B, L, H): A G gráfban van egy A-ból
% B-be menő N szakaszból álló L út, melynek összhossza H.
útvonal_4(0, _Gráf, Hová, Hová, [Hová], 0).
útvonal_4(N, Gráf, Honnan, Hová, [Honnan|Út], H) :-
    N > 0, N1 is N-1,
    select(Él, Gráf, Gráf1),
    él_végpontok_hossz(Él, Honnan, Közben, H1),
    útvonal_4(N1, Gráf1, Közben, Hová, Út, H2),
    H is H1+H2.

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

| ?- hálózat(_Gráf), útvonal_4(2, _Gráf, 'Budapest', _, Út, H).
    H = 880, Út = ['Budapest', 'Bécs', 'Berlin'] ? ;
    H = 1510, Út = ['Budapest', 'Bécs', 'Párizs'] ? ;
    no

```


Bináris fákra vonatkozó példasor — fa levele

- Ismétlés: egészekből álló bináris fa:

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

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

- `% fa_levele(Fa, Ertek): A Fa bináris fa levelében szerepel az Ertek szám.`
`fa_levele(leaf(V), V). % ha a fa egyetlen levélből áll és a levélbeli`
`% érték megegyezik a keresettel, akkor ``siker```
`fa_levele(node(L,_), V) :-`
`fa_levele(L, V). % ha a bal részében van, akkor az egészben is`
`fa_levele(node(_,R), V) :-`
`fa_levele(R, V). % ha a jobb részében van, akkor az egészben is`

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

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

Összetett adatstruktúrák konjunktív és diszjunktív bejárása

- Prologban egy összetett adatstruktúrát kétféleképpen lehet bejárni:

- konjunktívan: a részek bejárása **ÉS** kapcsolatban van, általában egy eredményt ad

- pl. fa összegzése (`sum_tree`), fa ellenőrzése (`itree`), fa kiírása:

```
% faki(Fa): Fa kiírható (mindig teljesül :-). Mellékhatásként kiírja a Fa fát.
faki(leaf(V)) :-
    write(@), write(V).    % A write(X) beépített pred. kiírja az X kifejezést.
faki(node(L,R)) :-
    write('('), faki(L), write(' -- '), faki(R), write(')').
```

```
| ?- faki(node(node(leaf(1),leaf(8)),leaf(7))).    => ((@1 -- @8) -- @7)
yes
```

- diszjunktívan: a részek bejárása **VAGY** kapcsolatban van, visszalépéskor új eredmény

- pl. fa leveleinek felsorolása (`fa_levele`)

- A diszjunktív, felsoroló bejárás könnyen kiegészíthető további feltételekkel

- Keressük egy fának az (5,10) intervallumba eső leveleit:

```
| ?- _Fa = node(node(leaf(1),leaf(8)),leaf(7)), fa_levele(_Fa, E), 5 < E, E < 10.
    => E = 8 ? ; E = 7 ? ; no
| ?- _Fa = (...), fa_levele(_Fa, E), 5 < E, E < 10, write(E), write(' '), fail.
    => 8 7 => no
```

- A `fail` beépített predikátum mindig meghiúsul, pl. ún. visszalépéses ciklus szervezésére jó.

Levél elhagyása bináris fából

- Írjunk egy predikátumot annak eldöntésére, hogy egy adott érték szerepel-e egy összetett fa levelében! A predikátum adja vissza a levél elhagyása után fennmaradó fát!

```
% flm(Fa, Ertek, Marad): A Fa összetett bináris fa egy Ertek értékű
% levelének elhagyása után marad a Marad fa. (flm = fa_level_maradek)
flm(node(leaf(V),T), V, T).    % ha a bal részfa a keresett levél
                             % akkor a jobb részfa a maradék
flm(node(T,leaf(V)), V, T).    % ugyanez jobboldali levél esetére
flm(node(L0,R), V, node(L,R)) :-
    flm(L0, V, L).            % ha a bal részfából elhagyható a levél
                             % akkor ennek maradéka, kiegészítve
                             % a jobb részfával, lesz a teljes fa maradéka
flm(node(L,R0), V, node(L,R1)) :-
    flm(R0, V, R1).          % ugyanez jobb részfa esetére
```

- Az flm/3 predikátum használható ellenőrzése, de fa szétbontására is:

```
| ?- flm(node(leaf(1),node(leaf(2),leaf(3))), 2, T). ==>
  T = node(leaf(1),leaf(3)) ? ; no
| ?- flm(node(leaf(1),node(leaf(2),leaf(3))), 7, T). ==> no
| ?- flm(node(leaf(1),node(leaf(2),leaf(3))), X, T). ==>
  T = node(leaf(2),leaf(3)), X = 1 ? ;
  T = node(leaf(1),leaf(3)), X = 2 ? ;
  T = node(leaf(1),leaf(2)), X = 3 ? ; no
```

Levél beszúrása bináris fába

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

- Nem kell írunk, már megírtuk! Az `flm` predikátum erre is jó:

*% flm(Fa, Ertek, Marad): A Fa összetett bináris fa egy Ertek értékű
% levelének elhagyása után marad a Marad fa. Röviden: Fa - Ertek = Marad.*

*% flm(Fa, Ertek, Marad): A Fa (összetett) bináris fa úgy áll elő, hogy
% a Marad fába beszúrunk egy E értékű levelet. Fa = Marad + Ertek.*

`flm(node(leaf(V),T), V, T). % Egy T fába beszúrhatunk egy levelet
(...) % úgy, hogy az egylevelű fát T elé tesszük`

- Példák:

```
| ?- flm(Fa, 2, leaf(1)), faki(Fa), write(' '), fail.  
(@2 -- @1) (@1 -- @2)                               => no  
| ?- flm(Fa0, 2, leaf(1)), flm(Fa, 3, Fa0), faki(Fa), write(' '), fail.  
(@3 -- (@2 -- @1)) ((@2 -- @1) -- @3) ((@3 -- @2) -- @1) ((@2 -- @3) -- @1)  
(@2 -- (@3 -- @1)) (@2 -- (@1 -- @3)) (@3 -- (@1 -- @2)) ((@1 -- @2) -- @3)  
((@3 -- @1) -- @2) ((@1 -- @3) -- @2) (@1 -- (@3 -- @2)) (@1 -- (@2 -- @3)) => no  
  
negylevelu(X, Y, Z, U, Fa) :- % Fa az X, Y, Z, U levelekből áll  
    flm(Fa0, Y, leaf(X)), flm(Fa1, Z, Fa0), flm(Fa, U, Fa1).  
  
| ?- findall(Fa, negylevelu(1,3,4,6,Fa), Fak), length(Fak,Db). => Db = 120, Fak = (...)
```

Példa: adott értékű kifejezés előállítás

- A feladat: írjunk Prolog programot a következő feladvány megoldására:
 - Az 1, 3, 4, 6 számokból a négy alapművelet felhasználásával állítsuk elő a 24 számértéket!
 - Mind a négy számot fel kell használni, tetszőleges sorrendben.
 - Tetszőleges alapműveletek használhatók, tetszőlegesen zárójelezéssel.
- Már van egy predikátumunk (`negylevelu/5`), amely adott számokból tetszőleges fát épít.
- Definiáljunk egy predikátumot, amely egy fának megfelelő aritmetikai kifejezéseket készít!

```
% fa_kif(Fa, Kif): Kif a Fa fával azonos alakú, azonos számokból álló
% aritmetikai kifejezés, amelyben a négy alapművelet fordulhat elő.
```

```
fa_kif(leaf(V), V).
fa_kif(node(L,R), Exp) :-
    fa_kif(L, E1),
    fa_kif(R, E2),
    alap4(E1, E2, Exp).
```

```
% alap4(X, Y, Kif): Kif az X és Y kifejezésekből a négy alapművelet egyikével áll elő.
alap4(X, Y, X+Y).          alap4(X, Y, X-Y).
alap4(X, Y, X*Y).          alap4(X, Y, X/Y).
```

```
| ?- fa_kif(node(leaf(1),node(leaf(2),leaf(3))), Kif).
Kif = 1+(2+3) ? ; Kif = 1-(2+3) ? ; Kif = 1*(2+3) ? ; Kif = 1/(2+3) ? ;
(...)
Kif = 1+2/3 ? ; Kif = 1-2/3 ? ; Kif = 1*(2/3) ? ; Kif = 1/(2/3) ? ; no
```

Példa: adott értékű kifejezés előállítás (folyt.)

- Korábban elkészített predikátumok:

- adott számokból álló fákat felsoroló `negylevelu/5`
- adott fával azonos szerkezetű aritmetikai kifejezéseket felsoroló `fa_kif/2`

- Ezekre építve könnyen megírható a feladvány megoldására használható predikátum:

```
% Kif egy a négy alapművelettel az X, Y, Z, U számokból
% felépített kifejezés, amelynek értéke Ertek.
```

```
negylevelu_erteke(X, Y, Z, U, Ertek, Kif) :-
    negylevelu(X, Y, Z, U, Fa),
    fa_kif(Fa, Kif),
    Kif ::= Ertek.
```

```
| ?- negylevelu_erteke(1,3,4,6,24,Kif).
```

```
...
```

- Megjegyzések

- Az aritmetikai eljárásokban a változók nem csak számokra, hanem tömör aritmetikai kifejezésekre is be lehetnek helyettesítve.
- A `negylevelu_erteke` eljárás utolsó hívása helyett **nem** lenne jó: `Ertek is Kif`. Miért?