Deklaratív programozás 8. előadás

Kabódi László[1] Marussy Kristóf[2]
BME Számítástudományi és Információelméleti Tanszék
Mesterséges Intelligencia és Rendszertervezés Tanszék

[1] kabodi.laszlo@vik.bme.hu
[2] marussy@mit.bme.hu

2026. ősz

I. rész

Programozás Prologban

1 Programozás Prologban

Lista-komprehenzió: megoldások listába gyűjtése

- A `findall(Gyűjtő, Cél, Lista)` eljárás
  - a `Cél` kifejezést eljáráshívásként értelmezi;
  - a `Cél` eljárást meghívja, és minden sikeres lefutása után a `Gyűjtő` adatstruktúra másolatát elmenti;
  - a `Cél` hívás keresési terének kimerítésekor a `Gyűjtő` változó összes elmentett másolatát (a keletkezésük sorrendjében) egy listába gyűjti és ezt egyesíti a `Lista` kimenő paraméterrel.
- Példák az eljárás használatára:

```
| ?- findall(NSz, nsz('Imre', NSz), NSzk).
    ⟹    NSzk = ['B. Gizella','C. Henrik','Sarolt','Géza'] ? ; no
| ?- findall(Sz, sz('István',Sz), Szulok).
    ⟹    Szulok = ['Sarolt','Géza'] ? ; no
| ?- findall(Gy, sz(Gy, 'István'), Gyerekek).
    ⟹    Gyerekek = ['Imre'] ? ; no
| ?- findall(Gy, sz(Gy, 'Imre'), Gyerekek).
    ⟹    Gyerekek = [] ? ; no
```

- Ha nincs megoldás, akkor (értelemszerűen) egy üres listát kapunk eredményül.

Aritmetikai beépített eljárások

- Egy aritmetikai kifejezés[1] (`AKif`) a BIP végrehajtásakor kötelezően:
  - tömör (ground) – behelyettesítetlen változót nem tartalmaz;
  - csak számokból és megengedett aritmetikai függvényekből áll
- A legfontosabb (2-arg.-ú) függvények: `+`, `-`, `*`, `/` (lebegőp. eredményt ad), `//` (egész-osztás, 0 felé kerekít), `rem` (maradék, `//` szerint)
- `X is AKif`: Az `AKif` aritmetikai kif. értékét egyesíti `X`-szel, pl.

```
| ?- X = 2, Y is X+1.                    ⟹  X = 2, Y = 3 ?; no
| ?- Y is X+1, X = 2.                    ⟹  ! Instantiation error
| ?- 3 is 2+1.                           ⟹  yes
| ?- 1+3 is 6-2.                        ⟹  no  % a bal oldalt nem értékeli ki!
| ?- X = 1, Y is (X-27) rem (X+2).       ⟹  X = 1, Y = -2 ?; no
| ?- Kif = X/(X-1), X = 6, Y is Kif.     ⟹  Kif = 6/(6-1), X = 6, Y = 1.2 ?; no
```

- További aritmetikai BIP-ek: `AKif1 < AKif2`, `AKif1 > AKif2`, `AKif1 =< AKif2` (vigyázat: nem <=), `AKif1 >= AKif2`, `AKif1 =:= AKif2` (aritmetikailag egyenlő), `AKif1 =\= AKif2` (aritmetikailag nem-egyenlő) – ezek mindkét oldalt kiértékelik, és elvégzik a kért összehasonlítást:

```
| ?- 1+3 =:= 6-2.                        ⟹  yes
| ?- 1+1 =\= 6/3.                        ⟹  no
```

[1] pl. https://sicstus.sics.se/sicstus/docs/latest/html/sicstus/ref_002dari_002daex.html

Példa: faktoriális számítása Prologban

- Funkc. nyelven a faktoriális egy 1-argumentumú függvény: `F = fakt(N)`
- Prologban ennek egy kétargumentumú reláció felel meg: `fakt(N, F)`
- Konvenció: az utolsó argumentum(ok) a kimenő pararaméter(ek)
- Idézzük föl a faktoriális függvény Elixir megvalósítását:

```
def fakt0(0)              do 1               end
def fakt0(n) when n > 0   do n * fakt0(n-1)  end
```

- Írjuk át úgy, hogy a `fakt0` hívás különüljön el az egyéb számításoktól:

```
def fakt1(0)             do                                          1 end
def fakt1(n) when n > 0  do n1 = n-1; f1 = fakt1(n1); f = f1*n; f end
```

- Az „`F = fakt1(N)`” $\Longrightarrow$ `fakt(N, F)` transzformáció adja a Prolog kódot:

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

```
| ?- fakt(5, F).  ⟹  F = 120 ? ; no
| ?- fakt(4, 24). ⟹  yes
| ?- fakt(0, F).  ⟹  F = 1 ? ; no
```

```
| ?- fakt(0, 2).  ⟹  no
| ?- fakt(N, 1).  ⟹  N = 0 ? ;
                      ! Inst. err...
```

Aritmetika plusz lista-komprehenzió

Egy korábbi Elixir példa: gyűjtsük össze azokat a pitagoraszi számhármasokat, amelyek összege egy adott `N` számnál kisebb-egyenlő

```
:- use_module(library(between)). % SWI Prologban elhagyandó

% pitag(N, Hármasok):
% Hármasok = { p(A,B,C) | 1 =< A < B < C, A+B+C =< N, A*A + B*B = C*C }
pitag(N, Pk) :-
    findall(

      p(A,B,C),
      ( between(1, N, A),
        between(1, N, B), A < B,
        between(1, N, C), A+B+C =< N,
        A*A + B*B =:= C*C
      ),
      Pk).
```

```
% def pitag(n), do:
%
% (ls = 1..n
%
%  for a <- ls,
%      b <- ls, a < b,
%      c <- ls, a + b + c <= n,
%      a * a + b * b === c * c,
%        do: {a, b, c}
% )
```

```
| ?- pitag(12, Pk).      ⟹    Pk = [p(3,4,5)] ? ; no
| ?- pitag(36, Pk).
     ⟹    Pk = [p(3,4,5),p(5,12,13),p(6,8,10),p(9,12,15)] ? ; no
```

Adatstruktúrák Prologban – a bináris fák példája

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

Binárisfa-struktúra C-ben

```
enum treetype {Node, Leaf};
struct tree {
 enum treetype type;
 union {
   struct { struct tree *left;
            struct tree *right;
          } nd;
   struct { int value;
          } lf;
 } u;
};
```

A Prolog dinamikusan típusos nyelv – nincs szükség explicit típusdefinícióra

- Mercury típusleírás (komment)

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

- A típushoz tartozás ellenőrzése

```
% is_tree(T): T egy bináris fa
is_tree(leaf(V)) :- integer(V).
is_tree(node(Left,Right)) :-
    is_tree(Left),
    is_tree(Right).
```

Bináris fák összegzése

Egy bináris fa levélösszegének kiszámítása:

- egylevelű fa esetén a levélben tárolt egész
- csomópont esetén a két részfa levélösszegének összege

C nyelven

```
% S = tsum(T): T levélösszege S
int tsum(struct tree *tree)
{
 switch(tree->type) {
 case Leaf:
  return tree->u.lf.value;
 case Node:
  return tsum(tree->u.nd.left) +
         tsum(tree->u.nd.right);
   }
}
```

Prologban

```
% tree_sum(Tree, S): Σ Tree = S.
tree_sum(leaf(Value), Value).
tree_sum(node(Left,Right), S) :-
        tree_sum(Left, S1),
        tree_sum(Right, S2),
        S is S1+S2.
| ?- tree_sum(node(leaf(5),
                   node(leaf(3),
                        leaf(2))),S).
S = 10 ? ;
no
| ?- tree_sum(T, 3).
T = leaf(3) ? ;
! Inst. error in argument 2 of is/2
! goal:  3 is _73+_74
```

Tartalom

1. Programozás Prologban
   - A Prolog nyelv alapszintaxisa
   - Listakezelő eljárások Prologban
   - Nyomkövetés: 4-kapus doboz modell

Predikátumok, klózok

- Példa:

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

- Szintaxis:

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

Prolog kifejezések

- Példa – egy klózfej mint kifejezés:

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

- Szintaxis:

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

Lexikai elemek: példák és szintaxis

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

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

⟨ névkonstans ⟩ ::= ’⟨ idézett kar. ⟩...’ |
⟨ kisbetű ⟩⟨ alfanum. jel ⟩... |
⟨ tapadó jel ⟩... | ! | ; | [] | {}

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

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

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

⟨ alfanum. jel ⟩ ::= ⟨ kisbetű ⟩ | ⟨ nagybetű ⟩ | ⟨ számjegy ⟩ | _

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

Prolog programok formázása

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

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

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

Elixir és Prolog: néhány eltérés és hasonlóság

Elixir	**Prolog**
függvény, értéke tetsz. típusú	predikátum, azaz Boole-értékű függvény
arg. bemenő, a fv.érték kimenő	arg.-ok bemenők és kimenők is
egyetlen visszatérési érték	választási pontok, több megoldás lehet
külön ennes, lista típusok	a lista is összetett kifejezés
nincsenek felh. operátorok	felhasználói operátorok definiálhatók
Az = jobb oldalán tömör kif., bal oldalon mintakif.; őrfeltételekkel	az egyesítés szimmetrikus, mindkét oldalon minták

- Néhány hasonlóság:
  - az eljárás is klózokból áll, kiválasztás mintaillesztéssel, sorrendben, de míg Elixirben csak az első illeszkedő klózfej számít, Prologban az összes
  - változóhoz csak egyszer köthető érték
  - lista szintaxisa (de: Elixirben önálló típus), sztring (füzér), atom

Tartalom

1. Programozás Prologban
   - A Prolog nyelv alapszintaxisa
   - Listakezelő eljárások Prologban
   - Nyomkövetés: 4-kapus doboz modell

Listák összefűzése – az `append/3` eljárás

- Elixir megoldás:

```
def append([],    b) do                    b     end
def append([x|a], b) do c = append(a,b); [x|c] end
```

- Írjuk át a kétargumentumú `append` függvényt egy `app0/3` Prolog eljárássá!

```
% app0(A, B, C): A és B listák összefűzése a C lista, C = A ⊕ B
app0([], B, Ret) :- Ret = B.
app0([X|A], B, Ret) :-
    app0(A, B, C), Ret = [X|C].
```

- Logikailag tiszta Prolog programokban a `Vált = Kif` alakú hívások elhagyhatók, ha `Vált` többi előfordulását `Kif`-re cseréljük.

```
app([], B, B).
app([X|A], B, [X|C]) :-
    app(A, B, C).
```

- Mindkét eljárásban a (max) futási idő arányos az 1. arg. hosszával
- Miért jobb az `app/3` mint az `app0/3`?
  - `app/3` jobbrekurzív, ciklussal ekvivalens (nem fogyaszt vermet)
  - `app([1,...,1000],[0],[2,...])` 1, `app0(...)` 1000 lépésben hiúsul meg.
  - `app/3` használható szétszedésre is (lásd később), míg `app0/3` nem.

Lista építése *elölről* – nyílt végű listákkal

- **Ismétlés:** egy `X` Prolog kifejezés nyílt végű lista, ha `X` változó, vagy `X = [_|Farok]` ahol `Farok` nyílt végű lista.

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

- A beépített `append/3` azonos az `app/3`-mal:

```
append([], B, B).
append([X|A], B, [X|C]) :-
    append(A, B, C).
```

- Az `append` eljárás már az első redukciónál felépíti az eredmény fejét!
  - Példa-célsorozat: `append([1,2], [3,4,5], Ered), answer(Ered).`
  - Fej: `append([X|A], B, [X|C])`
  - Behelyettesítés: `X = 1, A = [2], B = [3,4,5], Ered = [1|C]`
  - Új célsorozat: `append([2], [3,4,5], C), answer([1|C]).`

  (`Ered` nyílt végű lista, farka még behelyettesítetlen.)

Lista építése *elölről* – a megvalósítás részletei

- A kimenő paraméter behelyettesítését explicitté tehetjük:

```
app1([], B, L) :-                                   % (a1)
    L = B.
app1([X|A], B, L) :-                                % (a2)
    L = [X|C],
    app1(A, B, C).
```

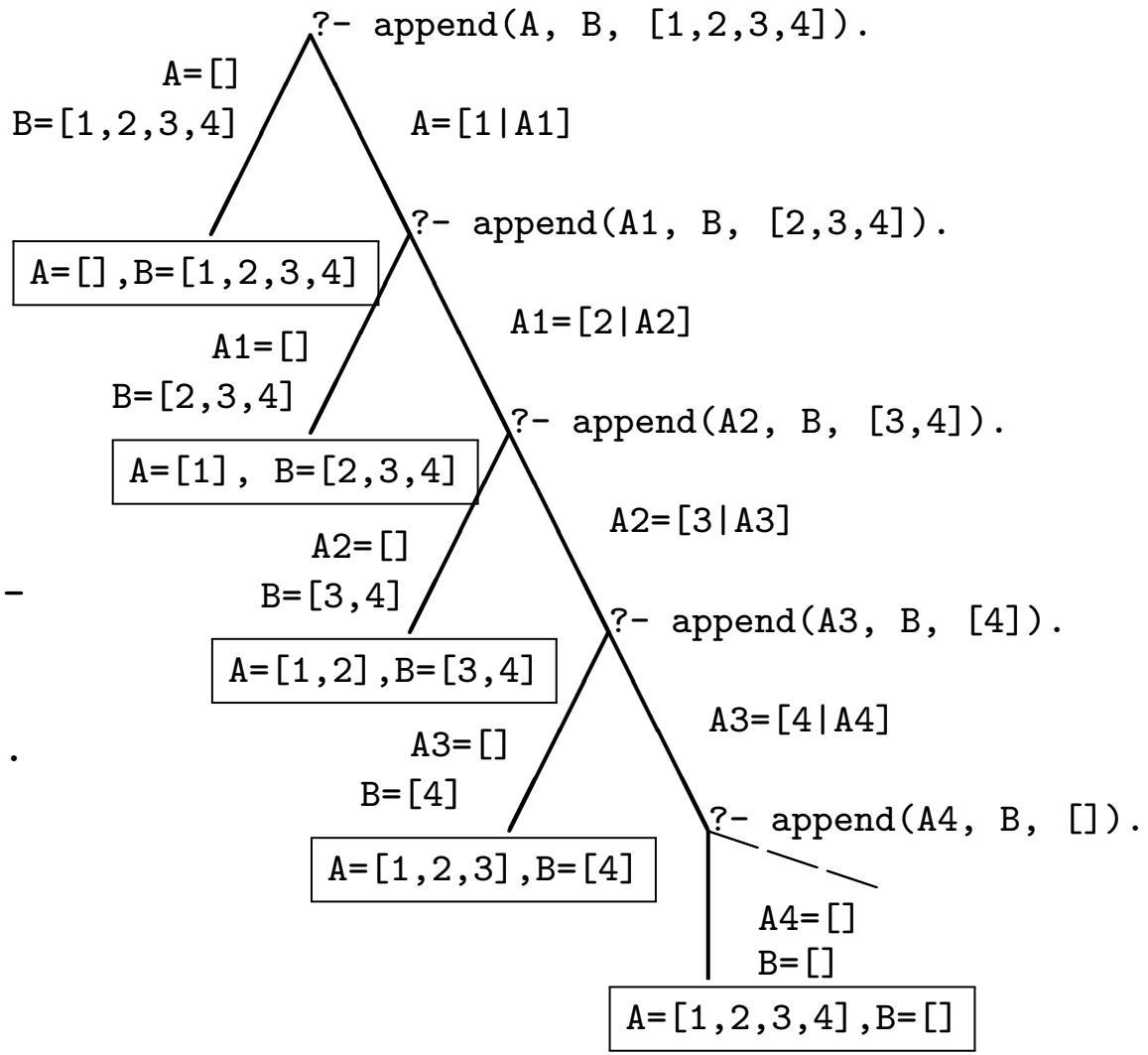
- Egy `app1/3` eljáráshívás redukciós lépései:

```
:- app1([1,2], [3,4,5], Ered), answer(Ered).                    % (cs0)
                                                  % + (a2) =>
:- Ered = [1|C1], app1([2], [3,4,5], C1), answer(Ered).         % (cs1)
                                                  % + BIP  =>
:- app1([2], [3,4,5], C1), answer([1|C1]).                      % (cs2)
                                                  % + (a2) =>
:- C1 = [2|C2], app1([], [3,4,5], C2), answer([1|C1]).          % (cs3)
                                                  % + BIP  =>
:- app1([], [3,4,5], C2), answer([1,2|C2]).                     % (cs4)
                                                  % + (a1) =>
:- C2 = [3,4,5], answer([1,2|C2]).                              % (cs5)
                                                  % + BIP  =>
:- answer([1,2,3,4,5]).                                         % (cs6)
```

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
```



Nyílt végű listák az append változatokban

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

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

- Ha az 1. argumentum zárt végű (n hosszú), mindkét változat legfeljebb $n+1$ lépésben egyértelmű választ ad, amely lehet nyílt végű:
  `| ?- app0([1,2], L2, L3).` $\Longrightarrow$ `L3 = [1,2|L2] ? ; no`
- A 2. arg.-ot nem bontjuk szét $\Longrightarrow$ mindegy, hogy nyílt vagy zárt végű
- Ha a 3. argumentum zárt végű (n hosszú), akkor az `append` változat legfeljebb $n+1$ megoldást ad, max. $\sim 2n$ lépésben (ld. előző dia); tehát:
  - `append(L1, L2, L3)` keresési tere véges, ha `L1` vagy `L3` zárt
- Ha az 1. és a 3. arg. is nyílt, akkor a válaszhalmaz csak végtelen sok Prolog kifejezéssel fedhető le, pl.
  `_` $\oplus$ `[1]` $=$ `L` ($\equiv$`L` utolsó eleme `1`): `L = [1]; [_,1]; [_,_,1]; ...`
- `app0` szétszedésre nem jó, mert pl. `app0(L, [1], [])` $\Longrightarrow \infty$ ciklus, hiszen redukálva a 2. klózzal $\Longrightarrow$ `app0(L1, [1], L3), [] = [X|L3].`
- Az `append` eljárás jobbrekurzív, hála a logikai változó használatának

Variációk `append`-re – három lista összefűzése (kieg. anyag)

- `append(L1,L2,L3,L123)`: $(\texttt{L1} \oplus \texttt{L2}) \oplus \texttt{L3} = \texttt{L123}$

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

- Lassú, 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).
```

  - `append(+,+,?,?)` esetén az első `append/3` hívás nyílt végű listát ad:

    `| ?- append([1,2], L23, L).` $\Longrightarrow$ `L = [1,2|L23] ?`
  - Az `L3` argumentum behelyettesítettsége (nyílt vagy zárt végű lista-e) nem számít.

Listák megfordítása

- Naív (négyzetes lépésszámú) megoldás

```
% nrev(L, R): R = L megfordítása.
nrev([], Ret) :- Ret = [].          % def nrev([])    do []  end
nrev([X|L], Ret) :-                 % def nrev([x|l]) do
    nrev(L, RL),                    %         rl = nrev(l);
    append(RL, [X], Ret).           %         append(rl,[x]) end
```

- Lineáris lépésszámú megoldás

```
% revapp(L0, R0, R): L0 megfordítását R0 elé fűzve kapjuk R-t.
revapp([], R0, R) :- R = R0.        % def revapp([], r0)     do r0 end
revapp([X|L0], R0, R) :-            % def revapp([x|l0], r0) do
    revapp(L0, [X|R0], R).          %         revapp(l0, [x|r0])   end

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

- `revapp`-ban *R0*, *R* egy akkumulátorpár: eddigi ill. végeredmény
- A `lists` könyvtár tartalmazza a `reverse/2` eljárás definícióját, betöltése:

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

Listák gyűjtése elölről és hátulról (kieg. anyag)

- Prolog

```
revapp([], L, L).
revapp([X|L0], L2, L3) :-
    revapp(L0, [X|L2], L3).
```

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

- C++

```
struct lnk  { char elem;
              lnk *next;
              lnk(char e): elem(e) {}   };

typedef lnk *list;
```

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

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

Keresés listában – a `member/2` beépített eljárás

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

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

- Eldöntendő (igen-nem) kérdés:

```
| ?- member(2, [1,2,3,2]).          ⟹   yes                          DE
| ?- member(2, [1,2,3,2]), R=yes.   ⟹   R=yes ? ; R=yes ? ; no
```

- Lista elemeinek felsorolása:

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

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

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

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

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

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

A `member/2` predikátum általánosítása: `select/3`

- `select(E, Lista, M)`: `E` elemet `Lista`ból pont egyszer elhagyva marad `M`.

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

- Felhasználási lehetőségek:

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

- A `select/3` eljárás keresési tere véges, ha vagy a 2., vagy a 3. argumentuma zárt végű lista (a `lists` könyvtár tartalmazza az eljárást)

Listák permutációja (kieg. anyag)

- `% perm0(+Lista, ?Perm)`: `Lista` permutációja a `Perm` lista.

```
perm0([], []).
perm0([Elso|Lista], Perm) :-
        perm0(Lista, Perm0),          % permutáljuk a bemenet farkát
        select(Elso, Perm, Perm0).    % ebbe beszúrjuk a bemenet fejét
```

- Felhasználási példák:

```
| ?- perm0([1,2], L).
    ⟹     L = [1,2] ? ; L = [2,1] ? ; no

| ?- perm0([a,b,c], L).
    ⟹     L = [a,b,c] ? ; L = [b,a,c] ? ; L = [b,c,a] ? ;
           L = [a,c,b] ? ; L = [c,a,b] ? ; L = [c,b,a] ? ; no

| ?- perm0(L, [1,2]).
    ⟹      L = [1,2] ? ;  végtelen keresési tér
```

- Ha `perm0/2`-ben az első argumentum változó, akkor a rekurzív hívás mindkét argumentuma változó lesz ⇒végtelen sok választás
- A `lists` könyvtárban van egy kétirányban működő `permutation/2` eljárás

Tartalom

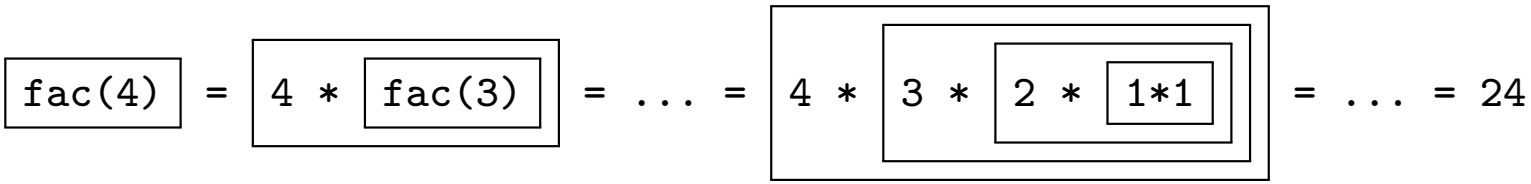
1. Programozás Prologban
   - A Prolog nyelv alapszintaxisa
   - Listakezelő eljárások Prologban
   - Nyomkövetés: 4-kapus doboz modell

Függvények és eljárások egymásba skatulyázása

- A deklaratív nyelvekben a rekurzió váltja ki a ciklust, így gyakran előfordulnak egymásba **skatulyázott** függvény- ill. eljáráshívások.
- Tekintsük a faktoriális Elixir definícióját!

```
% fac(N) = N faktoriálisa.
def fac(0), do: 1
def fac(n), do: n * fac(n-1)
```

- A `fac(4)` függvényhívás végrehajtásakor pl. az alábbi állapotokat kapjuk:



- A függvényhívásokba való be- és kilépés nyomon követése:

```
Call fac(4)
Call fac(3)
...
Call fac(0)
Exit fac(0) = 1
...
Exit fac(3) = 6
Exit fac(4) = 24
```

A `Call` nyomkövetési információ egy fenti doboz *létrehozásához* kapcsolható, míg az `Exit` a doboz kiértékelésének *befejezéséhez*.

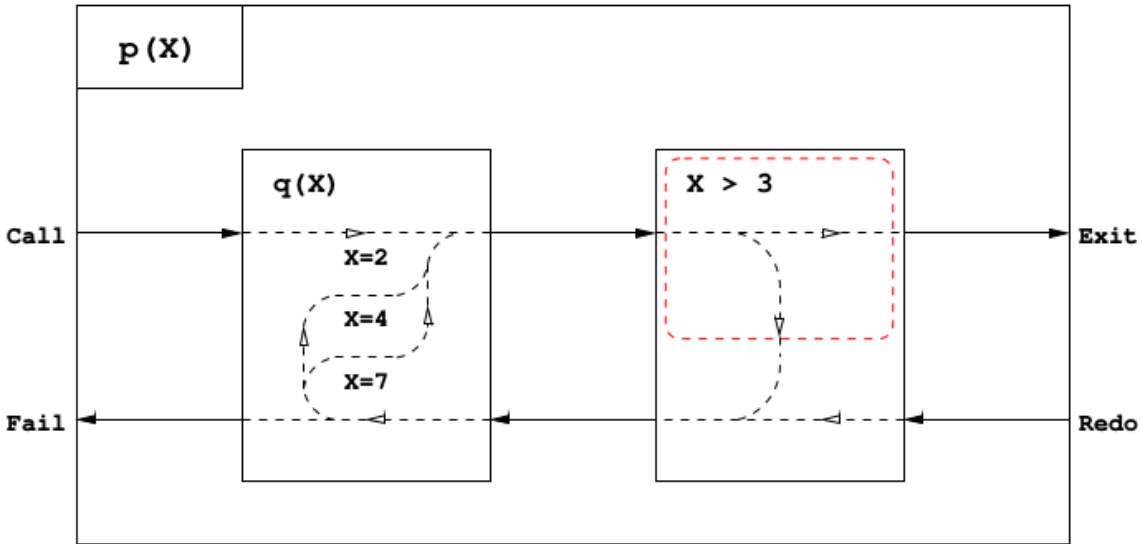
Prolog nyomkövetés eljárás-doboz modellel

- A Prolog doboz alapú nyomkövetésében is az eljárások be- és kilépési pontjain (ún. kapukon, angolul *port*) való áthaladásról kapunk információt:
  - **Call port** (hívás kapu) – belépés az eljárásba, doboz létrehozása
  - **Exit port** (kilépés kapu) – sikeres lefutás, esetleg doboz törlése
  - **Fail port** (meghiúsulás kapu) – sikertelen lefutás, doboz törlése
  - **Redo port** (újra kapu) – új megoldás kérése
- A Prolog eljárás-végrehajtás két fázisa
  - előre menő: egymásba **skatulyázott eljárás-be** és **-ki**lépések
  - visszafelé menő: **új megoldás** kérése egy már lefutott eljárástól
- Prolog végrehajtás objektum-orientált szemléletben (eljárás $\Rightarrow$ objektum):
  - eljárás meghívása (hívás kapu): objektum létrehozása
  - sikeres lefutás (kilépés kapu): változóbehelyettesítések visszaadása
  - sikertelen lefutás (meghiúsulás kapu): meghiúsulás jelzése
  - új megoldás kérése (újra kapu): további választási pont(ok) bejárása

Eljárás-doboz modell – grafikus szemléltetés

Egy egyszerű példaprogram, hívása `| ?- p(X).`

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



Előre: `Call p(X); Call q(X); Exit q(2); Call 2>3; Fail 2>3`

Vissza: `Redo q(2);`

Előre: `Exit q(4); Call 4>3; Exit 4>3; Exit p(4);` siker `X = 4 ? ;`

Vissza: `Redo p(4); Redo 4>3; Fail 4>3; Redo q(4);`

Előre: `Exit q(7); Call 7>3; Exit 7>3; Exit p(7);` siker `X = 7 ? ;`

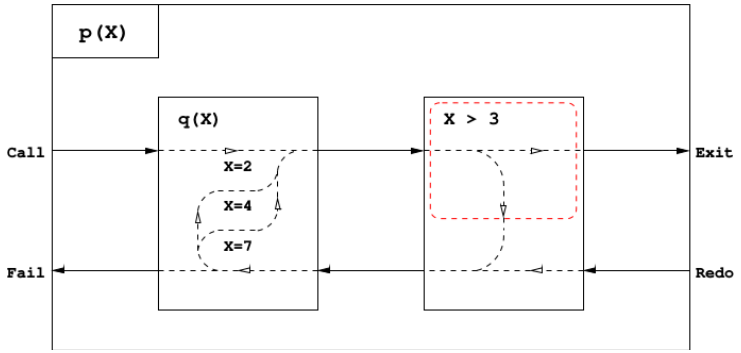
Vissza: `Redo p(7); Redo 7>3; Fail 7>3; ...; Fail p(X);` meghiúsulás `no`

Egy egyszerű nyomkövetési példa (SICStus Prolog)

- SICStusban `?...Exit` jelzi, hogy van választási pont a lefutott eljárásban
- Ha nincs `?` az Exit kapunál, akkor a doboz törlődik (lásd a szaggatott piros téglalapot az `X > 3` hívás körül)

```
q(2).
q(4).
q(7).

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

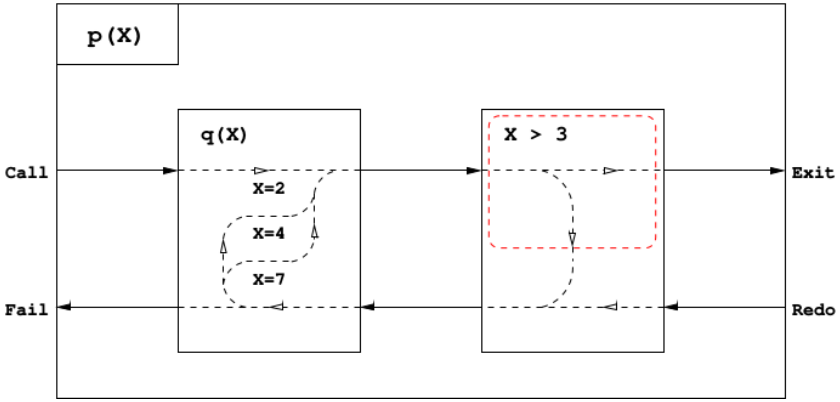



```
% Sorszám      Mélység
| ?- consult(pq0), trace, p(X).     % compile esetén a > /2 hívásokat nem látjuk
        1      1 Call: p(_463) ?
        2      2 Call: q(_463) ?
?       2      2 Exit: q(2) ?      % ? ≡ maradt választási pont q-ban
        3      2 Call: 2>3 ?
        3      2 Fail: 2>3 ?
        2      2 Redo: q(2) ?
?       2      2 Exit: q(4) ?
        4      2 Call: 4>3 ?
        4      2 Exit: 4>3 ?       % nincs ? ⇒ a doboz törlődik, (ld. a szaggatott piros téglalapot)
?       1      1 Exit: p(4) ?
X = 4 ? ;
```

Egy egyszerű nyomkövetési példa (folyt.)

```
q(2).
q(4).
q(7).

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

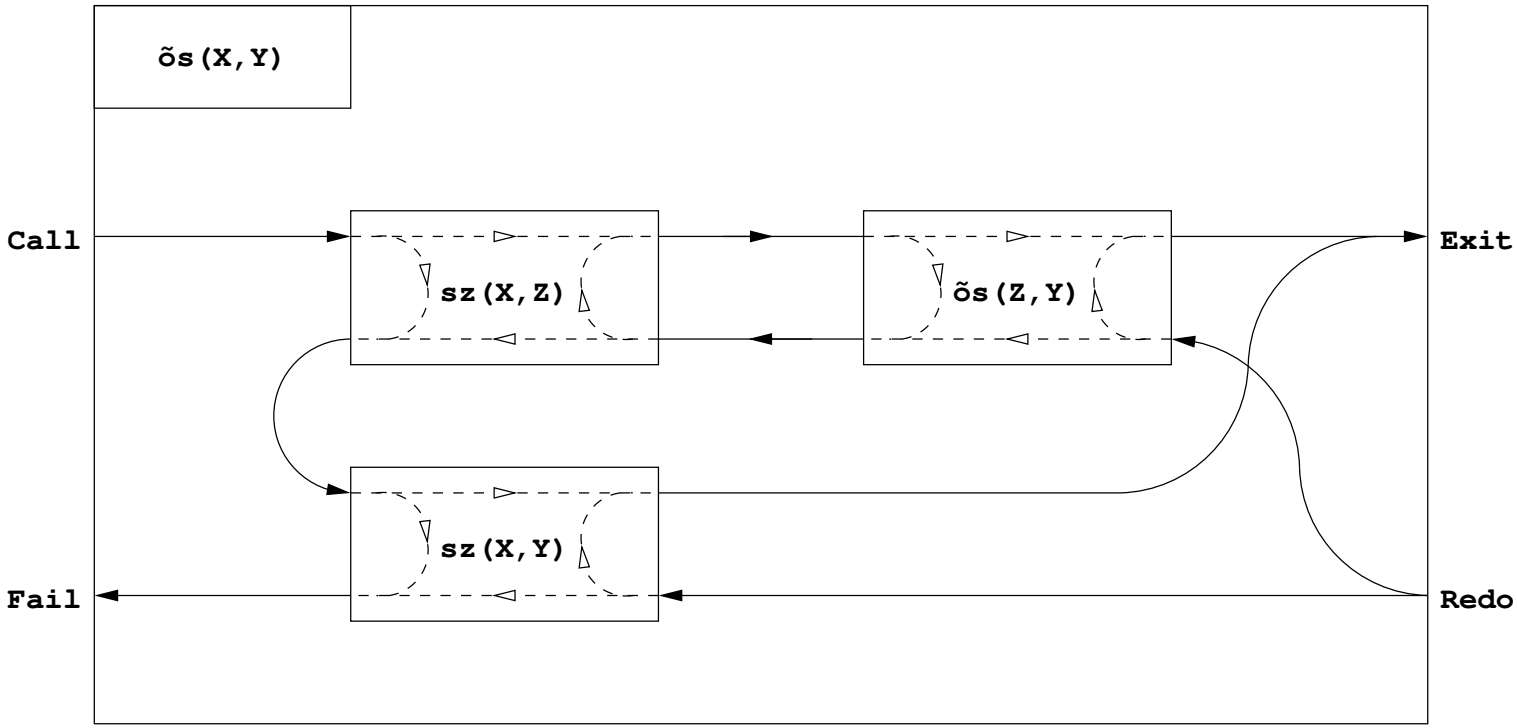



```
| ?- consult(pq0), trace, p(X).
(...)
        4      2 Exit: 4>3 ?        % nincs ? ⇒ a doboz törlődik (*)
?       1      1 Exit: p(4) ?
X = 4 ? ;
        1      1 Redo: p(4) ?
                                    % (*) miatt nem látjuk a Redo-Fail kapukat a 4>3 hívásra
        2      2 Redo: q(4) ?
        2      2 Exit: q(7) ?       % nincs ? ⇒ a doboz törlődik
        5      2 Call: 7>3 ?
        5      2 Exit: 7>3 ?        % nincs ? ⇒ a doboz törlődik
        1      1 Exit: p(7) ?       % nincs ? ⇒ a doboz törlődik
X = 7 ? ;
no
```

Eljárás-doboz: több klózból álló eljárás

```
ős(X,Y) :- sz(X,Z), ős(Z,Y). % X őse Y ha X szülője Z és Z őse Y (a szülő őse ős)
ős(X,Y) :- sz(X,Y).          % X őse Y ha X szülője Y           (a szülő ős)

sz(a,b).   sz(b,c).   sz(b,d).
```



Eljárás-doboz modell – „kapcsolási” alapelvek

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

Nyomkövetés – legfontosabb parancsok (SICStus + SWI)

- Beépített eljárások
  - `trace`, `debug` – a `c`, `l` paranccsal indítja a nyomkövetést
  - `notrace`, `nodebug` – kikapcsolja a nyomkövetést
  - `spy(P)`, `nospy(P)`, `nospyall` – töréspont be/ki a `P` eljárásra, $\forall$ ki.
- Alapvető nyomkövetési parancsok (SICStus: `<RET>`-tel kell lezárni)
  - `h` (help) – parancsok listázása
  - `c` (creep) vagy csak `<RET>` – lassú futás (minden kapunál megáll)
  - `l` (leap) – csak töréspontnál áll meg
  - `+` ill. `-` – töréspont be/ki a kurrens eljárásra
  - `s` (skip) – eljárástörzs átlépése (Call/Redo $\Rightarrow$ Exit/Fail)
  - `w` (write) – teljes mélységű kiíratás
  - `o` (out) SICStus, `u` (up) SWI – kilépés az eljárástörzsből
  - `r` (retry) – újrakezdi a kurrens hívás végrehajtását
- Információ-megjelenítő és egyéb parancsok
  - `g` (goals) – a kurrens hívást tartalmazó célok kiíratása
  - `b` (break) – új, beágyazott Prolog interakciós szint létrehozása
  - `n` (notrace) – nyomkövető kikapcsolása
  - `a` (abort) – a kurrens futás abbahagyása