SML-Prolog átvezetés: párhuzamok a két nyelv között

SML

```
fun append ([], ys) = ys
  | append (x::xs, ys) =
                x::append (xs, ys)
```

Prolog

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

SML „Prologosítva”

```
fun append([], L) = L
  | append(X::L1, L2) =
    let val L3 = append(L1, L2)
    in  X::L3 end
```

Prolog „SML-esítve”

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

SML	Prolog
függvény	predikátum
klóz	klóz (lazább a kapcsolat a pred.-mal)
változó: egyetlen, ismert érték	változó: egy, esetleg ismeretlen érték
egyirányú mintaillesztés	kétirányú mintaillesztés
egyértelmű klózválasztás	többértelmű klózválasztás
egy eredmény	több eredmény (nemdeterminizmus)
egyirányú használat	többirányú használat (pl. összerakó és szétszedő `append`)
adatkonstruktor-függvény	struktúra (rekord)
egymásba ágyazott függvényhívások	konjunkció, segéd-változóval

SML-Prolog átvezetés: további példák

SML

```
fun append xs ys = foldr op:: ys xs
```

Prolog

```
/* Prologban kevésbé használtak
   a magasabbrendű eljárások */
```

SML

```
fun fakt 0 = 1
  | fakt n = n * fakt (n-1)
```

Prolog

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

SML	Prolog
típusos nyelv	típustalan nyelv
magasabbrendű függvény	rekurzió, ritkábban magasabbr. pred.
rekurzió	visszalépéses ciklus (pl. két lista közös eleme)
kivétel (pl. `fakt` negatív számmal)	meghiúsulás, kivétel

Összefoglalás: Prolog programok szemantikája

- Prolog program jelentése = milyen válaszokat (behelyettesítéseket) kapunk egy cél futtatásakor:
  - Procedurális szemantika — az ismertetett végrehajtási, egyesítési algoritmus.
  - Deklaratív szemantika:
    - program: logikai állítások (klózok, azaz implikációk) halmaza;
    - egy cél futási eredménye: olyan behelyettesítés, amelyre a cél *következménye* a programnak.
- A Prolog procedurális szemantika csak olyan választ ad, amely a deklaratív szemantika szerint is helyes! (Ha predikátumaink „igazak”, akkor rossz eredményt nem kaphatunk, csak végtelen ciklust. `:-(`)

Ismétlés: A Prolog végrehajtási mechanizmus, dióhéjban

- (Kezdet:) Ha célsorozat üres → sikeres lefutás.
- (Folytatás:) Keresünk az *első* céllal egyesíthető klózfejet (a klózból friss másolatot képezve, felülről lefelé haladva a programbeli klózokon).
- Ha van ilyen:
  - Ha van esély további illesztésre, akkor választási pontot hozunk létre: a futás jelenlegi állapotát (célsorozat + hányadik klózzal illesztettünk) megjegyezzük, azaz a veremre rakjuk.
  - Az egyesítéshez szükséges behelyettesítéseket a klóztörzsön és a célsorozaton is elvégezzük.
  - Az első cél helyébe a klóztörzset rakjuk, ez lesz az új célsorozat, majd vissza a (Kezdet)-hez.
- Ha nincs illeszthető klózfej, akkor visszalépünk a *legutolsó* választási pontnak megfelelő állapotba (azt leemelve a verem tetejéről), és új egyesíthető fejű klóz keresésével folytatjuk a (Folytatás)-nál.

Ismétlés: A Prolog egyesítési algoritmus, dióhéjban

- Legáltalánosabb egyesítő behelyettesítés meghatározása
  - Azonos változók ill. konstansok behelyettesítés nélkül egyesíthetőek.
  - Változó minden más kifejezéssel egyesíthető, triviális behelyettesítéssel (tartalmazás-vizsgálat nélkül)
  - Két összetett kifejezés egyesíthető, ha funktoraik azonosak, és az argumentumaik sorra egyesíthetőek, úgy, hogy a megelőző argumentumok egyesítéséhez szükséges behelyettesítéseket már elvégeztük. Az argumentumok egyesítését biztosító behelyettesítések kompozíciója a legáltalánosabb egyesítő.
  - Minden más esetben a két kifejezés nem egyesíthető, az egyesítési algoritmus meghiúsul.

4. fejezet: Prolog programozási módszerek

- Az előző előadás-blokk (jegyzetbeli 3. fejezet) célja volt:
  - a Prolog nyelv alapjainak bemutatása,
  - a logikailag „tiszta” résznyelvre koncentrálva.
- A jelen előadás-blokk (jegyzetben a 4. fejezet) célja: olyan
  - beépített eljárások,
  - programozási technikák

  bemutatása, amelyekkel
  - hatékony Prolog programok készíthetők,
  - esetleg a tiszta logikán túlmutató eszközök alkalmazásával.

Prolog programozási módszerek: tartalomjegyzék

- A keresési tér szűkítése
- Vezérlési eljárások
- Determinizmus és indexelés
- Jobbrekurzió és akkumulátorok
- Algoritmusok Prologban
- Megoldások gyűjtése és felsorolása
- Meta-logikai eljárások
- Magasabbrendű eljárások
- Modularitás
- Dinamikus adatbáziskezelés
- Nyelvtani elemzés

A KERESÉSI TÉR SZŰKÍTÉSE

Prolog nyelvi eszközök a keresési tér szűkítésére

Eszközök

- a vágó beépített eljárás: `!`
- feltételes diszjunktív szerkezet:
  `( felt -> akkor ; egyébként )`

Miért vágunk le ágakat a keresési térben?

- mert mi tudjuk, hogy ott nincs megoldás, de a Prolog megvalósítás nem — zöld vágás, szemantikailag „ártalmatlan”
  - (Például, a legtöbb Prolog megvalósítás „nem tudja”, hogy a $X = 0$ és $X > 0$ feltételek kizárják egymást, lásd később: indexelés.)
- ténylegesen eldobunk megoldásokat — vörös vágás, a program jelentését megváltoztatja
  - (Vörös vágás legtöbbször úgy keletkezik, hogy egy zöld vágót tartalmazó programban a „felesleges” feltételeket elhagyjuk.)

Példák a vágó eljárás használatára

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

% last(+L, ?E): az L lista utolsó eleme E. (A lists könyvtárban van!)
last([E], E) :- !.                                      % zöld vágó
last([_|L], E) :- last(L, E).

% pirosak(+L, -Pirosak): Pirosak az L piros/1 funktorú elemeiből áll.
pirosak([], []).
pirosak([E|Ek], [E|Pk]) :-
        E = piros(_), !,                                % vörös vágó
        pirosak(Ek, Pk).
pirosak([_E|Ek], Pk) :-
        /* \+ _E = piros(_), */ pirosak(Ek, Pk).
```

Figyelem: a fenti példák nem tökéletesek, hatékonyabb ill. általánosabb változatukat később ismertetjük!

A vágó definíciója

Segédfogalom

- Egy cél ***szülője*** az a cél, amelyik az őt tartalmazó klóz fejével illesztődött.
- Pl. a `last([E], E) :- !.` klózbeli vágó szülője lehet a `last([7], X)` hívás.
- A `g` (ancestors) nyomkövetési parancs kiírja a kurrens cél őseit (szülőjét, annak szülőjét, stb.)

A vágó végrehajtása:

- mindig sikerül; és a végrehajtás adott állapotától visszafelé egészen a szülő célig, azt is beleértve, minden választási pontot megszüntet.

A vágás kétféle választási pontot szüntet meg:

```
r(X):- s(X), !.  % az s(X)-beli választási pontokat --- a vágót megelőző
                 % cél(ok)nak az első megoldására szorítkozunk
                 % az r(X) többi klózának választását --- a vágót tartalmazó
r(X):- t(X).     % klóz mellett kötelezzük el magunkat (commit)
```

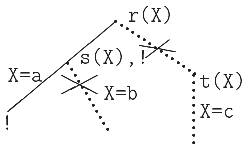

A vágó által megszüntetett választási pontok

```
% vágó nélküli példa
q(X):- s(X).
q(X):- t(X).

% ugyanaz a példa vágóval
r(X):- s(X), !.
r(X):- t(X).

s(a).     s(b).      t(c).

% a vágó nélküli példa futása
:- q(X), write(X), fail.
                          --->        abc
% a vágót tartalmazó példa futása
:- r(X), write(X), fail.
                          --->        a
```



A diszjunktív feltételes szerkezet definíciója

- A diszjunktív feltételes szerkezet, a diszjunkcióhoz hasonlóan egy segédeljárással váltható ki:

```
p :-                              p :-
    ...                               ...
    (   felt1 -> akkor1               segéd(...)
    ;   felt2 -> akkor2               ... .
    ;   ...                 ==>
    ;   egyébként                 segéd(...) :- felt1, !, akkor1.
    )                             segéd(...) :- felt2, !, akkor2.
    ... .                         ...
                                  segéd(...) :- egyébként.
```

- Az `egyébként` alternatíva elmaradhat, ilyenkor a megfelelő klóz is elmarad.
- SICStus módban a `felt` részekben vágó nem lehet, ISO módban lehet, de hatásköre (szülője) a `felt` rész.
- Az `akkor` részekben lehet vágó. Ennek hatásköre, a `->` nyílból generált vágóval ellentétben, a teljes `p` predikátum (távolbaható vágó).

Példák a diszjunktív feltételes szerkezet használatára

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

% last(+L, ?E): az L nem üres lista utolsó eleme E.
last([E|L], Last) :-
        (   L = [] -> Last = E
        ;   last(L, Last)
        ).

% pirosak(+L, ?Pirosak): Pirosak az L piros/1 funktorú elemeiből áll.
pirosak([], []).
pirosak([E|Ek], Pk) :-
        (   E = piros(_) -> Pk = [E|Pk0]
        ;   Pk = Pk0
        ), pirosak(Ek, Pk0).
```


A vágás első alapesete — klóz mellett való elkötelezés

- A klóz melletti elkötelezés általában egyszerű feltételes szerkezetet jelent.

```
szülő :- feltétel, !, akkor.
szülő :- egyébként.
```

- A vágó szükségtelenné teszi a `feltétel` negációjának végrehajtását a többi klózban. A logikailag tiszta, de nem hatékony alak:

```
szülő :- feltétel, akkor.
szülő :- \+ feltétel, egyébként.
```

  A fenti két alak csak akkor ekvivalens, ha `feltétel` egyszerű, nincs benne választás.
- Analógia: ha `a`, `b` és `c` logikai változók (pl. Pascalban), akkor `if a then b else c` $\equiv$ `a` $\wedge$ `b` $\vee \neg$ `a` $\wedge$ `c`
- A vágó által kiváltott negált feltételt célszerű kommentként jelezni:

```
szülő :- feltétel, !, akkor.
szülő :- /* \+ feltétel, */ egyébként.
```


Feltételes szerkezetek

Feltételes szerkezet — példa

```
% abs(X, A): A az X abszolút értéke.
abs(X, A)    :- X < 0, !, A is -X.
abs(X, X) /* :- X >= 0 */.
```

Általános alak

```
p :- felt1, !, akkor1.
p :- felt2, !, akkor2.
...
p :- egyébként.
```

Diszjunktív feltételes szerkezet

```
abs(X, Y) :-
        (   X < 0 -> Y is -X
        ;   Y = X
        ).
```

Általános alak

```
p :-
        (   felt1 -> akkor1
        ;   felt2 -> akkor2
        ;   ...
        ;   egyébként
        ).
```

Feltételes szerkezetek és fejillesztés

- Vigyázat: a tényleges feltétel részét képezik a fejbeli egyesítések!

```
% a vágót fej-egyesítés előzi meg      % az egyesítés explicitté téve:
abs(X, X) :- X >= 0, !.                abs(X, A) :- A = X, X >= 0, !.
abs(X, A) :- A is -X.                  abs(X, A) :- A is -X.
```

- A fej-egyesítés gondot okozhat, ha az eljárást ellenőrzésre használjuk:

```
| ?- abs(10, -10).  --->  yes
```

- A megoldás a ***vágás alapszabálya***:
  - A kimenő paraméterek értékadását mindig a vágó után végezzük!

    ```
    abs(X, A) :- X >= 0, !, A = X.
    abs(X, A) :- A is -X.
    ```

  - Ez nemcsak általánosabban használható, hanem hatékonyabb kódot is ad!
  - (*„kimenő” paraméterek* — vágó alkalmazásakor általában nincs többirányú használat :-)

A bevezető példáknak a vágás alapszabályát betartó változata

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

% last(+L, ?E): az L nem üres lista utolsó eleme E.
last([E], Last) :- !, Last = E.
last([_|L], E)  :- last(L, E).

% pirosak(+L, ?Pirosak): Pirosak az L piros/1 funktorú elemeiből áll.
pirosak([], []).
pirosak([E|Ek], Pk) :-
        E = piros(_), !, Pk = [E|Pk0], pirosak(Ek, Pk0).
pirosak([_E|Ek], Pk) :-
        /* \+ _E = piros(_), */ pirosak(Ek, Pk).
```

Megjegyzés: a diszjunktív alakban a feltételek eleve explicitek, nincs fejillesztési probléma.

Példasor: `max(X, Y, Z)`: `X` és `Y` maximuma `Z`.

- 1. változat, tiszta Prolog. Lassú, választási pontot hagy.

```
max(X, Y, X) :- X >= Y.
max(X, Y, Y) :- Y > X.
```

- 2. változat, zöld vágóval. Lassú, nem hagy választási pontot.

```
max(X, Y, X) :- X >= Y, !.
max(X, Y, Y) :- Y > X.
```

- 3. változat, vörös vágóval. Gyors, nem hagy választási pontot, de nem használható ellenőrzésre, pl. `| ?- max(10, 1, 1)` sikerül.

```
max(X, Y, X) :- X >= Y, !.
max(X, Y, Y).
```

- 4. változat, vörös vágóval. Helyes, gyors és nem hagy választási pontot.

```
max(X, Y, Z) :- X >= Y, !, Z = X.
max(X, Y, Y) /* :- Y > X */.
```


A vágás második alapesete — első megoldásra való megszorítás

Mikor használjuk az első megoldásra megszorító vágót?

- behelyettesítést nem okozó, eldöntendő kérdés esetén;
- feladatspecifikus optimalizálásra;
- végtelen választási pontot létrehozó eljárások hasznosítására.

Eldöntendő kérdés: eljáráshívás csupa bemenő paraméterrel

```
% van_elég_hosszú_út(+N, +A, +B, +Min):
% A és B között van N lépéses út, amelynek összhossza legalább Min km.
van_elég_hosszú_út(N, A, B, Min) :-
        útvonal(N, A, B, Hossz), Hossz >= Min, !.
```

Eldöntendő kérdés esetén általában nincs értelme többszörös választ adni/várni.

Feladatspecifikus optimalizálás

- Példafeladat: egy nem-negatív számokból álló lista első pozitív eleme
- Első megoldás, rekurzióval (mérnöki :-)

```
első_poz_elem([0|L], EP)     :-  !, első_poz_elem(L, EP).
első_poz_elem([EP|_], EP) /* :- EP > 0 */.
```

- Második megoldás, visszalépéses kereséssel (matematikusi :-)

```
első_poz_elem(L, EP) :-
        append(Nk, [EP|_], L), \+ poz_elem(Nk, _).

poz_elem(L, P)  :- member(P, L), P > 0.
```

- Harmadik megoldás, vágóval (Prolog hacker megoldása :-)

```
első_poz_elem(L, EP) :- member(EP, L), EP > 0, !.
```

- Figyelem: a harmadik megoldás épít a `member/2` felsorolási sorrendjére!

Feladatspecifikus optimalizálás — 2. példa

A feladat: megkeresendő egy lista elején álló „plató” hossza (platónak hívjuk a csupa azonos elemből álló folytonos részlistát).

```
% Az L lista első eleme H-szor ismétlődik a lista kezdőszeleteként.
kezdethossz(L, H) :-
        L = [E|_], append(Ek, Farok, L),
        \+ Farok = [E|_], !,            % vörös vágó
        /* egyformák(Ek, E), */
        length(Ek, H).
/*
% egyformák(Ek, E): Az Ek lista minden eleme E.
egyformák([], _).
egyformák([E|Ek], E) :-
        egyformák(Ek, E).
*/
| ?- kezdethossz([1,1,1,2,3,5], H).
H = 3 ? ; no
```


Végtelen választás megszelidítése: `memberchk`

```
% memberchk(X, L): "X eleme az L listának" kérdés első megoldása.

% 1. változat                  % 2. ekvivalens változat
memberchk(X, L):-              memberchk(X, [X|_]) :- !.
        member(X, L), !.       memberchk(X, [_|L]) :-
                                       memberchk(X, L).
```

`memberchk/2` használata

- Eldöntő kérdésben (visszalépéskor nem keresi végig a lista maradékát.)
  `memberchk(1, [1,2,3,4,5,6,7,8,9])`
- Nyílt végű lista elemévé tesz, pl.:

```
| ?- memberchk(1,L), memberchk(2,L), memberchk(1,L).
        L = [1,2|_A] ?
```


Nyílt végű listák kezelése `memberchk` segítségével: szótárprogram

```
szótaraz(Sz):-
        read(M-A), !,  % kifejezést olvas be és egyesíti az argumentummal
        memberchk(M-A,Sz),
        write(M-A), nl,
        szótaraz(Sz).
szótaraz(_).
```

Egy futása:

```
| ?- szótaraz(Sz).
|: alma-apple.             |: alma-X.
alma-apple                 alma-apple
|: korte-pear.             |: X-pear.
korte-pear                 korte-pear
                           |: vege.

                           Sz = [alma-apple,korte-pear|_A] ?
```

VEZÉRLÉSI ELJÁRÁSOK

Vezérlési eljárások: call/1

- Vezérlési eljárás: A Prolog végrehajtási mechanizmusához kapcsoloódó beépített eljárás.
- A vezérlési eljárások többsége *magasabbrendű* eljárás, azaz olyan eljárás, amely egy vagy több argumentumát eljáráshívásként értelmezi. (A magasabbrendű Prolog eljárásokat szokás *meta-eljárásnak* is hívni.)
- A meta-eljárások alapvető építőeleme a `call/1` beépített eljárás:
  - Hívási minta: `call(+Cél)`
  - Argumentumok: `Cél` egy „meghívható kifejezés” (callable), azaz struktúra, vagy atom.
  - Jelentése: `Cél` igaz.
  - Hatása: a `Cél` kifejezést eljáráshívássá alakítja és meghívja
- Klóztörzsben célként szereplő `X` változót a rendszer egy `call(X)` hívássá alakítja át.
- `Cél`-ban további vezérlési eljárások is előfordulhatnak, pl. `','/2`, `';'/2`. A `Cél`-beli vágó csak a `call` belsejében vág (szülője a `call(Cél)` hívás).

call/1: példák

```
| ?- [user].
| kétszer(Hívás) :- call(Hívás), Hívás.
| ^D
{consulted user in module user, 0 msec 224 bytes}
yes
 ?- kétszer(write(ba)), nl.
baba
yes
| ?- _Cél = (kétszer(write(ba)), write(' ')), kétszer(_Cél), nl.
baba baba
yes
| ?- listing(kétszer).
kétszer(A) :-
        call(user:A),
        call(user:A).
yes
```


call/1 példa: futási időt mérő meta-eljárás

```
% Kiírja Goal első megoldásának előállításához vagy a meghiúsuláshoz
% szükséges időt, a Txt szöveg kiséretében (lásd: peldak/call_koltsege.pl).
time(Txt, Goal) :-
        statistics(runtime, [T0,_]),  % T0 az indítás óta eltelt CPU idő,
                                      % msec-ban (szemétgyűjtés nélkül).
        (   call(Goal) -> Res = true
        ;   Res = false
        ),
        statistics(runtime, [T1,_]), T is T1-T0,
        format('~w futási idő = ~3d sec\n', [Txt,T]),
        Res = true.
```

A `call/1` viszonylag költséges: egy 1414 hosszú lista megfordítása `nrev`-vel (kb 1 millió `append` hívás), minden `append` körül egy felesleges `call`-lal ill. anélkül:

	`call` nélkül	`call`-lal
lefordítva	0.220 sec	9.710 sec
interpretálva	1.850 sec	11.940 sec

Vezérlési szerkezetek mint eljárások

- A `call/1` argumentumában azért szerepelhetnek vezérlési szerkezetek, mert ezek maguk beépített eljárásként is jelen vannak a Prolog rendszerben:
  - `(',')/2`: konjunkció.
  - `(;)/2`: diszjunkció.
  - `(->)/2`: if-then.
  - `(;)/2`: if-then-else.
- A `call`-ban szereplő vezérlési szerkezetek lényegében ugyanúgy futnak, mint az interpretált (`consult`-tal betöltött) kód.
- Példa

```
| ?- kétszer((member(X, [a,b,c,d]), write(X), fail ; nl)).
abcd
abcd

true ?
```


További vezérlési eljárások

- A „nem-bizonyíthatóságot” ellenőrző `\+` eljárás definíciója:

```
\+ X :- call(X), !, fail.
\+ _X.
```

- Az első megoldásra megszorító `once` beépített eljárás definíciója:

```
once(X) :- call(X), !.
```

- A `true` ill. `fail` beépített eljárás mindig sikerül ill. mindig meghiúsul.
- A `repeat` beépített eljárás egy végtelen választási pontot hoz létre:

```
repeat.
repeat :- repeat.
```

- A `repeat` eljárást mindig egy vágóval semlegesítjük. Példa:

```
fociklus:-  repeat, read(X),
            (   X = end_of_file -> true
            ;   Xsq is X*X, write(Xsq), nl, fail
            ), !.
```