II. RÉSZ

SML-Prolog átvezetés: párhuzamok a két nyelv között

SML

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

SML „Prologosítva”

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

függvény
klóz
változó: egyetlen, ismert érték
minta: csak fordítási időben értelmes
egyszerű minták: `x::x::xs` nem megengedett
egyirányú mintaillesztés
egyértelmű klózválasztás
egy eredmény
egyirányú használat

adatkonstruktor-függvény
egymásba ágyazott függvényhívások

Prolog

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

Prolog „SML-esítve”

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

predikátum
klóz (lazább a kapcsolat a predikátummal)
változó: egy, esetleg ismeretlen érték
minta: teljes jogú adatstruktúra
összetett minták, pl. `[X,X|Xs]`
kétirányú mintaillesztés
többértelmű klózválasztás
több eredmény (nemdeterminizmus)
többirányú használat
(pl. összerakó és szétszedő `append`)
struktúra (rekord)
konjunkció, segéd-változóval

SML-Prolog átvezetés: további példák

SML

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

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

típusos nyelv
magasabbrendű függvény
rekurzió

kivétel

Prolog

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

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

típustalan nyelv
rekurzió, ritkábban magasabbrendű predikátum
visszalépéses ciklus
(pl. két lista közös eleme)
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
- Megoldásgyűjtő eljárások
- Meta-logikai eljárások
- Modularitás
- Magasabbrendű eljárások
- Dinamikus adatbáziskezelés
- Nyelvtani elemzés
- „Hagyományos” beépített eljárások

A KERESÉSI TÉR SZŰKÍTÉSE

Prolog nyelvi eszközök a keresési tér szűkítésére

- Eszközök
  - a vágó beépített eljárás: `!` (az első Prolog rendszerektől kezdve)
  - feltételes diszjunktív szerkezet (későbbi kiterjesztés): `( if -> then ; else )`
- Feltételes szerkezet — procedurális szemantika (ismétlés)
  A `(felt->akkor;egyébként),folyt` célsorozat végrehajtása:
  - Végrehajtjuk a `felt` hívást (egy önálló végrehajtási környezetben).
  - Ha `felt` sikeres, akkor az `akkor,folyt` 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,folyt` célsorozatra redukáljuk.
- Feltételes szerkezet — alternatív procedurális szemantika:
  - A feltételes szerkezetet egy speciális diszjunkciónak tekintjük:

```
(   felt, {vágás}, akkor
;   egyébként
)
```

  - A **{vágás}** jelentése: megszünteti a `felt`-beli választási pontokat, és `egyébként` választását is letiltja.

Feltételes szerkezet: választási pontok a feltételben

- Eddig a főleg determinisztikus (választásmentes) feltételeket mutattunk.
- Példafeladat: `első_poz_elem(L, P)`: `P` az `L` lista első pozitív eleme.
  - Első megoldás, rekurzióval (mérnöki `:-)`)

```
első_poz_elem([EP|_], EP) :- EP > 0.
első_poz_elem([X|L], EP) :- X =< 0, első_poz_elem(L, EP).
```

  - Második megoldás, visszalépéses kereséssel (matematikusi `:-)`)

```
első_poz_elem(L, EP) :-
        append(Nk, [EP|_], L), EP > 0, \+ van_poz_eleme(Nk).

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

  - Harmadik megoldás, feltételes szerkezettel (gyorsprogramozás — Prolog hekker `:-)`)

```
első_poz_elem(L, EP) :-
        (   member(EP, L), EP > 0 -> true
        ;   fail                          % ez a sor elhagyható
        ).
```

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

A vágó eljárás

- A vágó beépített eljárás (neve: `!`) végrehajtása: letiltja a a többi klóz választását és megszünteti az összes választási pontot a klóztörzsben őt megelőző eljáráshívásokban.
- Példák a vágó használatára (lista első pozitív eleme)
  - Mérnöki megoldás:

```
első_poz_elem([EP|_], EP) :- EP > 0, !.
első_poz_elem([X|L], EP) :- X =< 0, első_poz_elem(L, EP).
```

  - Prolog hekker megoldása:

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

- 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 $X \leq 0$ feltételek kizárják egymást, lásd indexelés.)
  - ténylegesen eldobunk megoldásokat — vörös vágás, a program jelentését megváltoztatja
    - (Vörös vágás sokszor úgy keletkezik, hogy egy zöld vágót tartalmazó programban a „felesleges” feltételeket elhagyjuk (pl. az `X =< 0` feltételt a fenti 2. klózban)

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): L utolsó eleme E. (lists könyvtárbeli)
last([E], E) :- !.                              % zöld vágó
last([_|L], E) :- last(L, E).

% pozitívak(+L, -P): P az L pozitív elemeiből áll.
pozitívak([], []).
pozitívak([E|Ek], [E|Pk]) :-
       E > 0, !,                                % vörös vágó
       pozitívak(Ek, Pk).
pozitívak([_E|Ek], Pk) :-
       /* \+ _E > 0, */ pozitívak(Ek, Pk).
```

Figyelem: a fenti példák nem tökéletesek, hatékonyabb ill. általánosabban használható 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
r(X):- t(X).      % az r(X) többi klózának választását --- a vágót tartalmazó
                  % klóz mellett kötelezzük el magunkat (commit)
```

- A vágó szemléltetése a 4-kapus doboz modellben: a vágó **Újra** kapujából egyenesesen a körülvevő (**szülő**) doboz **Meghiúsulási** kapujára megyünk.

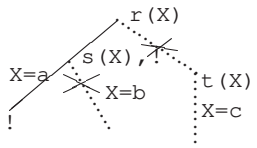
A vágó által megszüntetett választási pontok

```
% 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 megvalósítása a vágó segítségével

- 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 (ilyenkor a Prolog megvalósítás egy speciális, ún. távolbaható vágót használ).
- Vágót rendkívül ritkán szükséges feltételes szerkezetben szerepeltetni.

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)
        ).
% pozitívak(+L, ?Pk): Pk az L pozitív elemeiből áll.
pozitívak([], []).
pozitívak([E|Ek], Pk) :-
        (   E > 0 -> Pk = [E|Pk0]
        ;   Pk = Pk0
        ),
        pozitívak(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. SML-ben), 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 */.
```

Diszjunktív feltételes szerkezet

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

Általános alak

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

Általános alak

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


Feltételes szerkezetek és fejillesztés

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

```
% a vágó előtt fej-egyesítés:
abs(X, X) :- X >= 0, !.
abs(X, A) :- A is -X.
```

```
% az egyesítés explicitté téve:
abs(X, A) :- A = X, X >= 0, !.
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: csak akkor helyettesíti be a kimenő paramétert, ha már tudja, mi az értéke (nincs „előre-behelyettesítés”, mint a fenti első két példában).
  - (**„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).

% pozitívak(+L, ?Pk): Pk az L pozitív elemeiből áll.
pozitívak([], []).
pozitívak([E|Ek], Pk) :-
        E > 0, !, Pk = [E|Pk0], pozitívak(Ek, Pk0).
pozitívak([_E|Ek], Pk) :-
        /* \+ _E > 0, */ pozitívak(Ek, Pk).
```

Megjegyzés: a diszjunktív alakban a feltételek eleve explicitek, nincs fejillesztési probléma, ezért **a diszjunktív feltételes szerkezet használatát javasoljuk a vágó helyett**.

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

- 1. változat, tiszta Prolog. Lassú (előre-behelyettesítés, két hasonlítás), választási pontot hagy.

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

- 2. változat, zöld vágóval. Lassú (előre-behelyettesítés, két hasonlítás), nem hagy választási pontot.

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

- 3. változat, vörös vágóval. Gyorsabb (előre-behelyettesítés, egy hasonlítás), 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, nagyon gyors (egy hasonlítás, nincs előre-behelyettesítés) és nem is hoz létre 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

- 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` (`lists` könyvtár)

- `memberchk/2` definíciója:

```
% 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), !,
        % A read(X) beépített eljárás egy kifejezést
        % olvas be és egyesíti X-szel
        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.     % nem egyesíthető M-A-val

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


VEZÉRLÉSI ELJÁRÁSOK

Vezérlési eljárások, a `call/1` beépített eljárás

- Vezérlési eljárás: A Prolog végrehajtáshoz kapcsolódó beépített eljárás (pl. vágó, if-then-else).
- 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 fő képviselője és alapvető építőeleme a `call/1`:
  - Hívási minta: `call(+Cél)`
  - `Cél` egy „meghívható kifejezés” (callable, vö. `callable/1`), azaz struktúra, vagy névkonstans.
  - Jelentése (deklaratív szemantika): `Cél` igaz.
  - Hatása (procedurális szemantika): a `Cél` kifejezést eljáráshívássá alakítja és meghívja.
- A klóztörzsben célként megengedett egy `X` változó használata, ezt a rendszer egy `call(X)` hívássá alakítja át.

```
| kétszer(Hívás) :- call(Hívás), Hívás.

| ?- kétszer(write(ba)), nl.      --->  baba
| ?- listing(kétszer).            --->  kétszer(A) :-
                                            call(user:A), call(user:A).
```

Vezérlési szerkezetek mint eljárások

- A `call/1` argumentumában szerepelhetnek vezérlési szerkezetek is, 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.
- Példák:

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


`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]),
                    % ~w formázó: kiírás a write/1 segítségével
                    % ~3d formázó: I egész kiírása I/1000-ként, 3 tizedesre
        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	Lassulás
lefordítva	0.140 sec	1.680 sec	12.00
interpretálva	1.710 sec	3.520 sec	2.06

További beépített vezérlési eljárások

- `\+ Cél`: `Cél` „nem bizonyítható”. A beépített eljárás definíciója:

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

- `once(Cél)`: `Cél` igaz, és csak az első megoldását kérjük. Definíciója:

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

- `true`: azonosan igaz (mindig sikerül), `fail`: azonosan hamis (mindig meghiúsul).
- `repeat`: végtelen sokszor igaz (egy végtelen választási pontot hoz létre). Definíciója:

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

- A `repeat` eljárást legtöbbször egy mellékhatásos eljárás ismétlésére használjuk. A végtelen választási pontot kötelező egy vágóval semlegesíteni.
- Példa (egyszerű kalkulátor):

```
bc :-  repeat, read(Expr),
            (   Expr = end_of_file -> true
            ;   Res is Expr, write(Expr = Res), nl, fail
            ),
       !.
```


Példa: magasabbrendű reláció definiálása

- Az implikáció (`P` $\Rightarrow$ `Q`) megvalósítása negáció segítségével:

```
% P minden megoldása esetén Q igaz.
forall(P, Q) :-
     \+ (P, \+Q). % Szintaktikus emlékeztető:
                  % az első \+ után kötelező a szóköz!

| ?- _L = [1,2,3],
     % _L minden eleme pozitív:
     forall(member(X, _L), X > 0).
true ?
| ?- _L = [1,-2,3], forall(member(X, _L), X > 0).
no
| ?- _L = [1,2,3],
     % _L szigorúan monoton növő:
     forall(append(_,[A,B|_], _L), A < B).
true ?
```

- `forall/2` csak eldöntendő kérdés esetén használható.

DETERMINIZMUS ÉS INDEXELÉS

Determinizmus

- Egy eljáráshívás **determinisztikus**, ha (legfeljebb) egyféleképpen sikerülhet.
- Egy eljáráshívásnak egy sikeres végrehajtása **determinisztikusan futott le**:
  - ha nem hagyott választási pontot a híváshoz tartozó részfában, azaz
    - vagy választásmentesen futott le, azaz létre sem hozott választási pontot (figyelem: ez a Prolog megvalósítástól függ!);
    - vagy létrehozott ugyan választási pontot, de megszüntette (kimerítette, levágta).
- A SICStus Prolog nyomkövetésében **?** jelzi a **nem**determinisztikus lefutást:

```
p(1, a).
p(2, b).
p(3, b).
```

```
| ?- p(1, X).                        % det. hívás,
                1 1 Exit: p(1,a)     %      det. lefutás
| ?- p(Y, a).                        % det. hívás,
              ? 1 1 Exit: p(1,a)     %      nemdet. lefutás
| ?- p(Y, b), Y > 2.                 % nemdet. hívás
              ? 1 1 Exit: p(2,b)     %      nemdet. lefutás
                1 1 Exit: p(3,b)     %      det. lefutás
```


A determinisztikus lefutás

- Mi a determinisztikus lefutás haszna?
  - a futás gyorsabb lesz,
  - a tárigény csökken,
  - más optimalizálások (pl. jobbrekurzió) alkalmazható.
- Hogyan ismeri fel a fordító azt, hogy nem kell választási pont?
  - indexelés (indexing)
  - vágók és feltételes szerkezetek
- Az alábbi definíciók esetén a `p(`*`Nonvar`*`, Y)` hívás nem hoz létre választási pontot:

```
p(1, a).
p(2, b).
```

```
p(1, Y) :- !,
    Y = a.
p(_, b).
```

```
p(X, Y) :-
    (   X =:= 1 -> Y = a
    ;   Y = b
    ).
```


Indexelés — ismétlés

- Mi az indexelés?
  - egy adott hívásra 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 funktora:
  - `C` szám vagy névkonstans esetén `C/0`;
  - `R` nevű és `N` argumentumú struktúra esetén `R/N`;
  - változó esetén nem értelmezett.
- Az indexelés megvalósítása:
  - Fordításkor a funktorokhoz elkészítjük az illeszthető klózok részhalmazát.
  - Futáskor lényegében konstans idő alatt választunk a részhalmazok közül.
  - **Fontos:** ha egyelemű a részhalmaz, nem hozunk létre választási pontot!

Példa indexelésre

```
p(0, a).          /* (1) */
p(X, t) :- q(X).  /* (2) */
p(s(0), b).       /* (3) */
p(s(1), c).       /* (4) */
p(9, z).          /* (5) */
```

```
q(1).
q(2).
```

- A `p(A, B)` hívással illesztendő klózhalmaz:
  - `{(1) (2) (3) (4) (5)}` ha `A` változó;
  - `{(1) (2)}` ha `A = 0`;
  - `{(2) (3) (4)}` ha `A` fő funktora `s/1`;
  - `{(2) (5)}` ha `A = 9`;
  - `{(2)}` minden más esetben.
- Példák hívásokra:
  - `p(1, Y)` nem hoz létre választási pontot.
  - `p(s(1), Y)` létrehoz választási pontot, de determinisztikusan fut le.
  - `p(s(0), Y)` nemdeterminisztikusan fut le.

Struktúrák, változók a fejargumentumban

- Azonos funktorú struktúrák az első fejargumentumban:
  - Ha a klózok szétválasztásához szükség van az első (struktúra) argumentum részeire is, akkor érdemes segédeljárást bevezetni.
  - Például `p/2` és `q/2` ekvivalens, de `q(Nonvar, Y)` determinisztikusan fut le!

```
p(0, a).
p(s(0), b).
p(s(1), c).
p(9, z).
```

```
q(0, a).
q(s(X), Y) :-
     q_seged(X, Y).
q(9, z).
```

```
q_seged(0, b).
q_seged(1, c).
```

- Fejillesztés kiváltása egyenlőséggel (vö. SML rétegelt minta)
  - Az indexelés figyelembe veszi a törzs elején szereplő egyenlőséget: `p(X, ...) :- X = Kif, ...` esetén `Kif` funktora szerint indexel.
  - Példa: lista hosszának reciproka, üres lista esetén 0:

```
rhossz([], 0).
rhossz(L, RH) :- L = [_|_], length(L, H), RH is 1/H.
% rhossz([X|L], RH) :-  length([X|L], H), RH is 1/H.
        % kevésbé hatékony, mert újra felépíti az [X|L] listát.
% rhossz(L, RH) :-  L \= [], length(L, H), RH is 1/H.
        % kevésbé hatékony, mert L=[] esetben választási pontot hagy.
```


Indexelés — további tudnivalók

- Indexelés és aritmetika
  - Az indexelés nem foglalkozik aritmetikai vizsgálatokkal.
  - Pl. az `N = 0` és `N > 0` feltételek nem „zárják ki” egymást.
  - Az alábbi `fakt/2` eljárás lefutása nem-determinisztikus:

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

- Indexelés és listák
  - Gyakran kell az üres és nem-üres lista esetét szétválasztani.
  - A bemenő lista-argumentumot célszerű az első argumentum-pozícióba tenni.
  - Az `[]` és `[...|...]` eseteket az indexelés megkülönbözteti (funktoruk: `'[]'/0` ill. `'.'/2`).
  - A két klóz sorrendje nem érdekes (feltéve, hogy zárt listával hívjuk az első pozíción) — de azért tegyük a leálló klózt mindig előre.

Listakezelő eljárások indexelése: példák

- Az `append/3` választásmentesen fut le, ha első argumentuma zárt végű lista.

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

- A `last/2` közvetlen megfogalmazása nemdeterminisztikusan fut le:

```
% last(L, E): Az L lista utolsó eleme E.
last([E], E).
last([_|L], E) :- last(L, E).
```

- Érdemes segédeljárást bevezetni, `last2/2` választásmentesen fut

```
last2([X|L], E) :- last2(L, X, E).

% last2(L, X, E): Az [X|L] lista utolsó eleme E.
last2([], E, E).
last2([X|L], _, E) :- last2(L, X, E).
```

- Az utolsó listaelemet választásmentesen felsoroló member (`lists` könyvtárból):

```
member(E, [H|T]) :-     member_(T, H, E).

% member_(L, X, E): Az [X|L] lista eleme E.
member_(_, E, E).
member_([H|T], _, E) :- member_(T, H, E).
```

Az indexelés és a vágó kölcsönhatása

- Hogyan vehető figyelembe a vágó az indexelés fordításakor?
- Példa: a `p(1, A)` hívás választásmentes, de a `q(1, A)` nem!

```
p(1, Y) :- !, Y = 2.      % (1)
p(X, X).                  % (2)
Arg1=1 → (1), Arg1≠1 → (2)
```

```
q(1, 2) :- !.             % (1)
q(X, X).                  % (2)
Arg1=1 → {(1),(2)}, Arg1≠1 → (2)
```

- A fordító figyelembe veszi a vágót az indexelésben, ha garantált, hogy egy adott fő funktor esetén a vágót elérjük. Ennek feltételei:
  - az első argumentum változó, konstans, vagy csak változókat tartalmazó struktúra legyen,
  - a további argumentumok változók legyenek,
  - a fejben az összes változóelőfordulás különböző legyen,
  - a törzs első hívása a vágó (megengedve a fejillesztést kiváltó egyenlőséget).
- Ilyenkor a fordító az adott funktorhoz tartozó listából kihagyja a vágót követő klózokat.
- Példa: `p(X, D, E) :- X = s(A, B, C), !, ....  p(X, Y, Z) :- ....`
- Ez egy újabb érv a vágás alapszabálya mellett:

A kimenő paraméterek értékadását mindig a vágó után végezzük!

A vágó és az indexelés hatékonysága

- Egy Fibonacci-szerű sorozat: $f_1 = 1;\ \ f_2 = 2;\ \ f_n = f_{\lceil 3n/4 \rceil} + f_{\lfloor 2n/3 \rfloor},\ \ n > 2$

```
% determinisztikus
fib(1, 1).
fib(2, 2).
fib(N, F) :-
        N > 2,
        N2 is N*3//4,
        N3 is N*2//3,
        fib(N2, F2),
        fib(N3, F3),
        F is F2+F3.
```

```
% determ. lefutású
fibc(1, 1) :- !.
fibc(2, 2) :- !.
fibc(N, F) :-
        N > 2,
        N2 is N*3//4,
        N3 is N*2//3,
        fibc(N2, F2),
        fibc(N3, F3),
        F is F2+F3.
```

```
% választásmentes
fibci(1, F) :- !, F = 1.
fibci(2, F) :- !, F = 2.
fibci(N, F) :-
        N > 2,
        N2 is N*3//4,
        N3 is N*2//3,
        fibci(N2, F2),
        fibci(N3, F3),
        F is F2+F3.
```

- Futási idők $N = 2000$ esetén

	`fib`	`fibc`	`fibci`
futási idő	990 msec	890 msec	830 msec
meghiúsulási idő	440 msec	30 msec	0 msec
összesen	1430 msec	920 msec	830 msec
nyom-verem mérete	4.1Mbyte	2.0 Mbyte	256 byte

- `fibc` esetén a meghiúsulási idő azért nem 0, mert a rendszer a nyom-vermet (trail-stack) dolgozza fel. A nyom-verem tárolja a változó-értékadások visszacsinálási információit.

Választás-mentesség diszjunktív feltételes szerkezetek esetén

- Feltételes szerkezet végrehajtásakor általában választási pont jön létre.
- A **SICStus Prolog** a „`( felt -> akkor ; egyébként )`” szerkezetet választásmentesen hajtja végre, ha a `felt` konjunkció tagjai csak:
  - aritmetikai összehasonlító eljáráshívások (pl. `<, =<, =:=`), és/vagy
  - kifejezés-típust ellenőrző eljáráshívások (pl. `atom, number`), és/vagy
  - általános összehasonlító eljáráshívások (ld. később, pl. `@<, @=<, ==`).
- Analóg módon választásmentes kód keletkezik a „`fej :- felt, !, akkor.`” klózból, ha `fej` argumentumai különböző változók, és `felt` olyan mint fent.
- Például választásmentes kód keletkezik az alábbi definíciókból:

```
vektorfajta(X, Y, Fajta) :-
    (   X =:= 0, Y =:= 0
        % X = 0, Y = 0  nem lenne jó
    ->  Fajta = null
    ;   Fajta = nem_null
    ).
```

```
vektorfajta(X, Y, Fajta) :-
    X =:= 0, Y =:= 0, !,
    Fajta = null.
vektorfajta(_X, _Y, nem_null).
```


JOBBREKURZIÓ ÉS AKKUMULÁTOROK