

A logikai programozás alap gondolata

BEVEZETÉS A LOGIKAI PROGRAMOZÁSBA

- 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-klózo** nyelve,
 - végrehajtási mechanizmusa: **mintaillesztés**es eljáráshíváson alapuló **visszalépés**es keresés.

Deklaratív programozás. BME VIK, 2008. őszi félév

(Logikai Programozás)

LP-3

Az előadás LP részének áttekintése

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

Deklaratív programozás. BME VIK, 2008. őszi félév

(Logikai Programozás)

LP-4

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 Nagyhatékonyágú Prolog fordítóprogramok

Deklaratív programozás. BME VIK, 2008. őszi félév

(Logikai Programozás)

Információk a logikai programozásról

- A legfontosabb 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

PROLOG: EGY KIS GYAKORLATI BEMUTATÁS

Példák

- adatbázis jellegű: családi kapcsolatok
- aritmetika: faktoriális
- adatstruktúrák: bináris fák

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

• Feladatok:

- Definiálandó az unoka–nagyözülő kapcsolat, pl. keressük egy adott személy nagyözüleit.
- Definiálandó az leszármazott–ős kapcsolat, pl. keressük egy adott személy őseit.

A nagyözülő feladat — C nyelvű megoldás

```

/* Az adatbázis */
struct gysz {
    char *gyerek, *szulo;
} szulok[] = {
    "Imre",    "István",
    "Imre",    "Gizella",
    "István",  "Géza",
    "István",  "Sarolt",
    "Gizella", "Civakodó Henrik",
    "Gizella", "Burgundi Gizella",
    NULL,     NULL
};

/* unoka nagyözüleinek kiírása */
void nagyözuloi(char *unoka)
{
    struct gysz *mgysz = szulok;
    for (; mgysz->gyerek; ++mgysz)
        if (!strcmp(unoka, mgysz->gyerek))
            { struct gysz *mszn = szulok;
              for (; mszn->gyerek; ++mszn)
                  if (!strcmp(mgysz->szulo,
                              mszn->gyerek))
                      puts(mszn->szulo);
            }
}

```

A nagyözülő 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 nagyözülője Nagyszulo.
nagyözuloje(Gyerek, Nagyszulo) :-
    szuloje(Gyerek, Szulo),
    szuloje(Szulo, Nagyszulo).

% Kik Imre nagyözülei?
| ?- nagyözuloje('Imre', NSz).
NSz = 'Géza' ? ;
NSz = 'Sarolt' ? ;
NSz = 'Civakodó Henrik' ? ;
NSz = 'Burgundi Gizella' ? ;
no

% Kik Géza unokái?
| ?- nagyözuloje(U, 'Géza').
U = 'Imre' ? ;
no

```

A Prolog és az adatbáziskezelés

- Miben különbözik a Prolog egy adatbáziskezelőtől
- Mivel több?
 - rekurzió
 - összetett adatszerkezetek
- De: a Prolog egy programozási nyelv
 - pl. nem optimalizálja a részkérdések sorrendjét

Az őse – rekurzív – kapcsolat

- Egy egyszerű megoldás
 - A szülő ős.
 - A szülő őse is ős.

```
% ose(Lesz, Os): A Lesz leszármazott őse Os
ose(Lesz, Os) :- szuloje(Lesz, Os).
ose(Lesz, Os) :- szuloje(Lesz, Sz),
                 ose(Sz, Os).
```

- Egy alternatív megoldás, diszjunkció használatával

- Tekintsük sorra a szülőket.

- Az adott szülő ős.
- Az adott szülő őse is ős.

```
ose1(Lesz, Os) :-
    szuloje(Lesz, Sz),
    ( Os = Sz
    ; ose1(Sz, Os)
    ).
```

A Prolog végrehajtási mechanizmusa dióhéjban

- A Prolog eljárásos szemléletben
 - Egy eljárás: azon klózek összesége, amelyek fejének neve és argumentumszáma azonos.
 - Egy klóz: `Fej :- Törzs`, ahol `Törzs` egy célsorozat
 - Egy célsorozat: C_1, \dots, 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 klózeket 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)

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.
```

Néhány beépített predikátum

- Kifejezések egyesítése: $x = y$: az x és y **szimbolikus** kifejezések változók behelyettesítésével azonos alakra hozhatók (és el is végzi a behelyettesítéseket).
- Kifejezések nem-egyesíthetősége: $x \neq y$: az x és y kifejezések nem egyesíthetőek.
- Aritmetikai predikátumok
 - x is Kif : A Kif **aritmetikai** kifejezést kiértékeli és **értékét** egyesíti x -szel.
 - $Kif1 < Kif2$, $Kif1 = < Kif2$, $Kif1 > Kif2$, $Kif1 >= Kif2$, $Kif1 = = Kif2$, $Kif1 \neq Kif2$: A $Kif1$ és $Kif2$ aritmetikai kifejezések értéke a megadott relációban van egymással ($= =$ jelentése: aritmetikai egyenlőség, \neq 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 meghiúsul.
 - $trace$, $notrace$: 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** kevésbe pontosan nyomkövethető.
- $halt$: A Prolog rendszer befejezi működését.

```
> sicstus
SICStus 3.12.7 (x86-linux-glibc2.3): Fri Oct 6 00:10:34 CEST 2006
| ?- consult(szulok).
% consulted /home/user/szulok.pl in module user, 0 msec 376 bytes
yes
| ?- nagyszuloje('Imre', 'Istvan').
no
| ?- listing(nagyszuloje).
(...)
yes
| ?- halt.
>
```

Adatstruktúrák Prologban — 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

<pre>% 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; };</pre>	<pre>% Adattípus-leírás Prologban % (ún. Mercury jelölés): % :- type tree ---> % node(tree, tree) % leaf(int).</pre>
--	---

Bináris fák összegzése

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

<pre>% 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); } }</pre>	<pre>% Prolog eljárás (predikátum) tree_sum(leaf(Value), Value). tree_sum(node(Left,Right), S) :- tree_sum(Left, S1), tree_sum(Right, S2), S is S1+S2.</pre>
--	--

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
| ?- tree_sum(node(leaf(5),
                    node(leaf(3), leaf(2))), Sum).
Sum = 10 ? ;
no
| ?- tree_sum(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:

```
tree_sum(leaf(Value), Value).
tree_sum(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.
    tree_sum(Left, S1),
    tree_sum(Right, S2).
```

• Az eljárás futása:

```
| ?- tree_sum(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'`, `szuloje`, `+`, `-`, `<`, `sum_tree`

• összetett- vagy struktúra-kifejezés (*compound*)

- ún. kanonikus alak: $\langle \text{struktúranév} \rangle (\langle \text{arg}_i \rangle, \dots)$
 - 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 behelyettesíthető, értékkel nem bír, az egyesítés (mintaillesztés) művelete során egy tetszőleges Prolog kifejezést vehet fel értékül (akár egy másik változót)

Predikátumok, klózek

• Példa:

```
% két klózból álló predikátum definíciója, funktora: tree_sum/2
tree_sum(leaf(Val), Val).           % 1. klóz, tényállítás
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>
```

Deklaratív programozás. BME VIK, 2008. őszi félév

(Logikai Programozás)

A PROLOG NYELV KÖZELÍTŐ SZINTAXISA

LP-27

Prolog programok formázása

• Programok javasolt formázása:

- Az egy predikátumhoz tartozó klózek 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

Deklaratív programozás. BME VIK, 2008. őszi félév

(Logikai Programozás)

LP-28

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 kifejezés> | {Funktora: <struktúranév>/<arg.szám>}
              <egyéb kifejezés> | {Operátoros, lista, zárójeles, ld. később}

<konstans> ::= <névkonstans> |
              <számkonstans>

<számkonstans> ::= <egész szám> |
                  <lebegőpontos szám>

<összetett kifejezés> ::= <struktúranév> ( <argumentum>, ... )
<struktúranév> ::= <névkonstans>
<argumentum> ::= <kifejezés>
```

Deklaratív programozás. BME VIK, 2008. őszi félév

(Logikai Programozás)

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

LISTA, MINT SZINTAKTIKUS „ÉDESÍTŐSZER”

A Prolog lista-fogalma

A Prolog lista

- Az üres lista `[]` 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([]).

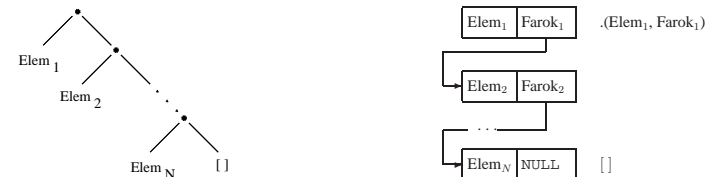
| ?- számlista([1,2]).    % [1,2] == .(1,.(2,[])) == [1|[2|[]]]
yes
| ?- számlista([1,a,f(2)]).
no
```

Listák írásmódjai

Egy N elemű lista lehetséges írásmódjai:

- alapstruktúra-alak: `.(Elem1, .(Elem2, ..., .(ElemN, [])...))`
- ekvivalens lista-alak: `[Elem1, Elem2, ..., ElemN]`
- kevésbé kényelmes ekvivalens alak: `[Elem1 | [Elem2 | ... | [ElemN | []] ...]]`

A listák fastruktúra alakja és megvalósítása



Listák jelölése — szintaktikus édesítőszerek

- 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 fark $[]$: $[Elem_1, Elem_2, \dots, Elem_N] \equiv [Elem_1, Elem_2, \dots, Elem_N | []]$
- | ?- [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]). $\Rightarrow A=3, B=[6]/3, X=3, Y=[6] ?$

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
- Erlang-ban 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).  $\Rightarrow L = [X,X|_A] ? ; no$ 
```

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

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

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

- (Ismétlés:) Tömör (ground) kifejezés: változót nem tartalmazó kifejezés
- Minta: egy általában nem 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, ...

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].
```

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

```
> append([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([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 $append0/append(L1, \dots)$ komplexitása: futási ideje arányos $L1$ hosszával.

- Miért jobb az $append/3$ mint az $append0/3$?

- $append/3$ **jobbrekurzív**, ciklussal ekvivalens (nem fogyaszt vermet)
- $append([1, \dots, 1000], [0], [2, \dots])$ azonnal, $append0(\dots)$ 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 fejtét!
(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 nyomkövető programozására, új parancsokra
- új parancs: `'N <név>'` — fókuszált argumentum elnevezése
- szabványos parancs: `'^ <argszám>'` — adott argumentumra fókuszálás
- új parancs: `'P [<név>]'` — adott nevű (ill összes) kifejezés kiírása

```
| ?- 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]) ? => 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
```

Deklaratív programozás. BME VIK, 2008. őszi félév

(Logikai Programozás)

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)).
```

Deklaratív programozás. BME VIK, 2008. őszi félév

(Logikai Programozás)

`append` és `revapp` — listák gyűjtési iránya

- Prolog megvalósítás

<pre>append([], L, L). append([X L1], L2, [X L3]) :- append(L1, L2, L3).</pre>		<pre>revapp([], L, L). revapp([X L1], L2, L3) :- revapp(L1, [X L2], L3).</pre>
--	--	--

- C++ megvalósítás

<pre>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; }</pre>		<pre>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; }</pre>
--	--	---

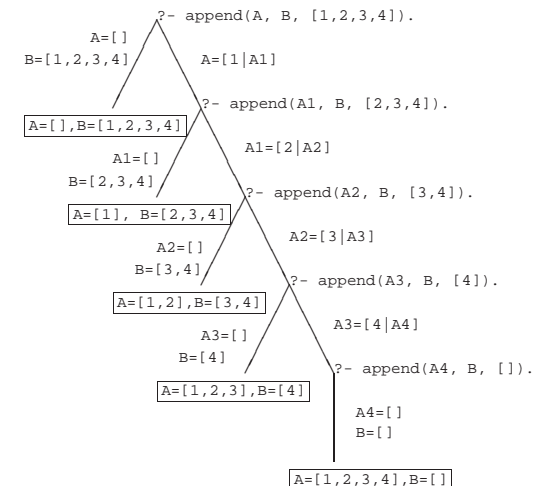
Deklaratív programozás. BME VIK, 2008. őszi félév

(Logikai Programozás)

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
```



Deklaratív programozás. BME VIK, 2008. őszi félév

(Logikai Programozás)

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 behelyettesíthetősége (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 beszűrése!
    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]).
                                % Beszűrhető-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!

OPERÁTOROK, MINT SZINTAKTIKUS „ÉDESÍTŐSZER”

Operátor-kifejezések

- Példa:

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

- Operátoros kifejezések

```
⟨összetett kifejezés⟩ ::=
    ⟨struktúranév⟩ (⟨argumentum⟩, ...)           {eddig csak ez volt}
  | ⟨argumentum⟩ ⟨operátornév⟩ ⟨argumentum⟩     {infix kifejezés}
  | ⟨operátornév⟩ ⟨argumentum⟩                 {prefix kifejezés}
  | ⟨argumentum⟩ ⟨operátornév⟩                 {posztfix kifejezés}

⟨operátornév⟩ ::= ⟨struktúranév⟩                {ha operátorként lett definiálva}
```

- Operátor-kezelő beépített predikátumok:

- `op(Prioritás, Fajta, OpNév)` vagy `op(Prioritás, Fajta, [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 (írasmódot) és az asszociativitást:

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 asszociativitás 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 fajtánévben az y betű mutatja az asszociativitás 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 rész kifejezés 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:
 - $:- op(500, xfy, +^).$
 - $| ?- :- write((1 +^ 2) + 3), nl. \Rightarrow (1+^2)+3$
 - $| ?- :- write(1 +^ (2 + 3)), nl. \Rightarrow 1+^2+3$
 - 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, @).
tree_sum(@V, V).             (...)
```
- A „vessző” kettős szerepe
 - struktúra-kifejezés argumentumait választja el
 - 1000 prioritású xfy operátorként működik pl.: $(p :- a,b,c) = :- (p, ', '(a, ', '(b,c))$
 - a „pucér” vessző (,) nem névkonstans, de operátorként aposztrófok nélkül is írható.
 - struktúra-argumentumban 999-nél nagyobb prioritású kifejezést zárójelezni kell:


```
| ?- write_canonical((a,b,c)). \Rightarrow ', '(a, ', '(b,c)
| ?- write_canonical(a,b,c). \Rightarrow ! procedure write_canonical/3 does not exist
```
- Az egyértelmű elemezhetőség érdekében a Prolog szabvány kiköti, hogy
 - operandusként előforduló operátort zárójelbe kell tenni, pl. $Comp = (>)$
 - nem létezhet azonos nevű infix és posztfix operátor.
- Sok Prolog rendszerben nem kötelező betartani ezeket a megszorításokat.

Operátorok felhasználása

- Mire jók az operátorok?
 - aritmetikai eljárások kényelmes 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
```

TÍPUSOK PROLOGBAN

Típusok leírása Prologban

- Típusleírás: (tömör) Prolog kifejezések egy halmazának megadása
- Alaptípusok leírása: `int`, `float`, `number`, `atom`, `any`
- Új típusok felépítése:
 - $\{ \text{str}(T_1, \dots, T_n) \}$ jelentése $\{ \text{str}(e_1, \dots, e_n) \mid e_1 \in T_1, \dots, e_n \in T_n, n \geq 0 \}$
 - 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.
 - `{személy(atom,atom,int)} \| {atom-atom} \| 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

\langle típusdeklaráció \rangle	<code>::=</code>	\langle típuselnevezés \rangle \langle típuskonstrukció \rangle
\langle típuselnevezés \rangle	<code>::=</code>	<code>:- type</code> \langle típusazonosító \rangle <code>==</code> \langle típusleírás \rangle .
\langle típuskonstrukció \rangle	<code>::=</code>	<code>:- type</code> \langle típusazonosító \rangle <code>---></code> \langle megkülönb. únió \rangle .
\langle megkülönb. únió \rangle	<code>::=</code>	\langle konstruktor \rangle ; ...
\langle konstruktor \rangle	<code>::=</code>	\langle névkonstans \rangle \langle struktúranév \rangle (\langle típusleírás \rangle , ...)
\langle típusleírás \rangle	<code>::=</code>	\langle típusazonosító \rangle \langle típusváltozó \rangle { \langle konstruktor \rangle } \langle típusleírás \rangle \ \langle típusleírás \rangle
\langle típusazonosító \rangle	<code>::=</code>	\langle típusnév \rangle \langle típusnév \rangle (\langle típusváltozó \rangle , ...)
\langle típusnév \rangle	<code>::=</code>	\langle névkonstans \rangle
\langle típusváltozó \rangle	<code>::=</code>	\langle változó \rangle

Predikátumtípus-deklarációk

- Predikátumtípus-deklaráció
 - `:- pred` \langle eljárásnév \rangle (\langle típusazonosító \rangle , ...)
- Példa:
 - `:- pred tree_sum(tree(int), int).`
- Predikátummód-deklaráció (Nem kötelező, több is megadható.)
 - `:- mode` \langle eljárásnév \rangle (\langle módazonosító \rangle , ...) ahol \langle módazonosító \rangle `::=` `in` | `out` | `inout`.
 - (Mercury-ban az `inout` módazonosító nem megengedett.)
- Példák:
 - `:- mode tree_sum(in, in).` % ellenőrzés
 - `:- mode tree_sum(in, out).` % fa-összeg előállítás
 - `:- mode tree_sum(out, in).` % adott összegű fa építése
- Vegyes típus- és móddeklaráció
 - `:- pred` \langle eljárásnév \rangle (\langle típusazonosító \rangle : : \langle módazonosító \rangle , ...)
- 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.
 - `tree_sum(+T, ?Sum).`
- Mód-jelölő karakterek:
 - + bemenő argumentum (behelyettesített)
 - - kimenő argumentum (behelyettesítetlen)
 - : eljárás-paraméter (meta-eljárásokban)
 - ? tetszőleges