V. rész

Haladó Prolog

1 Bevezetés

2 Cékla: deklaratív programozás C++-ban

3 Prolog alapok

4 Erlang alapok

5 Haladó Prolog

6 Haladó Erlang

- Az előző Prolog 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

- Megoldásgyűjtő eljárások
- Meta-logikai eljárások
- A keresési tér szűkítése
- Vezérlési eljárások
- A Prolog megvalósítási módszereiről
- Determinizmus és indexelés
- Jobbrekurzió, akkumulátorok
- Kényelmi eszközök: Definite Clause Grammars (DCG), ciklusok
- Imperatív programok átírása Prologba
- Modularitás
- Magasabbrendű eljárások
- Dinamikus adatbáziskezelés
- „Hagyományos” beépített eljárások
- Fejlettebb nyelvi és rendszerelemek

Tartalom

5 Haladó Prolog
- Megoldásgyűjtő beépített eljárások
- Meta-logikai eljárások
- A keresési tér szűkítése
- Vezérlési eljárások
- A Prolog megvalósítási módszereiről
- Determinizmus és indexelés
- Jobbrekurzió és akkumulátorok
- Kényelmi eszközök: Definite Clause Grammars (DCG), ciklusok
- Listák és fák akkumulálása – példák
- Imperatív programok átírása Prologba
- Modularitás
- Magasabbrendű eljárások
- Dinamikus adatbáziskezelés
- Egy összetettebb példaprogram
- „Hagyományos” beépített eljárások
- Fejlettebb nyelvi és rendszerelemek

Visszalépéses keresési feladatok – egy aritmetikai példa

- Példa: a SEND+MORE=MONEY feladat
- A program:

```
% dec1(J): J egy pozitív decimális számjegy.
dec1(1). dec1(2). dec1(3). dec1(4).
dec1(5). dec1(6). dec1(7). dec1(8). dec1(9).

% dec(J): J egy decimális számjegy.
dec(0).
dec(J) :- dec1(J).

% sendmory(L): L a SEND+MORE=MONEY feladatmegoldása
sendmory(L):-
        L = [S,E,N,D,M,O,R,Y], all_different(L),
        dec1(S), dec(E), dec(N), dec(D),
        dec1(M), dec(O), dec(R), dec(Y),
        S*1000+E*100+N*10+D + M*1000+O*100+R*10+E =:=
        M*10000+O*1000+N*100+E*10+Y.
```


Visszalépéses keresés – számintervallum felsorolása

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

```
% between(M, N, I): M =< I =< N, I egész.
between(M, N, M) :-
        M =< N.
between(M, N, I) :-
        M < N,
        M1 is M+1,
        between(M1, N, I).

% dec(X): X egy decimális számjegy
dec(X) :- between(0, 9, X).

| ?- between(1, 2, _X), between(3, 4, _Y), Z is 10*_X+_Y.
Z = 13 ? ; Z = 14 ? ; Z = 23 ? ; Z = 24 ? ;
no
```

- A fenti eljárás (optimalizált változata) elérhető a `between` könyvtárban.

Keresési feladat Prologban – felsorolás vagy gyűjtés?

- Keresési feladat: adott feltételeknek megfelelő dolgok meghatározása.
- Prolog nyelven egy ilyen feladat alapvetően kétféle módon oldható meg:
  - gyűjtés – az összes megoldás összegyűjtése, pl. egy listába;
  - felsorolás – a megoldások visszalépéses felsorolása: egyszerre egy megoldást kapunk, de visszalépéssel sorra előáll minden megoldás.
- Egyszerű példa: egy lista páros elemeinek megkeresése:

% Gyűjtés:

```
% páros_elemei(L, Pk): Pk az L
% lista páros elemeinek listája.
páros_elemei([], []).
páros_elemei([X|L], Pk) :-
    X mod 2 =\= 0,
    páros_elemei(L, Pk).
páros_elemei([P|L], [P|Pk]) :-
    P mod 2 =:= 0,
    páros_elemei(L, Pk).
```

% Felsorolás:

```
% páros_eleme(L, P): P egy páros
% eleme az L listának.
páros_eleme([X|L], P) :-
    X mod 2 =:= 0, P = X.
páros_eleme([_X|L], P) :-
    % _X akár páros, akár páratlan
    % folytatjuk a felsorolást:
    páros_eleme(L, P).

% egyszerűbb megoldás:
páros_eleme2(L, P) :-
    member(P, L), P mod 2 =:= 0.
```


Gyűjtés és felsorolás kapcsolata

- Vizsgáljuk meg, hogyan lehet egy felsoroló eljárást visszavezetni a gyűjtőre, és fordítva:
  - felsorolás gyűjtésből: a `member/2` könyvtári eljárás segítségével, pl.

    ```
    páros_eleme(L, P) :-
        páros_elemei(L, Pk), member(P, Pk).
    ```

    Természetesen ez így nem hatékony!
  - gyűjtés felsorolásból: a megoldásgyűjtő beépített eljárások segítségével, pl.

    ```
    páros_elemei(L, Pk) :-
        findall(P, páros_eleme(L, P), Pk).
        % A páros_eleme(L, P) cél
        % összes P megoldásának listája Pk.
    ```
  - a `findall/3` beépített eljárás – és társai – az Erlang listanézetnek felelnek meg, pl.:

    ```
    % seq(+A, +B, ?L): L = [A,...,B], A és B egészek.
    seq(A, B, L) :-
        B >= A-1
        findall(X, between(A, B, X), L).
    ```

A `findall(?Gyűjtő, :Cél, ?Lista)` beépített eljárás

- Az eljárás végrehajtása (procedurális szemantikája):
  - a `Cél` kifejezést eljáráshívásként értelmezi, meghívja (A `:` annotáció meta- (azaz eljárás) argumentumot jelez);
  - minden egyes megoldásához előállítja `Gyűjtő` egy *másolatát*, azaz a változókat, ha vannak, szisztematikusan újakkal helyettesíti;
  - Az összes `Gyűjtő` másolat listáját egyesíti `Lista`-val.
- Példák az eljárás használatára:

```
| ?- findall(X, (member(X, [1,7,8,3,2,4]), X>3), L).
                ⟹     L = [7,8,4] ? ; no
| ?- findall(X-Y, (between(1, 3, X), between(1, X, Y)), L).
                ⟹     L = [1-1,2-1,2-2,3-1,3-2,3-3] ? ; no
```

- Az eljárás jelentése (deklaratív szemantikája):
  `Lista` $=$ { `Gyűjtő` másolat | ($\exists$ `X` ... `Z`)`Cél` igaz }
  ahol `X`, ..., `Z` a `findall` hívásban levő szabad változók (azaz olyan, a hívás pillanatában behelyettesítetlen változók, amelyek a `Cél`-ban előfordulnak de a `Gyűjtő`-ben nem).

A `bagof(?Gyűjtő, :Cél, ?Lista)` beépített eljárás

- Az eljárás végrehajtása (procedurális szemantikája):
  - a `Cél` kifejezést eljáráshívásként értelmezi, meghívja;
  - összegyűjti a megoldásait (a `Gyűjtő`-t és a szabad változók behelyettesítéseit);
  - a szabad változók összes behelyettesítését *felsorolja* és mindegyik esetén a `Lista`-ban megadja az összes hozzá tartozó `Gyűjtő` értéket.
- Példák az eljárás használatára:

```
gráf([a-b,a-c,b-c,c-d,b-d]).

| ?- gráf(_G), findall(B, member(A-B, _G), VegP).
                              ⟹    VegP = [b,c,c,d,d] ? ; no
| ?- gráf(_G), bagof(B, member(A-B, _G), VegP).
                              ⟹    A = a, VegP = [b,c] ? ;
                                   A = b, VegP = [c,d] ? ;
                                   A = c, VegP = [d] ? ; no
```

- A `bagof` eljárás jelentése (deklaratív szemantikája):
  `Lista` $=$ { `Gyűjtő` | `Cél` igaz }, `Lista` $\neq$ `[]`.

A `bagof` megoldásgyűjtő eljárás (folyt.)

- Explicit egzisztenciális kvantorok
  - `bagof(Gyűjtő, V1 ^ ...^ Vn ^ Cél, Lista)` alakú hívása a `V1, ..., Vn` változókat egzisztenciálisan kvantáltnak tekinti, így ezeket nem sorolja fel.
  - jelentése: `Lista` $=$ { `Gyűjtő` | ($\exists$ `V1`, ..., `Vn`)`Cél` igaz } $\neq$ `[]`.

```
| ?- gráf(_G), bagof(B, A^member(A-B, _G), VegP).
                              ⟹    VegP = [b,c,c,d,d] ? ; no
```

- Egymásba ágyazott gyűjtések
  - szabad változók esetén a `bagof` nemdeterminisztikus lehet, így skatulyázható:

```
% A G irányított gráf fokszámlistája FL:
% FL = { A-N | N = |{ V | A-V ∈ G }|, N > 0 }
fokszámai(G, FL) :-
    bagof(A-N, Vk^(bagof(V, member(A-V, G), Vk),
                   length(Vk, N)                ), FL).

| ?- gráf(_G), fokszámai(_G, FL).
                ⟹    FL = [a-2,b-2,c-1] ? ; no
```


A `bagof` megoldásgyűjtő eljárás (folyt.)

- Fokszámlista hatékonyabb előállítása
  - meta-argumentumban a célsorozat interpretáltan fut
  - segédeljárás bevezetésével a kvantor is szükségtelenné válik:

```
% Az A pont foka a G irányított gráfban N, N>0.
pont_foka(A, G, N) :-
        bagof(V, member(A-V, G), Vk), length(Vk, N).

% A G irányított gráf fokszámlistája FL:
fokszámai(G, FL) :-    bagof(A-N, pont_foka(A, G, N), FL).
```

- Példák a `bagof/3` és `findall/3` közötti kisebb különbségekre:

```
| ?- findall(X, (between(1, 5, X), X<0), L). ⇒ L = [] ? ; no
| ?-   bagof(X, (between(1, 5, X), X<0), L). ⇒ no
| ?- findall(S, member(S, [f(X,X),g(X,Y)]), L).
                              ⇒ L = [f(_A,_A),g(_B,_C)] ? ; no
| ?-   bagof(S, member(S, [f(X,X),g(X,Y)]), L).
                              ⇒ L = [f(X,X),g(X,Y)] ? ; no
```

- A `bagof/3` logikailag tisztább mint a `findall/3`, de időigényesebb!

A `setof(?Gyűjtő, :Cél, ?Lista)` beépített eljárás

- az eljárás végrehajtása:
  - ugyanaz mint: `bagof(Gyűjtő, Cél, L0), sort(L0, Lista),`
  - itt `sort/2` egy univerzális rendező eljárás (lásd később), amely
  - az eredménylistát rendezi (az ismétlődések kiszűrésével).
- Példa a `setof/3` eljárás használatára:

```
gráf([a-b,a-c,b-c,c-d,b-d]).

% Gráf egy pontja P.
pontja(P, Gráf) :- member(A-B, Gráf), ( P = A ; P = B).

% A G gráf pontjainak listája Pk.
gráf_pontjai(G, Pk) :- setof(P, pontja(P, G), Pk).

| ?- gráf(_G), gráf_pontjai(_G, Pk). ⟹   Pk = [a,b,c,d] ? ; no
```


Tartalom

5 Haladó Prolog
- Megoldásgyűjtő beépített eljárások
- Meta-logikai eljárások
- A keresési tér szűkítése
- Vezérlési eljárások
- A Prolog megvalósítási módszereiről
- Determinizmus és indexelés
- Jobbrekurzió és akkumulátorok
- Kényelmi eszközök: Definite Clause Grammars (DCG), ciklusok
- Listák és fák akkumulálása – példák
- Imperatív programok átírása Prologba
- Modularitás
- Magasabbrendű eljárások
- Dinamikus adatbáziskezelés
- Egy összetettebb példaprogram
- „Hagyományos” beépített eljárások
- Fejlettebb nyelvi és rendszerelemek

A meta-logikai, azaz a logikán túlmutató eljárások fajtái:

- A Prolog kifejezések pillanatnyi behelyettesítettségi állapotát vizsgáló eljárások (értelemszerűen logikailag nem tiszták):
  - kifejezések osztályozása (1)

```
| ?- var(X) /* X változó? */, X = 1. ⟹   X = 1
| ?- X = 1, var(X). ⟹   no
```

  - kifejezések rendezése (4)

```
| ?- X @< 3 /* X megelőzi 3-t? */, X = 4. ⟹   X = 4
   % a változók megelőzik a nem változó kifejezéseket
| ?- X = 4, X @< 3. ⟹   no
```

- Prolog kifejezéseket szétszedő vagy összerakó eljárások:
  - (struktúra) kifejezés $\Longleftrightarrow$ név és argumentumok (2)

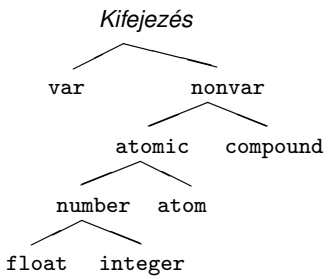
```
| ?- X = f(alma,körte), X =.. L ⟹   L = [f,alma,körte]
```

  - névkonstansok és számok $\Longleftrightarrow$ karaktereik (3)

```
| ?- atom_codes(A, [0'a,0'b,0'a]) ⟹   A = aba
```


Kifejezések osztályozása

- Kifejezésfajták – osztályozó beépített eljárások (ismétlés)



`var(X)`	`X` változó
`nonvar(X)`	`X` nem változó
`atomic(X)`	`X` konstans
`compound(X)`	`X` struktúra
`atom(X)`	`X` atom
`number(X)`	`X` szám
`integer(X)`	`X` egész szám
`float(X)`	`X` lebegőpontos szám

- SICStus-specifikus osztályozó eljárások:
  - `simple(X)`: `X` nem összetett (konstans vagy változó);
  - `callable(X)`: `X` atom vagy struktúra (nem szám és nem változó);
  - `ground(X)`: `X` tömör, azaz nem tartalmaz behelyettesítetlen változót.
- Az osztályozó eljárások használata – példák
  - `var`, `nonvar` – többirányú eljárásokban elágaztatásra
  - `number`, `atom`, ... – nem-megkülönböztetett úniók feldolgozása (pl. szimbolikus deriválás)

Osztályozó eljárások: elágaztatás behelyettesítettség alapján

- Példa: a `length/2` beépített eljárás megvalósítása

```
% length(?L, ?N): Az L lista N hosszú.
length(L, N) :-    var(N), length(L, 0, N).
length(L, N) :- nonvar(N), dlength(L, 0, N).
```

```
% length(?L, +I0, -I):
%   Az L lista I-I0 hosszú.
length([], I, I).
length([_|L], I0, I) :-
        I1 is I0+1,
        length(L, I1, I).
```

```
% dlength(?L, +I0, +I):
%   Az L lista I-I0 hosszú.
dlength([], I, I).
dlength([_|L], I0, I) :-
        I0<I, I1 is I0+1,
        dlength(L, I1, I).
```

```
| ?- length([1,2], Len).   (length/3)   ⇒  Len = 2 ? ; no
| ?- length([1,2], 3).     (dlength/3)  ⇒  no
| ?- length(L, 3).         (dlength/3)  ⇒  L = [_A,_B,_C] ?;no
| ?- length(L, Len).       (length/3)   ⇒  L = [], Len = 0 ? ;
                                           L = [_A], Len = 1 ? ; L = [_A,_B], Len = 2 ?
```


Struktúrák szétszedése és összerakása: az *univ* eljárás

- Az *univ* eljárás hívási mintái: `+Kif =..  ?Lista`
  `-Kif =..  +Lista`
- Az eljárás jelentése:
  - `Kif` = $Fun(A_1, \ldots, A_n)$ és `Lista` = $[Fun, A_1, \ldots, A_n]$, ahol Fun egy névkonstans és $A_1, \ldots, A_n$ tetszőleges kifejezések; vagy
  - `Kif` = C és `Lista` = $[C]$, ahol C egy konstans.
- Példák

```
| ?- el(a,b,10) =.. L.          ⇒   L = [el,a,b,10]
| ?- Kif =.. [el,a,b,10].       ⇒   Kif = el(a,b,10)
| ?- alma =.. L.                ⇒   L = [alma]
| ?- Kif =.. [1234].            ⇒   Kif = 1234
| ?- Kif =.. L.                 ⇒   hiba
| ?- f(a,g(10,20)) =.. L.       ⇒   L = [f,a,g(10,20)]
| ?- Kif =.. [/,X,2+X].         ⇒   Kif = X/(2+X)
| ?- [a,b,c] =.. L.             ⇒   L = ['.',a,[b,c]]
```


Struktúrák szétszedése és összerakása: a `functor` eljárás

- `functor/3`: kifejezés funktorának, adott funktorú kifejezésnek az előállítása
  - Hívási minták: `functor(-Kif, +Név, +Argszám)`
    `functor(+Kif, ?Név, ?Argszám)`
  - Jelentése: `Kif` egy `Név/Argszám` funktorú kifejezés.
    - A konstansok 0-argumentumú kifejezésnek számítanak.
    - Ha `Kif` kimenő, az adott funktorú legáltalánosabb kifejezéssel egyesíti (argumentumaiban csupa különböző változóval).
- Példák:

```
| ?- functor(el(a,b,1), F, N).     ⇒   F = el, N = 3
| ?- functor(E, el, 3).            ⇒   E = el(_A,_B,_C)
| ?- functor(alma, F, N).          ⇒   F = alma, N = 0
| ?- functor(Kif, 122, 0).         ⇒   Kif = 122
| ?- functor(Kif, el, N).          ⇒   hiba
| ?- functor(Kif, 122, 1).         ⇒   hiba
| ?- functor([1,2,3], F, N).      ⇒   F = '.', N = 2
| ?- functor(Kif, ., 2).           ⇒   Kif = [_A|_B]
```


Struktúrák szétszedése és összerakása: az `arg` eljárás

- `arg/3`: kifejezés adott sorszámú argumentuma.
  - Hívási minta: `arg(+Sorszám, +StrKif, ?Arg)`
  - Jelentése: A `StrKif` struktúra `Sorszám`-adik argumentuma `Arg`.
  - Végrehajtása: `Arg`-ot az adott sorszámú argumentummal **egyesíti**.
  - Az `arg/3` eljárás így nem csak egy argumentum elővételére, hanem a struktúra változó-argumentumának behelyettesítésére is használható (ld. a 2. példát alább).
- Példák:

```
| ?- arg(3, el(a, b, 23), Arg).      ⇒   Arg = 23
| ?- K=el(_,_,_), arg(1, K, a),
     arg(2, K, b), arg(3, K, 23).    ⇒   K = el(a,b,23)
| ?- arg(1, [1,2,3], A).            ⇒   A = 1
| ?- arg(2, [1,2,3], B).            ⇒   B = [2,3]
```

- Az *univ* visszavezethető a `functor` és `arg` eljárásokra (és viszont), például:

```
Kif =.. [F,A1,A2]   ⟺   functor(Kif, F, 2),
                        arg(1, Kif, A1), arg(2, Kif, A2)
```

Az *univ* alkalmazása: ismétlődő sémák összevonása

- A feladat: egy szimbolikus aritmetikai kifejezésben a kiértékelhető (infix) részkifejezések helyettesítése az értékükkel.
- 1. megoldás, *univ* nélkül:

```
% Az X szimbolikus kifejezés egyszerűsítése EX.
egysz0(X, X) :- atomic(X).
egysz0(U+V, EKif) :-
    egysz0(U, EU), egysz0(V, EV),
    kiszamol(EU+EV, EU, EV, EKif).
egysz0(U*V, EKif) :-
    egysz0(U, EU), egysz0(V, EV),
    kiszamol(EU*EV, EU, EV, EKif).
%...
% EU és EV részekből képzett EUV egyszerűsítése EKif.
kiszamol(EUV, EU, EV, EKif) :-
    (   number(EU), number(EV) -> EKif is EUV.
    ;   EKif = EUV
    ).
```

```
| ?- deriv((x+y)*(2+x), x, D), egysz0(D, ED).
     ⟹    D = (1+0)*(2+x)+(x+y)*(0+1), ED = 1*(2+x)+(x+y)*1 ? ; no
```


Az *univ* alkalmazása: ismétlődő sémák összevonása (folyt.)

- Kifejezés-egyszerűsítés, 2. megoldás, *univ* segítségével

```
egysz(X, EX) :-
    atomic(X), EX = X.
egysz(Kif, EKif) :-
    Kif =.. [Muv,U,V],     % Kif = Muv(U,V)
    egysz(U, EU), egysz(V, EV),
    EUV =.. [Muv,EU,EV],   % EUV = Muv(EU,EV)
    kiszamol(EUV, EU, EV, EKif).
```

- Kifejezés-egyszerűsítés, általánosítás tetszőleges *tömör* kifejezésre:

```
egysz1(Kif, EKif) :-
    Kif =.. [M|ArgL], egysz1_lista(ArgL, EArgL), EKif0 =.. [M|EArgL],
    % catch(:Cél,?Kiv,:KCél): ha Cél kivételt dob, KCél-t futtatja:
    catch(EKif is EKif0, _, EKif = EKif0).

egysz1_lista([], []).
egysz1_lista([K|Kk], [E|Ek]) :-
    egysz1(K, E), egysz1_lista(Kk, Ek).
```

```
| ?- egysz1(f(1+2+a, exp(3,2), a+1+2), E). ⟹    E = f(3+a,9.0,a+1+2)
```


Általános kifejezés-bejárás *univ*-val: kiiratás

- A feladat: egy tetszőleges kifejezés kiiratása úgy, hogy
  - a kétargumentumú operátorok zárójelezett infix formában,
  - minden más alap-struktúra alakban jelenjék meg.

```
ki(Kif) :- compound(Kif), Kif =.. [Func, A1|ArgL],
    (   % kétargumentumú kifejezés, funktora infix operátor
        ArgL = [A2], current_op(_, Kind, Func), infix_fajta(Kind)
    ->  write('('), ki(A1),
        write(' '), write(Func), write(' '), ki(A2), write(')')
    ;   write(Func), write('('), ki(A1), listaki(ArgL), write(')')
    ).
ki(Kif) :- simple(Kif), write(Kif).

% infix_fajta(F): F egy infix operátorfajta.
infix_fajta(xfx). infix_fajta(xfy). infix_fajta(yfx).

% Az [A1,...,An] listát ",A1,...,An" alakban kiírja.
listaki([]).
listaki([A|AL]) :- write(','), ki(A), arglistaki(AL).
```

```
| ?- ki(f(+a, X*c*X, e)). ⟹    f(+(a),((_117 * c) * _117),e)
```


Általános kifejezés-bejárás *univ*-val: változómentesítés

- A SICStus Prologban beépített `numbervars(?Kif, +N0, ?N)` eljárás:
  - A tetsz. `Kif` minden változóját helyettesíti egy `'$VAR'(I)` struktúrával, `I = N0, ..., N-1` (azaz `Kif`-ben `N-N0` különböző változó van).
- A `'$VAR'(0), '$VAR'(1), ...` kifejezések `write`-tal való kiíráskor változónévként (`A, B ...`) jelennek meg.
- A `write_term(Kif, Opciók)` beépített eljárás kiírja a `Kif` kifejezést, az `Opciók` által meghatározott módon.
- A `numbervars/3` által létrehozott `'$VAR'/1` struktúrák „eredetiben” is megjeleníthetők:

```
| ?- _K = [f(_X),g(_),_X], numbervars(_K, 0, N), write(_K), nl,
          write_term(_K, [quoted(true),numbervars(false)]), nl.
===>    [f(A),g(B),A]
        [f('$VAR'(0)),g('$VAR'(1)),'$VAR'(0)]
        N = 2
```

- A feladat: elkészítendő egy `numbervars1/3` eljárás, amely `'$VAR'` helyett `'$myvar'` funktort használ.

Változómentesítés (folyt.)

- A változómentesítés egy saját megvalósítása:

```
% A Term kifejezésben levő változókat '$myvar(I)' stb.
% struktúrákkal helyettesíti be, I = N0, ... N-1.
numbervars1(Term, N0, N) :-
        (   var(Term) ->
            Term = '$myvar'(N0), N is N0+1.
        ;   Term =.. [_|Args],
            numbervars1_list(Args, N0, N)
        ).
% numbervars1_list(L, N0, N): Az L listában levő változókat
% '$myvar(I)' stb. struktúrákkal helyettesíti be, I = N0, ... N-1.
numbervars1_list([], N, N).
numbervars1_list([A|As], N0, N) :-
        numbervars1(A, N0, N1), numbervars1_list(As, N1, N).

| ?- Kif = [f(_X),g(_),_X], numbervars1(Kif, 0, N).
====>       N = 2,
            Kif = [f('$myvar'(0)),g('$myvar'(1)),'$myvar'(0)]
```


A `numbervars1` eljárás egy alkalmazása

Két kifejezés azonossága

- *A* és *B* azonosak, ha változó-behelyettesítés *nélkül* egyesíthetőek; így:
- ha *A* változót tartalmaz, akkor *B*-ben ugyanott ugyanaz a változó áll, és viszont.
- `azonos/2` == néven, `nem_azonos/2` \== néven beépített eljárás és operátor.

```
nem_azonos(X, Y) :-
    (   numbervars1(X, 0, N), numbervars1(Y, N, _), X = Y -> fail
    ;   true
    ).

azonos(X, Y) :- \+ nem_azonos(X, Y).

% azonos2/2 és azonos/2 teljesen ekvivalens.
% \+ \+ Hívás ≡ Hívás, de változóbehelyettesítést nem okoz
azonos2(X, Y) :-
    \+ \+ (numbervars1(foo(X,Y), 0, _), X = Y).

| ?- azonos(X, 1).                          ---> no
| ?- azonos(X, Y).                          ---> no
| ?- azonos(X, X).                          ---> true ?
| ?- append([], L1, L2), azonos(L1, L2).    ---> L2 = L1 ?
```


Univ alkalmazása: részkifejezések keresése

- A feladat: egy tetszőleges kifejezéshez soroljuk fel a benne levő számokat, és minden szám esetén adjuk meg annak a *kiválasztóját*!
- Egy részkifejezés kiválasztója egy olyan lista, amely megadja, mely argumentumpozíciók mentén juthatunk el hozzá.
- Az $[i_1, i_2, \ldots, i_k]$ lista egy `Kif`-ből az i_1-edik argumentum i_2-edik argumentumának, ... i_k-adik argumentumát választja ki.
- Pl. `a*b+f(1,2,3)/c`-ben `b` kiválasztója `[1,2]`, `3` kiválasztója `[2,1,3]`.

```
% kif_szám(?Kif, ?N, ?Kiv): Kif Kiv kiválasztójú része az N szám.
kif_szám(X, X, []) :-
        number(X).
kif_szám(X, N, [I|Kiv]) :-
        compound(X), % a változó kizárása miatt fontos!
        functor(X, _F, ArgNo), between(1, ArgNo, I), arg(I, X, X1),
        kif_szám(X1, N, Kiv).

| ?- kif_szám(f(1,[b,2]), N, K).
====> K = [1], N = 1 ? ;
      K = [2,2,1], N = 2 ? ; no
```


Atomok szétszedése és összerakása

- `atom_codes/2`: névkonstans és karakterkód-lista közötti átalakítás
  - Hívási minták: `atom_codes(+Atom, ?KódLista)`
    `atom_codes(-Atom, +KódLista)`
  - Jelentése: `Atom` karakterkódjainak a listája `KódLista`.
  - Végrehajtása:
    - Ha `Atom` adott (bemenő), és a $c_1 c_2 \ldots c_n$ karakterekből áll, akkor `KódLista`-t egyesíti a $[k_1, k_2, \ldots, k_n]$ listával, ahol k_i a c_i karakter kódja.
    - Ha `KódLista` egy adott karakterkód-lista, akkor ezekből a karakterekből összerak egy névkonstanst, és azt egyesíti `Atom`-mal.
- Példák:

```
| ?- atom_codes(ab, Cs).            ⟹   Cs = [97,98]
| ?- atom_codes(ab, [0'a|L]).       ⟹   L = [98]
| ?- Cs="bc", atom_codes(Atom, Cs). ⟹   Cs = [98,99], Atom = bc
| ?- atom_codes(Atom, [0'a|L]).     ⟹   hiba
```

Atomok szétszedése és összerakása – példák

- Keresés névkonstansokban

```
% Atom-ban a Rész nem üres részatom kétszer ismétlődik.
dadogó_rész(Atom, Rész) :-
    atom_codes(Atom, Cs),
    Ds = [_|_],
    append([_,Ds,Ds,_]. Cs),
    atom_codes(Rész, Ds).
```

```
| ?- dadogó_rész(babaruhaha, R).    ⟹    R = ba ? ; R = ha ? ; no
```

- Atomok összefűzése

```
% atom_concat(+A, +B, ?C): A és B névkonstansok összefűzése C.
% (Szabványos beépített eljárás atom_concat(?A, ?B, +C) módban is.)
atom_concat(A, B, C) :-
    atom_codes(A, Ak), atom_codes(B, Bk),
    append(Ak, Bk, Ck),
    atom_codes(C, Ck).
```

```
| ?- atom_concat(abra, kadabra, A). ⟹    A = abrakadabra ?
```


Számok szétszedése és összerakása

- `number_codes/2`: szám és karakterkód-lista közötti átalakítás
  - Hívási minták: `number_codes(+Szám, ?KódLista)`
    `number_codes(-Szám, +KódLista)`
  - Jelentése: Igaz, ha `Szám` tizes számrendszerbeli alakja a `KódLista` karakterkód-listának felel meg.
  - Végrehajtása:
    - Ha `Szám` adott (bemenő), és a $c_1c_2...c_n$ karakterekből áll, akkor `KódLista`-t egyesíti a $[k_1, k_2, ..., k_n]$ kifejezéssel, ahol k_i a c_i karakter kódja.
    - Ha `KódLista` egy adott karakterkód-lista, akkor ezekből a karakterekből összerak egy számot (ha nem lehet, hibát jelez), és azt egyesíti `Szám`-mal.
- Példák:

```
| ?- number_codes(12, Cs).          ⟹   Cs = [49,50]
| ?- number_codes(0123, [0'1|L]).   ⟹   L = [50,51]
| ?- number_codes(N, " - 12.0e1").  ⟹   N = -120.0
| ?- number_codes(N, "12e1").       ⟹   hiba (nincs .0)
| ?- number_codes(120.0, "12e1").   ⟹   no  (mert a szám adott! :-)
```


Kifejezések rendezése: szabványos sorrend

- A Prolog szabvány definiálja két tetszőleges Prolog kifejezés sorrendjét.
- Jelölés: $X \prec Y$ – az X kifejezés megelőzi az Y kifejezést.
- A szabványos sorrend definíciója:
  1. X és Y azonos $\Leftrightarrow$ $X \prec Y$ és $Y \prec X$ egyike sem igaz.
  2. Ha X és Y különböző osztályba tartozik, akkor az osztály dönt: *változó* $\prec$ *lebegőpontos szám* $\prec$ *egész szám* $\prec$ *név* $\prec$ *struktúra*.
  3. Ha X és Y változó, akkor sorrendjük rendszerfüggő.
  4. Ha X és Y lebegőpontos vagy egész szám, akkor $X \prec Y \Leftrightarrow X < Y$.
  5. Ha X és Y név, akkor a lexikografikus (abc) sorrend dönt.
  6. Ha X és Y struktúrák:
     1. Ha X és Y aritása ($\equiv$ argumentumszáma) különböző, akkor $X \prec Y \Leftrightarrow X$ aritása kisebb mint Y aritása.
     2. Egyébként, ha a struktúrák neve különböző, akkor $X \prec Y \Leftrightarrow X$ neve $\prec Y$ neve.
     3. Egyébként (azonos név, azonos aritás) balról az első nem azonos argumentum dönt.
- (A SICStus Prologban kiterjesztésként megengedett végtelen (ciklikus) kifejezésekre a fenti rendezés nem érvényes.)

Kifejezések összehasonlítása – beépített eljárások

- Két tetszőleges kifejezés összehasonlítását végző eljárások:

hívás	igaz, ha
`Kif1 == Kif2`	`Kif1` $\not\prec$ `Kif2` $\wedge$ `Kif2` $\not\prec$ `Kif1`
`Kif1 \== Kif2`	`Kif1` $\prec$ `Kif2` $\vee$ `Kif2` $\prec$ `Kif1`
`Kif1 @< Kif2`	`Kif1` $\prec$ `Kif2`
`Kif1 @=< Kif2`	`Kif2` $\not\prec$ `Kif1`
`Kif1 @> Kif2`	`Kif2` $\prec$ `Kif1`
`Kif1 @>= Kif2`	`Kif1` $\not\prec$ `Kif2`

- Az összehasonlítás mindig a belső ábrázolás (kanonikus alak) szerint történik:

```
| ?- [1, 2, 3, 4] @< struktúra(1, 2, 3).    ⟹    sikerül (6.1 szabály)
```

Meta-logikai eljárások alkalmazása: $\prec$ megvalósítása

```
% T1 megelőzi T2-t. Ekvivalens T1 @< T2 -vel, kivéve a változókat.
precedes(T1, T2) :- \+ \+ (numbervars(T1-T2, 0, _), prec(T1, T2)).

% class(+T, -C): A T kifejezés a C-edik kifejezésosztályba tartozik.
class(T, C) :-
        (   T='$VAR'(_) -> C=0        % változó
        ;   float(T)    -> C=1        % lebegőpontos szám
        ;   integer(T)  -> C=2        % egész szám
        ;   atom(T)     -> C=3        % névkonstans
        ;   compound(T) -> C=4        % összetett kifejezés
        ).

% T1 megelőzi T2-t, a változók helyett már '$VAR'(n) áll.
prec(T1, T2) :- class(T1, C1), class(T2, C2),
        (   C1 =:= C2 ->
            (   C1 =:= 1 -> T1 < T2   % 4. szabály (lebegőpontos szám)
            ;   C1 =:= 2 -> T1 < T2   % 4. szabály (egész szám)
            ;   struct_prec(T1, T2)   % 3., 5. és 6. szabály
            )                         % (változó, név, struktúra)
        ;   C1 < C2                   % 2. szabály
        ).
```


A $\prec$ reláció megvalósítása (folyt.)

```
% S1 megelőzi S2-t (S1 és S2 struktúra-kifejezés vagy névkonstans).
struct_prec(S1, S2) :-
        functor(S1, F1, N1), functor(S2, F2, N2),
        (   N1 < N2 -> true
        ;   N1 = N2,
            (   F1 = F2 -> args_prec(1, N1, S1, S2)
            ;   atom_prec(F1, F2)
            )
        ).

% Az S1 struktúra-kifejezés N0, ..., N sorszámú argumentumai
% lexikografikusan megelőzik S2 azonos sorszámú argumentumait.
args_prec(N0, N, S1, S2) :- N0 =< N,
        arg(N0, S1, A1), arg(N0, S2, A2),
        (   A1 = A2 -> N1 is N0+1, args_prec(N1, N, S1, S2)
        ;   prec(A1, A2)
        ).

% Az A1 névkonstans megelőzi az A2 névkonstanst.
atom_prec(A1, A2) :-
    atom_codes(A1, C1), atom_codes(A2, C2), struct_prec(C1, C2).
```


Összefoglalás: a Prolog egyenlőség-szerű beépített eljárásai

- `U = V`: `U` egyesítendő `V`-vel. Soha sem jelez hibát.

```
| ?- X = 1+2.      ⟹   X = 1+2
| ?- 3 = 1+2.      ⟹   no
```

- `U == V`: `U` azonos `V`-vel. Soha sem jelez hibát és soha sem helyettesít be.

```
| ?- X == 1+2.     ⟹   no
| ?- 3 == 1+2.     ⟹   no
| ?- +(1,2)==1+2   ⟹   yes
```

- `U =:= V`: Az `U` és `V` aritmetikai kifejezések értéke megegyezik. Hibát jelez, ha `U` vagy `V` nem (tömör) aritmetikai kifejezés.

```
| ?- X =:= 1+2.    ⟹   hiba
| ?- 1+2 =:= X.    ⟹   hiba
| ?- 2+1 =:= 1+2.  ⟹   yes
| ?- 2.0 =:= 1+1.  ⟹   yes
```

- `U is V`: `U` egyesítendő a `V` aritmetikai kifejezés értékével. Hiba, ha `V` nem (tömör) aritmetikai kifejezés.

```
| ?- 2.0 is 1+1.   ⟹   no
| ?- X is 1+2.     ⟹   X = 3
| ?- 1+2 is X.     ⟹   hiba
| ?- 3 is 1+2.     ⟹   yes
| ?- 1+2 is 1+2.   ⟹   no
```

- (`U =.. V`: `U` „szétszedettje” a `V` lista)

```
| ?- 1+2 =.. X.    ⟹   X = [+,1,2]
| ?- X =.. [f,1].  ⟹   X = f(1)
```


Összefoglalás: a Prolog nem-egyenlő jellegű beépített eljárásai

A nem-egyenlőség jellegű eljárások soha sem helyettesítenek be változót!

- `U \= V`: `U` nem egyesíthető `V`-vel. Soha sem jelez hibát.

```
| ?- X \= 1+2.      ⟹   no
| ?- +(1,2) \= 1+2. ⟹   no
```

- `U \== V`: `U` nem azonos `V`-vel. Soha sem jelez hibát.

```
| ?- X \== 1+2.     ⟹   yes
| ?- 3 \== 1+2.     ⟹   yes
| ?- +(1,2)\==1+2   ⟹   no
```

- `U =\= V`: Az `U` és `V` aritmetikai kifejezések értéke különbözik. Hibát jelez, ha `U` vagy `V` nem (tömör) aritmetikai kifejezés.

```
| ?- X =\= 1+2.     ⟹   hiba
| ?- 1+2 =\= X.     ⟹   hiba
| ?- 2+1 =\= 1+2.   ⟹   no
| ?- 2.0 =\= 1+1.   ⟹   no
```

A Prolog (nem-)egyenlőség jellegű beépített eljárásai – példák

		Egyesítés		*Azonosság*		*Aritmetika*		
U	*V*	*U* = *V*	*U* \= *V*	*U* == *V*	*U* \== *V*	*U* =:= *V*	*U* =\= *V*	*U* is *V*
1	2	*no*	yes	*no*	yes	*no*	yes	*no*
a	b	*no*	yes	*no*	yes	error	error	error
1+2	+(1,2)	yes	*no*	yes	*no*	yes	*no*	*no*
1+2	2+1	*no*	yes	*no*	yes	yes	*no*	*no*
1+2	3	*no*	yes	*no*	yes	yes	*no*	*no*
3	1+2	*no*	yes	*no*	yes	yes	*no*	yes
X	1+2	X=1+2	*no*	*no*	yes	error	error	X=3
X	Y	X=Y	*no*	*no*	yes	error	error	error
X	X	yes	*no*	yes	*no*	error	error	error

Jelmagyarázat: `yes` – siker; *no* – meghiúsulás, error – hiba.

Tartalom

5 Haladó Prolog
- Megoldásgyűjtő beépített eljárások
- Meta-logikai eljárások
- A keresési tér szűkítése
- Vezérlési eljárások
- A Prolog megvalósítási módszereiről
- Determinizmus és indexelés
- Jobbrekurzió és akkumulátorok
- Kényelmi eszközök: Definite Clause Grammars (DCG), ciklusok
- Listák és fák akkumulálása – példák
- Imperatív programok átírása Prologba
- Moduláritás
- Magasabbrendű eljárások
- Dinamikus adatbáziskezelés
- Egy összetettebb példaprogram
- „Hagyományos” beépített eljárások
- Fejlettebb nyelvi és rendszerelemek

Prolog nyelvi eszközök a keresési tér szűkítésére

- Eszközök
  - Az első Prolog rendszerektől kezdve: vágó, szabványos jelölése `!`
  - Későbbi kiterjesztés: az `( if -> then ; else )` feltételes szerk.
- Feltételes szerkezet – procedurális szemantika (ismétlés)
  A `(felt->akkor;egyébként),folyt` célsorozat végrehajtása:
  - Végrehajtjuk a `felt` hívást (egy önálló végrehajtási környezetben).
  - Ha `felt` sikeres $\Longrightarrow$ „`akkor,folyt`” célsorozattal folytatjuk, a `felt` első megoldása által eredményezett behelyettesítésekkel. A `felt` cél többi megoldását nem keressük meg!
  - Ha `felt` meghiúsul $\Longrightarrow$ „`egyébként,folyt`” célsorozattal folytatjuk.
- 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 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([X|_],  X) :- X > 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(NemPozL, [EP|_], L), EP > 0,
        \+ van_poz_eleme(NemPozL).

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

  - Harmadik megoldás, feltételes szerkezettel (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, és `első_poz_elem(+,+)` módban hibásan működhet!

A vágó eljárás

- A vágó beépített eljárás (`!`) végrehajtása:
  1. letiltja az adott predikátum további klózainak választását,
     ```
     első_poz_elem([X|_], X) :- X > 0, !.
     első_poz_elem([X|L], EP) :- X =< 0, első_poz_elem(L, EP).
     ```
  2. megszünteti a választási pontokat az előtte levő eljáráshívásokban.
     ```
     első_poz_elem(L, EP) :-  member(EP, L), EP > 0, !.
     ```
- Miért vágunk le ágakat a keresési térben?
  - Mi tudjuk, hogy nincs megoldás, de a Prolog nem – zöld vágó
    - (Például, a legtöbb Prolog megvalósítás „nem tudja”, hogy a $X > 0$ és $X \leq 0$ feltételek kizárják egymást.)
  - Eldobunk megoldásokat – vörös vágó, ez a program jelentését megváltoztatja
    - (Vörös vágó lesz a zöldből ha 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
fakt(N, F) :- N > 0, N1 is N-1, fakt(N1, F1), F is N*F1.

% last(+L, ?E): L utolsó eleme E.
last([E], E) :- !.                                      zöld
last([_|L], Last) :- last(L, Last).

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

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

A vágó definíciója

- Segédfogalom: egy cél **szülőjének** az őt tartalmazó klóz fejével illesztett hívást nevezzük
  - A 4-kapus modellben a szülő a körülvevő dobozhoz rendelt cél.
  - Pl. `last([E], E) :- !.` – a vágó szülője lehet a `last([7], X)` hívás.
  - A `g` nyomkövető parancs a cél őseit listázza ki.
- A vágó végrehajtása:
  - mindig sikerül; de mellékhatásként a végrehajtás adott állapotától visszafelé egészen a szülő célig – azt is beleértve – megszünteti a választási pontokat.
- 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 való megszorítás
r(X):- t(X).      % az r(X) további klózainak választását – a vágót tartalmazó
                  % klóz mellett való elköteleződés (commit)
```

- A vágó szemléltetése a 4-kapus doboz modellben: a vágó `Redo` kapujából a körülvevő (szülő) doboz `Fail` kapujára megyünk.

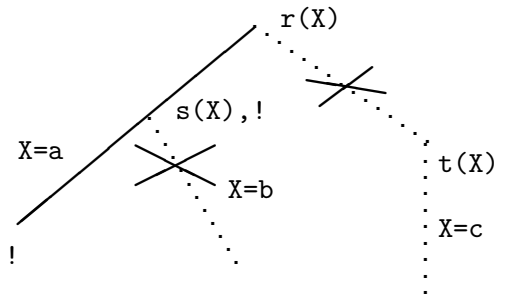
A vágó által megszüntetett választási pontok

```
% 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 visszavezetése vágóra

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

```
p :-                                p :-
    aaa,                                aaa, segéd(...), zzz.

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

- Az `egyébként` ág elmaradhat, ilyenkor a megfelelő klóz is elmarad.
- A `felt` részekben értelmetlen vágót használni
- 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.

A vágás első alapesete – klóz mellett való elkötelezés

- A klóz melletti elkötelezés egy 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.
```

  De: a fenti két alak csak akkor ekvivalens, ha `feltétel` egyszerű, nincs benne választás.
- Analógia: ha `a`, `b` és `c` Boole-értékű változók, 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 és fejillesztés

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

```
% abs(X, A): A = |X| (A az X abszolút értéke).
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 vágó előtt van fej-egyesítés  % az egyesítés explicitté téve
```

- 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 példákban).
  - (**„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], Last)  :- last(L, Last).

% 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**.

Példa: `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 (hekkelésre :-);
  - végtelen választási pontot létrehozó eljárások megszelidítésére.
- Eldöntendő kérdés: eljáráshívás csupa bemenő paraméterrel

```
% egy_komponensbeli(+A, +B, +Gráf):
% Az A és B pontok a G gráfnak ugyanabban a komponensében vannak.
egy_komponensbeli(A, B, Graf) :-
        utvonal(A, B, Graf), !.
```

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

Feladatspecifikus optimalizálás

- A feladat: megállapítandó egy lista első fennsíkjának a hossza. (*Fennsíknak* nevezzük egy számlista olyan folytonos, nem üres részlistáját, amelyik pozitív számokból áll és semelyik irányban sem terjeszthető ki.)

```
% Az L lista első fennsíkjának a hossza H.
efhossz(L, H) :-
        append(_NemFennsik, FennsikMaradek, L),
        FennsikMaradek = [X|_], X > 0, !,
        append(Fennsik, Maradek, FennsikMaradek),
        (   Maradek = []
        ;   Maradek = [Y|_], Y =< 0
        ), !,
        length(Fennsik, H).
```

- a fenti diszjunkció kiváltható egy negációval:

```
\+␣( Maradek = [Y|_], Y > 0 )
```


Végtelen választás megszelidítése: `memberchk`

- A `memberchk/2` beépített eljárás Prolog definíciója:

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

% 1. változat
memberchk(X, L):-
        member(X, L), !.

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


Tartalom

5 Haladó Prolog
- Megoldásgyűjtő beépített eljárások
- Meta-logikai eljárások
- A keresési tér szűkítése
- Vezérlési eljárások
- A Prolog megvalósítási módszereiről
- Determinizmus és indexelés
- Jobbrekurzió és akkumulátorok
- Kényelmi eszközök: Definite Clause Grammars (DCG), ciklusok
- Listák és fák akkumulálása – példák
- Imperatív programok átírása Prologba
- Modularitás
- Magasabbrendű eljárások
- Dinamikus adatbáziskezelés
- Egy összetettebb példaprogram
- „Hagyományos” beépített eljárások
- Fejlettebb nyelvi és rendszerelemek

Vezérlési eljárások, a `call/1` beépített eljárás

- Vezérlési eljárás: A Prolog végrehajtáshoz kapcsolódó 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 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.
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 költséges: egy 4472 hosszú lista megfordítása nrev-vel (kb. 10 millió append hívás), minden append körül egy felesleges call-lal ill. anélkül:

	call nélkül	call-lal	Lassulás
lefordítva	0.47 sec	2.46 sec	5.23
interpretálva	6.97 sec	8.66 sec	1.24

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), !.                    vagy
  once(X) :- (   call(X) -> true   ).
  ```
- true: azonosan igaz, fail: azonosan hamis (mindig meghiúsul).
- repeat: végtelen sokszor igaz (végtelen választási pont). Definíciója:
  ```
  repeat.
  repeat :- repeat.
  ```
- A repeat eljárást egy mellékhatásos eljárás ismétlésére használhatjuk. 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 ⇒ 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.

Tartalom

5 Haladó Prolog
- Megoldásgyűjtő beépített eljárások
- Meta-logikai eljárások
- A keresési tér szűkítése
- Vezérlési eljárások
- A Prolog megvalósítási módszereiről
- Determinizmus és indexelés
- Jobbrekurzió és akkumulátorok
- Kényelmi eszközök: Definite Clause Grammars (DCG), ciklusok
- Listák és fák akkumulálása – példák
- Imperatív programok átírása Prologba
- Modularitás
- Magasabbrendű eljárások
- Dinamikus adatbáziskezelés
- Egy összetettebb példaprogram
- „Hagyományos” beépített eljárások
- Fejlettebb nyelvi és rendszerelemek

A Prolog megvalósítás néhány mérföldköve

- 1973: Marseille Prolog (A. Colmerauer et al.)
  - értelmező (interpreter), Fortran nyelven
  - kifejezések ábrázolása: struktúra-osztásos (structure-sharing)
  - veremszervezés: egyetlen verem (csak visszalépéskor szabadul fel)
- 1977: DEC-10 Prolog (D. H. D. Warren)
  - fordítóprogram Prolog és assembly nyelven (+ értelmező Prologban)
  - kifejezések ábrázolása: struktúra-osztásos
  - veremszervezés: három verem (visszalépéskor felszabadulnak)
    - globális verem (global): struktúra-beli változók, szemétgyűjtött
    - fő/lokális verem (main/local): eljárások, választási pontok, változók, determinisztikus lefutáskor felszabadul
    - nyom (trail): változó-behelyettesítések (vágónál felszabadítható)
- 1983: WAM — Warren Abstract Machine (D. H. D. Warren)
  - absztrakt gép Prolog programok végrehajtására
  - kifejezések ábrázolása: struktúra-másolásos (structure-copying)
  - három verem, (ld. DEC-10), a struktúrákat a globális verem tárolja
  - A legtöbb mai Prolog WAM alapú (SICStus, SWI, GNU Prolog, . . .)

Struktúrák ábrázolása

- A kétféle kifejezés-ábrázolás összehasonlítása:

	struktúra-osztásos	struktúra-másolásos
tárigény:	O(változók száma)	O(struktúra mérete)
struktúra-építés ideje	konstans	O(struktúra mérete)
struktúra-szétszedés	költségesebb	kevésbé költséges

- Struktúra építése: egy változónak és egy programszövegbeli struktúrának az egyesítése
- FONTOS: egy változó értékeként megjelenő struktúra egyesítése egy behelyettesítetlen változóval mindenképpen konstans költségű!
- Példa:

```
hosszabbít(L, [1,2,3,...,n|L]).
sokszoroz(0, L) :- !, L = [].
sokszoroz(N, L) :-
        hosszabbít(L0, L), N1 is N-1, sokszoroz(N1, L0).
```

- `sokszoroz(n, L)` idő- és tárigénye struktúra-osztásnál $O(n)$, struktúra-másolásnál $O(n^2)$
- A gyakorlatban mégis a struktúra-másolásos megoldás a hatékonyabb.

WAM: Prolog kifejezések tárolása (LBT – low bit tagging scheme)

- Behelyettesítetlen változó:
- Változóra való utalás:
- Névkonstans:
- Egész szám:
- Lista:

globális vagy fő verem *csak globális verem*

saját cím		REF
másik vált. címe		REF
atom tábla index	A	CON
egész érték	I	CON
cím		LIST

cím:

fej-kifejezés
farok-kifejezés

- Struktúra:

cím	STRU

cím:

funktor tábla index
argumentum-kif.
...

- A SICStus 3.x rendszer a 4 legmagasabb helyiértékű biten tárolja jelzőket (tag) — ezért a veremterületek mérete 256 Mbyte-ban korlátozott. (SICStus 4-ben már LBT séma van.)

WAM: néhány további részlet

- Változók kezelése
  - Két változó egyesítése: a fiatalabbik = az öregebbre utaló REF
  - Utalástalanítás: az (esetleg többtagú) REF-lánc követése
  - Behelyettesítetlen változó ≡ önmagára mutató utalás ⇒ egyszerűbb utalástalanítás
- Visszalépés
  - Feltételes változó: olyan behelyettesítetlen változó, amely öregebb mint a legfrissebb választási pont
  - Feltételes változó behelyettesítése esetén a változó címét beírjuk a nyom-verembe
  - Visszalépéskor a nyom alapján „visszacsináljuk” a változó-behelyettesítéseket, majd a vermeket visszahúzzuk
- SICStus WAM visszafejtő (nem dokumentált, változhat):

```
| ?- use_module(library(disassembler)).
(...)
| ?- disassemble(<predikátum>/<argumentumszám>).
```

- A WAM bemutatása (tutorial): `http://wambook.sourceforge.net/`

Tartalom

5 Haladó Prolog
- Megoldásgyűjtő beépített eljárások
- Meta-logikai eljárások
- A keresési tér szűkítése
- Vezérlési eljárások
- A Prolog megvalósítási módszereiről
- Determinizmus és indexelés
- Jobbrekurzió és akkumulátorok
- Kényelmi eszközök: Definite Clause Grammars (DCG), ciklusok
- Listák és fák akkumulálása – példák
- Imperatív programok átírása Prologba
- Modularitás
- Magasabbrendű eljárások
- Dinamikus adatbáziskezelés
- Egy összetettebb példaprogram
- „Hagyományos” beépített eljárások
- Fejlettebb nyelvi és rendszerelemek

Determinizmus

- Egy hívás determinisztikus, ha (legfeljebb) egyféleképpen sikerülhet.
- Egy eljáráshívás egy sikeres végrehajtása determinisztikusan futott le, ha nem hagyott választási pontot a híváshoz tartozó részfában:
  - 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 nemdeterminisztikus 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 és a választásmentesség

- Mi a determinisztikus lefutás haszna?
  - a futás gyorsabb lesz,
  - a tárigény csökken,
  - más optimalizálások (pl. jobbrekurzió) alkalmazható.
- Hogyan ismerheti fel a fordító a választásmentességet
  - egyszerű feltételes szerkezet (vö. Erlang őrfeltétel)
  - indexelés (indexing)
  - vágó és indexelés kölcsönhatása
- Az alábbi definíciók esetén a `p(`*Nonvar*`, Y)` hívás választásmentes, azaz nem hoz létre választási pontot:

Egyszerű feltétel

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

Indexelés

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

Indexelés és vágó

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


Választásmentesség 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 (pl. `@<`, `@=<`,`==`).
- 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).
```

Indexelés

- 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:
  - Fordítási időben: funktor ⇒ illeszthető fejű klózok részhalmaza .
  - Futási időben: a részhalmaz lényegében konstans idejű kiválasztása (hash tábla használatával).
  - Fontos: ha egyelemű a részhalmaz, nincs választási pont!

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

- Ha a klózok szétválasztásához szükség van az első (struktúra) argumentum részeire is, akkor érdemes segédeljárást bevezetni.
- Pl. `p/2` és `q/2` ekvivalens, de `q(`*Nonvar*`, Y)` determinisztikus lefutású!

```
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).
```

- 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.
```

- A 2. klóz kevésbé hatékony változatai

```
rhossz([X|L], RH) :-  length([X|L], H), RH is 1/H.
                      % ^ újra felépíti [X|L]-t.
rhossz(L, RH) :-  L \= [], length(L, H), RH is 1/H.
                      % L=[] esetén választási pontot hagy.
```


Indexelés – további tudnivalók

- Indexelés és aritmetika
  - Az indexelés nem foglalkozik aritmetikai vizsgálatokkal.
  - Pl. az `N = 0` és `N > 0` feltételek esetén a SICStus Prolog nem veszi figyelembe, hogy ezek kizárják 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.

```
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/2`:

```
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:
  - 1. arg. változó, konstans, vagy csak változókat tartalmazó struktúra,
  - a további argumentumok változók,
  - a fejben az összes változóelőfordulás különböző,
  - a törzs első hívása a vágó (előtte megengedve egy fejillesztést kiváltó egyenlőséget).
- Ekkor az adott funktorhoz tartozó listából kihagyja a vágó utáni 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

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

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

- Futási idők $N = 6000$ esetén

	fib	fibc	fibci
futási idő	1.25 sec	1.22 sec	1.13 sec
meghiúsulási idő	0.29 sec	0.03 sec	0.00 sec
összesen	1.54 sec	1.25 sec	1.13 sec
nyom-verem mérete	37.4Mbyte	18.7 Mbyte	240 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.)

Tartalom

5 Haladó Prolog
- Megoldásgyűjtő beépített eljárások
- Meta-logikai eljárások
- A keresési tér szűkítése
- Vezérlési eljárások
- A Prolog megvalósítási módszereiről
- Determinizmus és indexelés
- Jobbrekurzió és akkumulátorok
- Kényelmi eszközök: Definite Clause Grammars (DCG), ciklusok
- Listák és fák akkumulálása – példák
- Imperatív programok átírása Prologba
- Modularitás
- Magasabbrendű eljárások
- Dinamikus adatbáziskezelés
- Egy összetettebb példaprogram
- „Hagyományos” beépített eljárások
- Fejlettebb nyelvi és rendszerelemek

Jobbrekurzió (farok-rekurzió, tail-recursion) optimalizálás

- Az általános rekurzió költséges, helyben és időben is.
- Jobbrekurzióról beszélünk, ha
  - a rekurzív hívás a klóztörzs utolsó helyén van, vagy az utolsó helyen szereplő diszjunkció egyik ágának utolsó helyén stb., és
  - a rekurzív hívás pillanatában nincs választási pont a predikátumban (a rekurzív hívást megelőző célok determinisztikusan futottak le, nem maradt nyitott diszjunkciós ág).
- Jobbrekurzió optimalizálás: az utolsó hívás végrehajtása előtt az eljárás által lefoglalt hely felszabadul ill. szemétgyűjtésre alkalmassá válik.
- Ez az optimalizálás nemcsak rekurzív hívás esetén, hanem minden **utolsó** hívás esetén megvalósul – a pontos név: utolsó hívás optimalizálás (last call optimisation).
- A jobbrekurzió így tehát nem növeli a memória-igényt, korlátlan mélységig futhat – mint a ciklusok az imperatív nyelvekben. Példa:

```
ciklus(Állapot) :- lépés(Állapot, Állapot1), !, ciklus(Állapot1).
ciklus(_Állapot).
```


Predikátumok jobbrekurzív alakra hozása – listaösszeg

- A listaösszegzés „természetes”, nem jobbrekurzív definíciója:

```
% sum0(+L, ?S): L elemeinek összege S (S = 0+L_n+L_{n-1}+...+L_1).
sum0([], 0).
sum0([X|L], S):-        sum0(L,S0), S is S0+X.
```

- Jobbrekurzív lista-összegző:

```
% sum(+L, ?S): L  elemeinek összege S (S = 0+L_1+L_2+...+L_n).
sum(L, S):-             sum(L, 0, S).

% sum(+L, +S0, ?S): L elemeit S0-hoz adva kapjuk S-t. (≡  Σ L = S-S0)
sum([], S, S).
sum([X|L], S0, S):-     S1 is S0+X, sum(L, S1, S).
```

- A jobbrekurzív `sum` eljárás több mint 3-szor gyorsabb mint a `sum0`!
- Az akkumulátor az imperatív (azaz megváltoztatható értékű) változó fogalmának deklaratív megfelelője:
  - A `sum/3`-ban az `S0` és `S` argumentumok akkumulátorpárt alkotnak.
  - Az akkumulátorpár két része az adott változó mennyiség (a példában az összeg) különböző időpontokban vett értékeit mutatja:
    - `S0` az összeg a `sum/3` meghívásakor: a változó kezdőértéke;
    - `S` az összeg a `sum/3` lefutása után: a változó végértéke.

Az akkumulátorok használata

- Az akkumulátorokkal általánosan több egymás utáni változtatást is leírhatunk:

```
p(..., A0, A):-
        q0(..., A0, A1), ...,
        q1(..., A1, A2), ...,
        qn(..., An, A).
```

- A `sum3/3` második klóza ilyen alakra hozva:

```
sum3([X|L], S0, S):- plus(X, S0, S1), sum3(L, S1, S).

plus(X, S0, S) :- S is S0+X.
```

- Akkumulátorváltozók elnevezési konvenciója: kezdőérték: *Vált*0; közbülső értékek: *Vált*1, ..., *Vált*n; végérték: *Vált*.
- A Prolog akkumulátorpár nem más mint a funkcionális programozásból ismert gyűjtőargumentum és a függvény eredményének együttese.
- A DCG formalizmus – akkumulátorpárok automatikus „átszövése”:

```
sum3([X|L]) --> plus(X), sum3(L).
```


Akkumulátorok használata – folytatás

- Többszörös akkumulálás – lista összege és négyzetösszege

```
% sum2(+L, +S0, ?S, +Q0, ?Q): S-S0 =Σ L_i, Q-Q0 = Σ L_i^2
sum2([], S, S, Q, Q).
sum2([X|L], S0, S, Q0, Q):-
        S1 is S0+X, Q1 is Q0+X*X, sum2(L, S1, S, Q1, Q).
```

- Többszörös akkumulátorok összevonása egyetlen állapotváltozóvá

```
% sum3(+L, +S0/Q0, ?S/Q): S-S0 =Σ L_i, Q-Q0 = Σ L_i^2
sum3([]) --> [].                        % DCG "tényállítás"
sum3([X|L]) -->
        plus3(X), sum3(L).

plus3(X, S0/Q0, S/Q) :- S is S0+X, Q is Q0+X*X.

| ?- listing(sum3).

sum3([], A, B) :- B=A.
sum3([X|L], A, C) :-
        plus3(X, A, B), sum3(L, B, C).
```

Különbséglisták

- A `revapp` mint akkumuláló eljárás

```
% revapp(Xs, L0, L): Xs megfordítását L0 elé füzve kapjuk L-t.
% Másképpen: Xs megfordítása L-L0.
revapp([], L, L).
revapp([X|Xs], L0, L) :-
        L1 = [X|L0], revapp(Xs, L1, L).
```

- Az `L-L0` jelölés (különbséglista): az a lista, amelyet úgy kapunk, hogy `L` végéről elhagyjuk `L0`-t (ez feltételezi, hogy `L0` szuffixuma `L`-nek).
- Például az `[1,2,3]` listának megfelelő különbséglisták:
  - `[1,2,3,4]-[4]`, `[1,2,3,a,b]-[a,b]`, `[1,2,3]-[]`, ...
  - A legáltalánosabb (nyílt) különbséglistában a „kivonandó” változó: `[1,2,3|L]-L`
- Egy nyílt különbséglista konstans időben összefűzhető egy másikkal:

```
% app_dl(DL1, DL2, DL3): DL1 és DL2 különbséglisták összefűzése DL3.
app_dl(L-L0, L0-L1, L-L1).
| ?- app_dl([1,2,3|L0]-L0, [4,5|L1]-L1, DL).
        ⟹ DL = [1,2,3,4,5|L1]-L1, L0 = [4,5|L1]
```

- A nyílt különbséglista „egyszer használatos”, egy hozzáfűzés után már nem lesz nyílt!

Különbséglisták (folyt.)

- Példa: lineáris idejű listafordítás, `nrev` stílusában, különbséglistával:

```
% nrev(L, DR): Az L lista megfordítása a DR különbséglista.
nrev_dl([], L-L).                % L-L ≡ üres különbséglista
nrev_dl([X|L], DR) :-
    nrev_dl(L, DR0),
    app_dl(DR0, [X|T]-T, DR).    % [X|T]-T ≡ egyelemű különbséglista
% app_dl(DL1, DL2, DL3): DL1 és DL2 különbséglisták összefűzése DL3.
app_dl(L-L0, L0-L1, L-L1).
% Az L lista megfordítása R
rev(L, R) :- nrev_dl(L, R-[]).
```

- Az `nrev_dl/2` eljárás törzsében érdemes a két hívást megcserélni (jobbrekurzió!).
- `nrev_dl(L, R-R0)` ⟹ `rev2(L, R0, R)` átalakítással és `app_dl` kiküszöbölésével a fenti `nrev_dl/2` eljárásból kapunk egy `rev2/3`-t, amely azonos `revapp/3`-mal!
- Ettől az átalakítástól kb 3-szor gyorsabb lesz a program ⇒ érdemes a különbséglisták helyett akumulátorpárokat használni!
- A továbbiakban a különbséglista jelölést csak a fejkommentek megfogalmazásában használjuk.

Az `append` mint akkumuláló eljárás

- Írjunk egy `eleje_marad(Eleje, L, Marad)` eljárást!

```
% eleje_marad(Eleje, L, Marad): Az L lista prefixuma az Eleje lista,
% ennek L-ből való elhagyása után marad a Marad lista.
eleje_marad([], L, L).
eleje_marad([X|Xs], L0, L) :-
        L0 = [X|L1], eleje_marad(Xs, L1, L).
```

- Az akkumulálási lépés: `L0 = [X|L1]`, egy elem elhagyása a lista elejéről.
- A 2. és 3. argumentum felcserélésével az `eleje_marad` eljárás átalakul az `append` eljárássá!
- Tehát az `append` is tekinthető akkumuláló eljárásnak (a 2. és 3. argumentum a szokásos akkumulátorpárokhoz képest fel van cserélve):

```
% append(Xs, L, L0): L0 elejéről Xs elemeit lehagyva marad L.
% Másképpen: Xs = L0-L.
append([], L, L).
append([X|Xs], L, L0) :-
        L0 = [X|L1], append(Xs, L, L1).
```

- Az akkumulálási lépés: az `L0` változó értékül kap egy listát, melynek farka `L1`, az akkumulálált mennyiség: az a változó, amelyben az összefűzés eredményét várjuk.

Tartalom

5 Haladó Prolog
- Megoldásgyűjtő beépített eljárások
- Meta-logikai eljárások
- A keresési tér szűkítése
- Vezérlési eljárások
- A Prolog megvalósítási módszereiről
- Determinizmus és indexelés
- Jobbrekurzió és akkumulátorok
- Kényelmi eszközök: Definite Clause Grammars (DCG), ciklusok
- Listák és fák akkumulálása – példák
- Imperatív programok átírása Prologba
- Modularitás
- Magasabbrendű eljárások
- Dinamikus adatbáziskezelés
- Egy összetettebb példaprogram
- „Hagyományos” beépített eljárások
- Fejlettebb nyelvi és rendszerelemek

A DCG (Definite Clause Grammars) formalizmus

- DCG: előfeldolgozó eszköz nyelvtani elemzők írásához.
- DCG szabály: *Fej*`-->`*Törzs*. $\Longrightarrow$ *Fej*(A_0, A_m)`:-`*Törzs*(A_0, A_m)`.` A törzsben:
  - `{Cél}` $\Longrightarrow$ `Cél` (akkumulálást nem végző cél)
  - `[`E_1`,`E_2`,...,`E_k`]`, $k \geq 0$ $\Longrightarrow$ A_n `= [`E_1`,`E_2`,...,`E_k`|`A_{n+1}`]` (elemek akk.-a)
  - `p(`X_1`,`X_2`,...,`X_j`)`, $l \geq 0$ $\Longrightarrow$ `p(`X_1`,`X_2`,...,`X_j`,`A_n`,`A_{n+1}`)` (akk.-t végző cél)
  - Vezérlés: konj. (`,`), diszj. (`;`), ha-akkor (`->`), vágó (`!`), negáció (`\+`)
- Példa: egy lista pozitív elemeinek kigyűjtése

```
% pe(L, Pk0, Pk): Az L számlista pozitív elemeinek listája Pk0-Pk.
% Másszóval: L pozitív elemeinek listáját Pk elé füzve kapjuk Pk0-t
pe([], Pk0, Pk) :-     Pk0 = Pk.
pe([X|L], Pk0, Pk) :-  (   X > 0 -> Pk0 = [X|Pk1], pe(L, Pk1, Pk)
                       ;   pe(L, Pk0, Pk)
                       ).
```

- A DCG jelölést használó, a fentivel azonos kódot eredményező eljárás:

```
pe2([]) -->            [].
pe2([X|L]) -->         (   {X > 0} -> [X], pe2(L)
                       ;   pe2(L)
                       ).
```


A DCG formalizmus használata nyelvtani elemzésre

- Példa – decimális számok elemzését végző `szám(L0, L)` Prolog eljárás
- Az L0, L paraméterek: karakterkódok listái

```
% szám(L0, L): Az L0-L különbséglista számjegykódok nem-üres listája
% Másszóval: L0 elejéről leelemezhető egy szám, és marad L
szám --> számjegy, számmaradék.

% számmaradék(L0, L): Az L0-L különbséglista számjegykódok listája
számmaradék --> számjegy, számmaradék ; "".          % "" ≡ []

% számjegy(L0, L): L0 - [K|L], ahol K egy számjegy kódja
számjegy --> "0";"1";"2";"3";"4";"5";"6";"7";"8";"9".  % "9" ≡ [0'9]
```

- A `számjegy/2` eljárás egy másik megvalósítása

```
számjegy --> [K], {decimális_jegy_kódja(K)}.

% K egy számjegy kódja.
decimális_jegy_kódja(K) :- K >= 0'0, K =< 0'9.
```

- A fenti DCG szabály Prolog megfelelője:

```
számjegy(L0, L) :-
        L0 = [K|L],                 % K a következő listaelem
        decimális_jegy_kódja(K).    % megfelelő-e a K?
```


DCG nyelvtani elemzés – további részletek

- Az elemzés – a Prolog végrehajtás miatt – nem-determinisztikus, pl.

```
| ?- szám("123 abc", L).
L = " abc" ? ;          % leelemeztük a 123 számot
L = "3 abc" ? ;         % leelemeztük a 23 számot
L = "23 abc" ? ;        % leelemeztük a 3 számot
no
```

- A számmaradék eljárás determinisztikus változata

```
% számmaradék2(L0, L): L0-L számjegykódok maximális listája
számmaradék2 --> (   számjegy -> számmaradék2
                 ;   ""
                 ).
```

  vagy

```
számmaradék3 --> számjegy, !, számmaradék3. % A vágó köré nem kell {}
számmaradék3 --> "".
```

- Futás:

```
| ?- szám2("123 abc", L).
L = " abc" ? ;          % leelemeztük a (lehető leghosszabb) 123 számot
no
```


Az elemző kiegészítése jelentéshordozó argumentumokkal

- Egy DCG szabály az elemzéssel párhuzamosan további (kimenő) argumentum(ok)ban felépítheti a kielemzett dolog „jelentését”
- Példa: szám elemzése és értékének kiszámítása:

```
% Leelemezhető egy Sz értékű nem-üres számjegysorozat
szám(Sz) --> számjegy(J), számmaradék(J, Sz).

% Leelemezhető számjegyek egy esetleg üres listája, amelynek
% az eddig leelemzett Sz0-val együtt vett értéke Sz.
számmaradék(Sz0, Sz) -->
    számjegy(J), !, {Sz1 is Sz0*10+J}, számmaradék(Sz1, Sz).
számmaradék(Sz0, Sz0) --> [].

% leelemezhető egy J értékű számjegy.
számjegy(J) --> [K], {decimális_jegy_kódja(K), J is K-0'0}.
```

  `| ?- szám(Sz, "102 56", L).` $\Longrightarrow$ `L = " 56", Sz = 102; no`

- A `számmaradék` DCG szabály Prolog alakja:

```
számmaradék(Sz0, Sz, L0,L) :-
    számjegy(J, L0,L1), !, Sz1 is Sz0*10+J, számmaradék(Sz1, Sz, L1,L).
számmaradék(Sz0, Sz0, L0,L) :- L=L0.
```

- Itt két akkumulátorpár van: egy „kézi” (`Sz`) és egy DCG-ből generált (`L`).

Aritmetikai kifejezések elemzése

- Egyszerű aritmetikai kifejezések elemzése és kiértékelése.

```
% kif0(Z, L0, L): L0-L egy Z aritmetikai kifejezéssé elemezhető ki
kif0(X+Y) --> tag0(X), "+", !, kif0(Y).
kif0(X-Y) --> tag0(X), "-", !, kif0(Y).
kif0(X) -->   tag0(X).

tag0(X) --> szám(X).            % egyelőre
```

`| ?- kif0(Z, "4-2+1", []).` $\Longrightarrow$ `Z = 4-(2+1)` Jobbról balra elemez!

- Egy lehetséges javítás

```
kif(Z) --> tag(X), kifmaradék(X, Z).

kifmaradék(X, Z) --> "+", tag(Y), !, kifmaradék(X+Y, Z).
kifmaradék(X, Z) --> "-", tag(Y), !, kifmaradék(X-Y, Z).
kifmaradék(X, X) --> [].

tag(Z) --> szám(X), tagmaradék(X, Z).

tagmaradék(X, Z) --> "*", szám(Y), !, tagmaradék(X*Y, Z).
tagmaradék(X, Z) --> "/", szám(Y), !, tagmaradék(X/Y, Z).
tagmaradék(X, X) --> [].
```

`| ?- kif(Z, "5*4-2+1", []), Val is Z.` $\Longrightarrow$ `Z = 5*4-2+1, Val = 19 ? ; no`

Do-ciklusok (do-loops)

- Szintaxis: (*Iterátor*$_1$, ..., *Iterátor*$_m$ do *Célsorozat*)
  - Az `L` lista minden elemét megnövelve 1-gyel kapjuk az `NL` listát:

```
novel(L, NL) :-
     (   foreach(X, L), foreach(Y, NL)
     do  Y is X+1
     ).
```

  - Az `L` lista minden elemét megszorozva N-nel kapjuk az `NL` listát:

```
szoroz(L, N, NL) :-
     (   foreach(X, L), foreach(Y, ML), param(N)
     do  Y is N*X
     ).
```


Do-ciklusok: további iterátorok

- Példák:

```
| ?- (   for(I,1,5), foreach(I,List)
     do  true
     ).
                          List = [1,2,3,4,5] ? ; no

| ?- (   foreach(X,[1,2,3]), fromto(0,In,Out,Sum)
     do  Out is In+X
     ).
                          Sum = 6 ? ; no

| ?- (   foreach(X,[a,b,c,d,e]), count(I,1,N), foreach(I-X,Pairs)
     do  true
     ).
                          N = 5, Pairs = [1-a,2-b,3-c,4-d,5-e] ? ; no

| ?- (   foreacharg(A,f(a,b,c,d,e),I), foreach(I-A,List)
     do  true
     ).
                          List = [1-a,2-b,3-c,4-d,5-e] ? ; no
```


Tartalom

5 Haladó Prolog
- Megoldásgyűjtő beépített eljárások
- Meta-logikai eljárások
- A keresési tér szűkítése
- Vezérlési eljárások
- A Prolog megvalósítási módszereiről
- Determinizmus és indexelés
- Jobbrekurzió és akkumulátorok
- Kényelmi eszközök: Definite Clause Grammars (DCG), ciklusok
- Listák és fák akkumulálása – példák
- Imperatív programok átírása Prologba
- Modularitás
- Magasabbrendű eljárások
- Dinamikus adatbáziskezelés
- Egy összetettebb példaprogram
- „Hagyományos” beépített eljárások
- Fejlettebb nyelvi és rendszerelemek

Egy mintafeladat: $a^n b^n$ alakú sorozat előállítása

- Első megoldás, $3n$ lépés

```
% anbn(N, L): Az L lista N db a-ból
% és azt követő N db b-ből áll.
anbn(N, L) :-
        an(N, a, AN),
        an(N, b, BN),
        append(AN, BN, L).

% an(N, A, L): L az A elemet N-szer
% tartalmazó lista
an(0, _A, L) :- !, L = [].
an(N, A, [A|L]) :-
        N > 0,
        N1 is N-1,
        an(N1, A, L).
```

- Második megoldás, $2n$ lépés

```
anbn(N, L) :-
        an(N, b, [], BN),
        an(N, a, BN, L).

% an(N, A, L0, L): L-L0 az A
% elemet N-szer tartalmazó lista
an(0, _A, L0, L) :- !, L = L0.
an(N, A, L0, [A|L]) :-
        N > 0,
        N1 is N-1,
        an(N1, A, L0, L).
```


$a^n b^n$ alakú sorozatok (folyt.)

- Harmadik megoldás, n lépés

```
anbn(N, L) :-
        anbn(N, [], L).

% anbn(N, L0, L): Az L-L0 lista N db a-ból és azt követő N db b-ből áll.
anbn(0, L0, L) :- !, L = L0.
anbn(N, L0, [a|L]) :-
        N > 0,
        N1 is N-1,
        anbn(N1, [b|L0], L).
```

- A második klóz nem jobbrekurzív változata

```
anbn(N, L0, L) :-
        N > 0, N1 is N-1,
        L1 = [b|L0],       % 1. lépés: L0 elé b => L1
        anbn(N1, L1, L2), % 2. lépés: L1 elé a^N1 b^N1 => L2
        L = [a|L2].        % 3. lépés: L2 elé a => L
```


$a^n b^n$ alakú sorozatok – C++ megoldás

- C++ megoldás

```
link *anbn(unsigned n) {
  link *l = 0, *b = 0;    // ez elé építjük a b-ket
  link **a = &l;          // ebbe tesszük az a-kat
  for (; n > 0; --n) {
    *a = new link('a');   // elölről
    a = &(*a)->next;      // hátra épít
    b = new link('b', b); // hátulról előre épít
  }
  *a = b; return l;
}
```


Összetettebb adatstruktúrák akkumulálása

- Az adatstruktúra:
  `% :- type bfa -> ures ; bfa(int, bfa, bfa).`
- A fa csomópontjaiban tároljuk a számértékeket, a levelek nem tárolnak információt.
- Egészek gyűjtése rendezett bináris fában
  - `beszur(BFa0, E, BFa)`: Az `E` egész számnak a `BFa0` fába való beszúrása a `BFa` bináris fát eredményezi.
  - Itt `BFa0` és `BFa` egy akkumulátorpár, de az indexelés érdekében `BFa0` az első argumentum-pozícióba kerül.
- Példafutás:

```
| ?- beszur(ures, 3, Fa0),
     beszur(Fa0, 1, Fa1),
     beszur(Fa1, 5, Fa2).

Fa0 = bfa(3,ures,ures),
Fa1 = bfa(3,bfa(1,ures,ures),ures),
Fa2 = bfa(3,bfa(1,ures,ures),bfa(5,ures,ures)) ?
```

Akkumulálás bináris fákkal

- Elem beszúrása bináris fába

```
% beszur(BF0, E, BF): E beszúrása BF0 rendezett fába
% a BF rendezett fát adja
% :- pred beszur(bfa::in, int::in, bfa::out).
beszur(ures, Elem, bfa(Elem, ures, ures)).
beszur(BF0, Elem, BF):-
        BF0 = bfa(E,B,J),  % az indexelés működik!
        (   Elem =:= E -> BF = BF0
        ;   Elem < E ->
            BF = bfa(E,B1,J),
            beszur(B, Elem, B1)
        ;   BF = bfa(E,B,J1),
            beszur(J, Elem, J1)
        ).
```


Akkumulálás bináris fákkal – folyt.

- Lista konverziója bináris fává

```
% lista_bfa(L, BF0, BF): L elemeit beszúrva BF0-ba kapjuk BF-t.
% :- pred lista_bfa(list(int)::in, bfa::in, bfa::out).
lista_bfa([], BF, BF).
lista_bfa([E|L], BF0, BF):-
        beszur(BF0, E, BF1),
        lista_bfa(L, BF1, BF).

| ?- lista_bfa([3,1,5], ures, BF).
BF = bfa(3,bfa(1,ures,ures),bfa(5,ures,ures)) ? ;
no

| ?- lista_bfa([3,1,5,1,2,4], ures, BF).
BF = bfa(3,bfa(1,ures,bfa(2,ures,ures)),
           bfa(5,bfa(4,ures,ures),ures)) ? ;
no
```


Akkumulálás bináris fákkal – folyt.

- Bináris fa konverziója listává

```
% bfa_lista(BF, L0, L): A BF fa levelei az L-L0 listát adják.
% :- pred bfa_lista(bfa::in, list(int)::in,
%                   list(int)::out).
bfa_lista(ures, L, L).
bfa_lista(bfa(E, B, J), L0, L) :-
        bfa_lista(J, L0, L1),
        bfa_lista(B, [E|L1], L).
```

- Rendezés bináris fával

```
% L lista rendezettje R.
% :- pred rendez(list(int)::in, list(int)::out).
rendez(L, R):-
        lista_bfa(L, ures, BF), bfa_lista(BF, [], R).

| ?- rendez([1,5,3,1,2,4], R).
R = [1,2,3,4,5] ? ;
no
```


Tartalom

5 Haladó Prolog
- Megoldásgyűjtő beépített eljárások
- Meta-logikai eljárások
- A keresési tér szűkítése
- Vezérlési eljárások
- A Prolog megvalósítási módszereiről
- Determinizmus és indexelés
- Jobbrekurzió és akkumulátorok
- Kényelmi eszközök: Definite Clause Grammars (DCG), ciklusok
- Listák és fák akkumulálása – példák
- Imperatív programok átírása Prologba
- Modularitás
- Magasabbrendű eljárások
- Dinamikus adatbáziskezelés
- Egy összetettebb példaprogram
- „Hagyományos” beépített eljárások
- Fejlettebb nyelvi és rendszerelemek

Hogyan írjunk át imperatív nyelvű algoritmust Prolog programmá?

- Példafeladat: Hatékony hatványozási algoritmus
  - Alaplépés: a kitevő felezése, az alap négyzetre emelése.
  - Lényegében a kitevő kettes számrendszerbeli alakja szerint hatványoz.
- Az algoritmust megvalósító C nyelvű függvény:

```
/* hatv(a, h) = a**h */
int hatv(int a, unsigned h)
{
  int e = 1;
  while (h > 0)
  {
    if (h & 1) e *= a;
    h >>= 1; a *= a;
  }
  return e;
}
```

- Az algoritmusban három változó van: `a, h, e`:
  - `a` és `h` végértékére nincs szükség,
  - `e` végső értéke szükséges (ez a függvény eredménye).

A `hatv` C függvénynek megfelelő Prolog eljárás

- Kétargumentumú C függvény $\Longrightarrow$ 2+1-argumentumú Prolog eljárás.
- A függvény eredménye $\Longrightarrow$ utolsó arg.: `hatv(+A, +H, ?E)`: $A^H = E$.
- Ciklus $\Longrightarrow$ segédeljárás: `hatv(+A0, +H0, +E0, ?E)`: $A0^{H0} * E0 = E$.
- »a« és »h« C változók $\Longrightarrow$ »+A0« és »+H0« bemenő *paraméterek* (nem kell végérték),
  »e« C változó $\Longrightarrow$ »+E0, ?E« *akkumulátorpár* (kezdőérték, végérték).

```
hatv(A, H, E) :-
    hatv(A, H, 1, E).

hatv(A0, H0, E0, E) :- H0 > 0, !,
    (   H0 /\ 1 =:= 1
        % /\ ≡ bitenkénti "és"
    ->  E1 is E0*A0
    ;   E1 = E0
    ),
    H1 is H0 >> 1,
    A1 is A0*A0,
    hatv(A1, H1, E1, E).
hatv(_, _, E, E).
```

```
int hatv(int a, unsigned h)
{       int e = 1;

  ism:  if (h > 0)
        {  if (h & 1)
             e *= a;




           h >>= 1;
           a *= a;
           goto ism;
        } else return e;
}
```


A C ciklus és a Prolog eljárás kapcsolata

- A ciklust megvalósító Prolog eljárás minden pontján minden C változónak megfeleltetethető egy Prolog változó (pl. `h`-nak `H0, H1, ...`):
  - A ciklusmag elején a C változók a megfelelő Prolog argumentumban levő változónak felelnek meg.
  - Egy C értékadásnak egy új Prolog változó bevezetése felel meg, az ez után következő kódban az új változó felel meg a C változónak.
  - Ha a diszjunkció egyik ága megváltoztat egy változót, akkor a többi ágon is be kell vezetni az új Prolog változót, a régivel azonos értékkel (ld. `if (h & 1) ...`).
- A C ciklusmag végén a Prolog eljárást vissza kell hívni, argumentumaiban az egyes C változóknak pillanatnyilag megfeleltetett Prolog változóval.
- A C ciklus ciklus-invariánsa nem más mint a Prolog eljárás fejkommentje, a példában:

  `% hatv(+A0, +H0, +E0, ?E)`: $A0^{H0} * E0 = E$.

Programhelyesség-bizonyítás

- Egy algoritmus (függvény) specifikációja:
  - előfeltételek: a bemenő paramétereknek teljesíteniük kell ezeket,
  - utófeltételek: a paraméterek és az eredmény kapcsolatát írják le.
- Egy algoritmus helyes, ha minden, az előfeltételeket kielégítő adatra a függvény hibátlanul lefut, és eredményére fennállnak az utófeltételek.
- Példa: `x = mfoku_gyok(a,b,c)`
  - előfeltételek: `b*b-4*a*c >= 0`, $a \neq 0$
  - utófeltétel: `a*x*x+b*x+c = 0`
  - a program:

```
double mfoku_gyok(a, b, c)
double a, b, c;
{  double d = sqrt(b*b-4*a*c);
   return (-b+d)/2/a;
}
```

- A program helyességének bizonyítása lineáris kódra viszonylag egyszerű.

Ciklikus programok helyességének bizonyítása

- A ciklusokat „fel kell vágni” egy ciklus-invariánssal, amely:
  - az előfeltételekből és a ciklust megelőző értékadásokból következik,
  - ha a ciklus elején fennáll, akkor a ciklus végén is (indukció),
  - belőle és a leállási feltételből következik a ciklus utófeltétele.

```
int hatv(int a0, unsigned h0) /*utófeltétel: hatv(a0, h0) = a0^h0 */
{ int e = 1, a = a0, h = h0;
  while /*ciklus-invariáns: a0^h0 == e*a^h */ (h > 0)
  {
    /* induláskor a kezdőértékek alapján triviálisan fennáll */
    if (h & 1) e *= a;          /* e' = e * a^(h&1) */
    h >>= 1;                    /* h' = (h-(h&1))/2 */
    a *= a;                     /* a' = a*a */
  }                             /*indukció: e'*a'^h' = ... = e*a^h */
  return e;
  /* Az invariánsból h = 0 miatt következik az utófeltétel */
}
```


Tartalom

5 Haladó Prolog
- Megoldásgyűjtő beépített eljárások
- Meta-logikai eljárások
- A keresési tér szűkítése
- Vezérlési eljárások
- A Prolog megvalósítási módszereiről
- Determinizmus és indexelés
- Jobbrekurzió és akkumulátorok
- Kényelmi eszközök: Definite Clause Grammars (DCG), ciklusok
- Listák és fák akkumulálása – példák
- Imperatív programok átírása Prologba
- Modularitás
- Magasabbrendű eljárások
- Dinamikus adatbáziskezelés
- Egy összetettebb példaprogram
- „Hagyományos” beépített eljárások
- Fejlettebb nyelvi és rendszerelemek

Modulok definiálása SICStus Prolog nyelven

- Minden modul külön állományba kell kerüljön.
- Az állomány első programeleme egy modul-parancs kell legyen:

  `:- module(` *Modulnév*`, [`*ExpFunktor1*`,` *ExpFunktor2*`, ...]).`
- *ExpFunktor* = az exportálandó eljárás funktora (név/arg.szám)
- Példa:

```
:- module(platók, [fennsík/3]).      % plato állomány első sora
```

- Modul-betöltésre szolgáló beépített eljárások:
  - `use_module(`*ÁllományNév*`)`
  - `use_module(`*ÁllományNév*`, [`*ImpFunktor1*`,`*ImpFunktor2*`,...])`
    ImpFunktor – az importálandó eljárás funktora
  - *ÁllományNév* lehet névkonstans, vagy pl. `library(`*KönyvtárNév*`)`:

```
:- use_module(plato).                    % a fenti modul betöltése
:- use_module(library(lists), [last/2]). % csak last/2 importált
```

- Modulkvalifikált hívási forma: *Modul*`:`*Hívás* a *Modul*-ban futtatja *Hívás*-t.
- A modulfogalom nem szigorú, egy nem exportált eljárás is meghívató modulkvalifikált formában, pl. `platók:első_fennsík(...)`.

Meta-eljárások modularizált programban

- Meta-paraméterek átadása modulközi hívásokban:

`modul1.pl` állomány:

```
:- module(modul1, [kétszer/1]).

%:- meta_predicate kétszer(:).  (*)
kétszer(X) :-
        X, X.

p :- write(bu).
```

`modul2.pl` állomány:

```
:- module(modul2, [q/0,r/0]).

:- use_module(modul1).

q :- kétszer(p).

r :- kétszer(modul2:p).

p :- write(ba).
```

- Futtatás:

```
| ?- [modul1,modul2].
| ?- q.   ⟹   bubu
| ?- r.   ⟹   baba
```

- Automatikus modul-kvalifikáció meta-predikátum deklarációval:
  Ha `modul1.pl`-ben elhagyjuk a `(*)`-gal jelzett sor előtti % kommentjelet, akkor

```
| ?- q.   ⟹   baba!
```

Meta-predikátum deklaráció, modulnév-kiterjesztés

- Meta-predikátum deklaráció
  - Formája:
    `:- meta_predicate` ⟨`eljárásnév`⟩(⟨`módspec`$_1$⟩, ..., ⟨`módspec`$_n$⟩), ...
  - ⟨`módspec`$_i$⟩ lehet ':', '+', '-', vagy '?'.
  - A ':' mód azt jelzi, hogy az adott argumentumot betöltéskor ún. modulnév-kiterjesztésnek kell alávetni. (A többi mód hatása azonos, be/kimenő irányt jelezhetünk segítségükkel.)
- Egy *Kif* kifejezés modulnév-kiterjesztése a következő átalakítást jelenti:
  - ha *Kif* `M:X` alakú, vagy olyan változó, amely az adott eljárás fejében meta-argumentum pozíción van, akkor változatlanul hagyjuk;
  - egyébként helyettesítjük *CurMod*`:`*Kif*-fel, ahol *CurMod* a kurrens modul.
- Példa folyt. (tfh. a `modul1`-beli `kétszer` meta-predikátumnak deklarált!)

```
:- module(modul2, [négyszer/1,q/0]).
:- use_module(modul1).                    % a tárolt alak:
q :- kétszer(p).                    ⟹    q :- kétszer(modul2:p).

:- meta_predicate négyszer(:).
négyszer(X) :- kétszer(X), kétszer(X). ⟹   változatlan
```


Tartalom

5 Haladó Prolog
- Megoldásgyűjtő beépített eljárások
- Meta-logikai eljárások
- A keresési tér szűkítése
- Vezérlési eljárások
- A Prolog megvalósítási módszereiről
- Determinizmus és indexelés
- Jobbrekurzió és akkumulátorok
- Kényelmi eszközök: Definite Clause Grammars (DCG), ciklusok
- Listák és fák akkumulálása – példák
- Imperatív programok átírása Prologba
- Modularitás
- Magasabbrendű eljárások
- Dinamikus adatbáziskezelés
- Egy összetettebb példaprogram
- „Hagyományos" beépített eljárások
- Fejlettebb nyelvi és rendszerelemek

Magasabbrendű eljárások – listakezelés

- Magasabbrendű (vagy meta-eljárás) egy eljárás,
  - ha eljárásként értelmezi egy vagy több argumentumát
  - pl. `call/1, findall/3, \+ /1 stb.`
- Listafeldolgozás `findall` segítségével – példák
  - Páros elemek kiválasztása (vö. Erlang filter)

```
% Az L egész-lista páros elemeinek listája Pk.
páros_elemei(L, Pk) :-
    findall(X, (member(X, L), X mod 2 =:= 0), Pk).

| ?- páros_elemei([1,2,3,4], Pk). ⟹   Pk = [2,4]
```

  - A listaelemek négyzetre emelése (vö. Erlang map)

```
% Az L számlista  elemei négyzeteinek listája Nk.
négyzetei(L, Nk) :-
    findall(Y, (member(X, L), négyzete(X, Y)), Nk).

négyzete(X, Y) :- Y is X*X.

| ?- négyzetei([1,2,3,4], Nk).    ⟹   Nk = [1,4,9,16]
```


Részlegesen paraméterezett eljáráshívások – segédeszközök

- A `négyzete/0` kifejezés a `négyzete/2` részlegesen paraméterezett hívásának tekinthető.
- Ilyen hívások kiegészítésére és meghívására szolgálnak a `call/N` eljárások.
- `call(RPred, A1, A2, ...)` végrehajtása: az `RPred` részleges hívást kiegészíti az `A1, A2, ...` argumentumokkal, és meghívja.
- A `call/N` eljárások SICStus 4-ben már beépítettek, SICStus 3-ban még definiálni kellett ezeket, pl. így:

```
:- meta_predicate call(:, ?), call(:, ?, ?), ....

% Pred az A utolsó argumentummal meghívva igaz.
call(M:Pred, A) :-
        Pred =.. FAs0, append(FAs0, [A], FAs1),
        Pred1 =.. FAs1, call(M:Pred1).

% Pred az A és B utolsó argumentumokkal meghívva igaz.
call(M:Pred, A, B) :-
        Pred =.. FAs0, append(FAs0, [A,B], FAs2),
        Pred2 =.. FAs2, call(M:Pred2).

...
```

Részlegesen paraméterezett eljárások – rekurzív `map/3`

- Részleges paraméterezéssel a `map/3` meta-eljárás rekurzívan definiálható:

```
% map(Xs, Pred, Ys): Az Xs lista elemeire a Pred transzformációt
% alkalmazva kapjuk az Ys listát.
map([X|Xs], Pred, [Y|Ys]) :-
        call(Pred, X, Y), map(Xs, Pred, Ys).
map([], _, []).

másodfokú_képe(P, Q, X, Y) :- Y is X*X + P*X + Q.
```

- Példák:

```
| ?- map([1,2,3,4], négyzete, L).            ⟹   L = [1,4,9,16]
| ?- map([1,2,3,4], másodfokú_képe(2,1), L). ⟹   L = [4,9,16,25]
```

- A `call/N`-re épülő megoldás előnyei:
  - általánosabb és hatékonyabb lehet, mint a `findall`-ra épülő;
  - alkalmazható akkor is, ha az elemekre elvégzendő műveletek nem függetlenek, pl. `foldl`.

Rekurzív meta-eljárások – `foldl` és `foldr`

-
```
% foldl(+Xs, :Pred, +Y0, -Y): Y0-ból indulva, az Xs elemeire
% balról jobbra sorra alkalmazva a Pred által leírt
% kétargumentumú függvényt kapjuk Y-t.
foldl([X|Xs], Pred, Y0, Y) :-
        call(Pred, X, Y0, Y1), foldl(Xs, Pred, Y1, Y).
foldl([], _, Y, Y).

jegyhozzá(Alap, Jegy, Szam0, Szam) :- Szam is Szam0*Alap+Jegy.

| ?- foldl([1,2,3], jegyhozzá(10), 0, E).  ⟹   E = 123
```

-
```
% foldr(+Xs, :Pred, +Y0, -Y): Y0-ból indulva, az Xs elemeire jobbról
% balra sorra alkalmazva a Pred kétargumentumú függvényt kapjuk Y-t.
foldr([X|Xs], Pred, Y0, Y) :-
        foldr(Xs, Pred, Y0, Y1), call(Pred, X, Y1, Y).
foldr([], _, Y, Y).

| ?- foldr([1,2,3], jegyhozzá(10), 0, E).  ⟹   E = 321
```


Tartalom

5 Haladó Prolog
- Megoldásgyűjtő beépített eljárások
- Meta-logikai eljárások
- A keresési tér szűkítése
- Vezérlési eljárások
- A Prolog megvalósítási módszereiről
- Determinizmus és indexelés
- Jobbrekurzió és akkumulátorok
- Kényelmi eszközök: Definite Clause Grammars (DCG), ciklusok
- Listák és fák akkumulálása – példák
- Imperatív programok átírása Prologba
- Modularitás
- Magasabbrendű eljárások
- Dinamikus adatbáziskezelés
- Egy összetettebb példaprogram
- „Hagyományos” beépített eljárások
- Fejlettebb nyelvi és rendszerelemek

Dinamikus predikátumok

- A dinamikus predikátum jellemzői:
  - a program szövegében lehet 0 vagy több klóza;
  - futási időben hozzáadhatunk és elvehetünk klózokat belőle;
  - végrehajtása mindenképpen interpretált.
- Létrehozása
  - programszövegbeli deklarációval:

    `:- dynamic(Eljárásnév/Argumentumszám).`

    (ha van klóza a programban, akkor az első előtt – ilyenkor kötelező);
  - futási időben, adatbáziskezelő beépített eljárással
- Adatbáziskezelő eljárások („adatbázis” = a program klózainak összessége):
  - klóz felvétele első, utolsó helyre: `asserta/1, assertz/1`
  - klóz törlése (illesztéssel, többszörösen sikerülhet): `retract/1`
  - klóz lekérdezése (illesztéssel, többszörösen sikerülhet): `clause/2`
- A klózfelvétel ill. törlés tartós mellékhatás, visszalépéskor nem áll vissza a korábbi állapot.

Klóz felvétele: `asserta/1, assertz/1`

- `asserta(:@Klóz)`
  - A `Klóz` kifejezést klózként értelmezve felveszi a programba az adott predikátum *első* klózaként. A `Klóz`ban levő változók szisztematikusan újakra cserélődnek.
  - A '`@`' mód jelentése: tisztán bemenő paraméter, az eljárás a paraméterbeli változókat nem helyettesíti be (a '`+`' mód speciális esete).
  - A '`:`' mód modul-kvalifikált paramétert jelez.
- `assertz(:@Klóz)`
  - Ugyanaz mint `asserta`, csak a `Klóz` kifejezést az adott predikátum *utolsó* klózaként veszi fel.
- Példa:

```
| ?- assertz((p(1,X):-q(X))), asserta(p(2,0)),
     assertz((p(2,Z):-r(Z))), listing(p).   ⟹   p(2, 0).
                                                p(1, A) :-   q(A).
                                                p(2, A) :-   r(A).

| ?- assert(s(X,X)), s(U,V), U == V, X \== U.
⟹  V = U ? ; no
```


Klóz törlése: `retract/1`

- `retract(:@Klóz)`
  - A `Klóz` klóz-kifejezésből megállapítja a predikátum funktorát.
  - Az adott predikátum klózait sorra megpróbálja illeszteni `Klóz`-zal.
  - Ha az illesztés sikerült, akkor kitörli a klózt és sikeresen lefut.
  - Visszalépés esetén folytatja a keresést (illeszt, töröl, sikerül stb.)
- Példa (folytatás):

```
| ?- listing(p), Cl = (p(2,_):-_),
     retract(Cl), format('Del: ~w.\n', [Cl]), listing(p), fail.
```

- A futás kimenete:

```
p(2, 0).
p(1, A) :-
        q(A).
p(2, A) :-
        r(A).
```

```
Del: p(2,0):-true.
p(1, A) :-
          q(A).
p(2, A) :-
          r(A).
```

```
Del: p(2,_537):-r(_537).
p(1, A) :-
        q(A).
```

⟹ no

Alkalmazási példa – egyszerűsített `findall`

- A `findall1/3` eljárás hatása megegyezik a beépített `findall`-lal, de
  - nem jó, ha a `Cél`-ban újabb, skatulyázott `findall1` hívás van.

```
:- dynamic(megoldás/1).

% findall1(Minta, Cél, L): Cél összes megoldására Minták listája L.
findall1(Minta, Cél, _MegoldL) :-
    call(Cél),
    asserta(megoldás(Minta)), % fordított sorrendben vesszük fel!
    fail.
findall1(_Minta, _Cél, MegoldL) :-
    megoldás_lista([], MegoldL).

% A megoldás/1 tényállításokban tárolt kifejezések
% fordított listája L-L0.
megoldás_lista(L0, L) :-
    retract(megoldás(M)), !,
    megoldás_lista([M|L0], L).
megoldás_lista(L, L).

| ?- findall1(Y, (member(X,[1,2,3]),Y is X*X), ML). ⟹  ML = [1,4,9]
```


Klóz lekérdezése: `clause/2`

- `clause(:@Fej, ?Törzs)`
  - A `Fej` alapján megállapítja a predikátum funktorát.
  - Az adott predikátum klózait sorra megpróbálja illeszteni a `Fej :- Törzs` kifejezéssel (tényállítás esetén `Törzs = true`).
  - Ha az illesztés sikerült, akkor sikeresen lefut.
  - Visszalépés esetén folytatja a keresést (illeszt, sikerül stb.)
- Példa:

```
:- listing(p), clause(p(2, 0), T).
```

```
p(2, 0).
p(1, A) :-
        q(A).
p(2, A) :-
        r(A).
```

```
T = true ? ;
T = r(0) ? ;
no
```

A `clause` eljárás alkalmazása: egyszerű nyomkövető interpreter

- Az alábbi interpreter csak „tiszta”, beépített eljárást nem alkalmazó Prolog programok futtatására alkalmas.

```
% interp(G, D): A G cél futását D bekezdésű nyomkövetéssel mutatja.
interp(true, _) :- !.
interp((G1, G2), D) :- !,
    interp(G1, D), interp(G2, D).
interp(G, D) :-
    (   trace(G, D, call)
    ;   trace(G, D, fail), fail % követi a fail kaput, tovább-hiúsul
    ),
    D2 is D+2,
    clause(G, B), interp(B, D2),
    (   trace(G, D, exit)
    ;   trace(G, D, redo), fail % követi a redo kaput, tovább-hiúsul
    ).

% A G cél áthaladását a Port kapun D bekezdésű nyomkövetéssel mutatja.
trace(G, D, Port) :-
    /*D szóközt ír ki:*/ format('~|~t~*+', [D]),
    write(Port), write(': '), write(G), nl.
```


Nyomkövető interpreter - példafutás

```
:- dynamic app/3, app/4.  % (*)

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

app(L1, L2, L3, L123) :-
    app(L1, L23, L123),
    app(L2, L3, L23).
```

A (*) sor elhagyható, ha a fenti (mondjuk `app34`) állományt az alábbi (SICStus-specifikus) beépített eljárással töltjük be:

```
| ?- load_files(app34,
        compilation_mode(
                assert_all)).
```

```
| ?- interp(app(_,[b,c],L,[c,b,c,b]), 0).
  call: app(_203,[b,c],_253,[c,b,c,b])
    call: app(_203,_666,[c,b,c,b])
    exit: app([],[c,b,c,b],[c,b,c,b])
    call: app([b,c],_253,[c,b,c,b])
    fail: app([b,c],_253,[c,b,c,b])
    redo: app([],[c,b,c,b],[c,b,c,b])
      call: app(_873,_666,[b,c,b])
      exit: app([],[b,c,b],[b,c,b])
    exit: app([c],[b,c,b],[c,b,c,b])
    call: app([b,c],_253,[b,c,b])
      call: app([c],_253,[c,b])
        call: app([],_253,[b])
        exit: app([],[b],[b])
      exit: app([c],[b],[c,b])
    exit: app([b,c],[b],[b,c,b])
  exit: app([c],[b,c],[b],[c,b,c,b])
L = [b] ?
```


Tartalom

5 Haladó Prolog
- Megoldásgyűjtő beépített eljárások
- Meta-logikai eljárások
- A keresési tér szűkítése
- Vezérlési eljárások
- A Prolog megvalósítási módszereiről
- Determinizmus és indexelés
- Jobbrekurzió és akkumulátorok
- Kényelmi eszközök: Definite Clause Grammars (DCG), ciklusok
- Listák és fák akkumulálása – példák
- Imperatív programok átírása Prologba
- Modularitás
- Magasabbrendű eljárások
- Dinamikus adatbáziskezelés
- Egy összetettebb példaprogram
- „Hagyományos” beépített eljárások
- Fejlettebb nyelvi és rendszerelemek

Egy nagyobb DCG példa: „természetes” nyelvű beszélgetés

```
% mondat(Alany, Áll, L0, L): L0-L kielemezhető egy Alany alanyból és Áll
% állítmányból álló mondattá. Alany lehet első vagy második személyű
% névmás, vagy egyetlen szóból álló (harmadik személyű) alany.
mondat(Alany, Áll) -->
        {én_te(Alany, Ige)}, én_te_perm(Alany, Ige, Áll).
mondat(Alany, Áll) -->
        szó(Alany), szavak(Áll).

% én_te(Alany, Ige):
% Az Alany első/második személyű névmásnak megfelelő létige az Ige.
én_te("én", "vagyok").
én_te("te", "vagy").

% én_te_perm(Ki, Ige, Áll, L0, L): L0-L kielemezhető egy Ki
% névmásból, Ige igealakból és Áll állítmányból álló mondattá.
én_te_perm(Alany, Ige, Áll)  -->
        (   szó(Alany), szó(Ige), szavak(Áll)
        ;   szó(Alany), szavak(Áll), szó(Ige)
        ;   szavak(Áll), szó(Ige), szó(Alany)
        ;   szavak(Áll), szó(Ige)
        ).
```

Példa: „természetes” nyelvű beszélgetés – szavak elemzése

```
% szó(Sz, L0, L): L0-L egy Sz betűsorozatból álló (nem üres) szó.
szó(Sz) --> betű(B), szómaradék(SzM), {illik([B|SzM], Sz)}, köz.

% szómaradék(Sz, L0, L): L0-L egy Sz kódlistából álló (esetleg üres) szó.
szómaradék([B|Sz]) --> betű(B), !, szómaradék(Sz).
szómaradék([]) --> [].

% illik(Szó0, Szó): Szó0 = Szó, vagy a kezdő kis-nagy betűben különböznek.
illik([B0|L], [B|L]) :-
        (   B = B0 -> true
        ;   abs(B-B0) =:= 32
        ).

% köz(L0, L): L0-L nulla, egy vagy több szóköz.
köz --> (   " " -> köz ; ""   ).

% betű(K, L0, L): L0-L egy K kódú "betű" (különbözik a " .?" jelektől)
betű(K) --> [K], {\+ member(K, " .?")}.


% szavak(SzL, L0, L): L0-L egy SzL szó-lista.
szavak([Sz|Szk]) -->          szó(Sz), (   szavak(Szk)
                                       ;   {Szk = []}
                                       ).
```


Példa: „természetes” nyelvű beszélgetés – párbeszéd-szervezés

```
% :- type mondás --> kérdez(szó) ; kijelent(szó,list(szó)) ; un.

% Megvalósít egy párbeszédet.
párbeszéd :-
        repeat,
            read_line(L),   % beolvas egy sort, L a karakterkódok listája
            (   menet(Mondás, L, []) ->  feldolgoz(Mondás)
            ;   write('Nem értem\n'), fail
            ),
        Mondás = un, !.

% menet(Mondás, L0, L): Az L0-L kielemzett alakja Mondás.
menet(kérdez(Alany)) -->
        {kérdő(Szó)}, mondat(Alany, [Szó]), "?".
menet(kijelent(Alany,Áll)) -->
        mondat(Alany, Áll), ".".
menet(un) --> szó("unlak"), ".".

% kérdő(Szó): Szó egy kérdőszó.
kérdő("mi").
kérdő("ki").
kérdő("kicsoda").
```


Példa: „természetes” nyelvű beszélgetés – válaszok előállítása

```
:- dynamic tudom/2.

% feldolgoz(Mondás): feldolgozza a felhasználótól érkező Mondás üzenetet.
feldolgoz(un) :-
        write('Én is.\n').
feldolgoz(kijelent(Alany, Áll)) :-
        assertz(tudom(Alany,Áll)),
        write('Felfogtam.\n').
feldolgoz(kérdez(Alany)) :-
        tudom(Alany, _), !,
        válasz(Alany).
feldolgoz(kérdez(_)) :-
        write('Nem tudom.\n').

% Felsorolja az Alany ismert tulajdonságait.
válasz(Alany) :-
        tudom(Alany, Áll),
        (   member(Szó, Áll), format('~s ', [Szó]), fail
        ;   nl
        ), fail.
válasz(_).
```


Beszélgetős DCG példa – egy párbeszéd

```
| ?- párbeszéd.
|: Magyar legény vagyok én.
Felfogtam.
|: Ki vagyok én?
Magyar legény
|: Péter kicsoda?
Nem tudom.
|: Péter tanuló.
Felfogtam.
|: Péter jó tanuló.
Felfogtam.
|: Péter kicsoda?
tanuló
jó tanuló
|: Boldog vagyok.
Felfogtam.
|: Én vagyok Jeromos.
Felfogtam.
|: Te egy Prolog program vagy.
Felfogtam.
|: Ki vagyok én?
Magyar legény
Boldog
Jeromos
|: Okos vagy.
Felfogtam.
|: Ki vagy  te?
egy Prolog program
Okos
|: Valóban?
Nem értem
|: Unlak.
Én is.
```

Tartalom

5 Haladó Prolog
- Megoldásgyűjtő beépített eljárások
- Meta-logikai eljárások
- A keresési tér szűkítése
- Vezérlési eljárások
- A Prolog megvalósítási módszereiről
- Determinizmus és indexelés
- Jobbrekurzió és akkumulátorok
- Kényelmi eszközök: Definite Clause Grammars (DCG), ciklusok
- Listák és fák akkumulálása – példák
- Imperatív programok átírása Prologba
- Modularitás
- Magasabbrendű eljárások
- Dinamikus adatbáziskezelés
- Egy összetettebb példaprogram
- „Hagyományos” beépített eljárások
- Fejlettebb nyelvi és rendszerelemek

Aritmetikai beépített eljárások

- `X is Kif`: `Kif` aritmetikai kifejezés kell legyen, értékét egyesíti `X`-szel.
- `Kif1` ρ `Kif2`: `Kif1` és `Kif2` aritmetikai kifejezések kell legyenek, értékeik között elvégzi a ρ összehasonlítást (ρ lehet `=, =\=, <, =<, >, >=`).
- Aritmetikai kifejezésekben felhasználható funktorok:

Infix operátorok					
`+`	összeadás	`//`	egész osztás	`/\`	bitenkénti és
`-`	kivonás	`**`	hatványozás	`\/`	bitenkénti vagy
`*`	szorzás	`mod`	modulus képzés	`<<`	bitenkénti balra léptetés
`/`	osztás	`rem`	maradék képzés	`>>`	bitenkénti jobbra léptetés
Prefix operátorok:		`-`	negáció	`\`	bitenkénti negáció

Függvény jelölésűek			
`abs/1`	`exp/1`	`floor/1`	`sign/1`
`atan/1`	`float/1`	`log/1`	`sin/1`
`ceiling/1`	`float_fractional_part/1`	`max/2,min/2`	`sqrt/1`
`cos/1`	`float_integer_part/1`	`round/1`	`truncate/1`

Listakezelő beépített eljárások

- Lista hossza: `length(?L, ?N)`
  - Jelentése: az `L` lista hossza `N`.
  - `length(-L, +N)` módban adott hosszúságú, csupa különböző változóból álló listát hoz létre.
  - `length(-L, -N)` módban rendre felsorolja a 0, 1, ... hosszú listákat.
  - Megvalósítását lásd korábban.
- Lista rendezése: `sort(@L, ?S)`
  - Jelentése: az `L` lista `@<` szerinti rendezése `S`, (`==/2` szerint azonos elemek ismétlődését kiszűrve).
- Lista kulcs szerinti rendezése: `keysort(@L, ?S)`
  - Az `L` argumentum `Kulcs-Érték` alakú kifejezések listája.
  - Az eljárás jelentése: az `S` lista az `L` lista `Kulcs` értékei szerinti szabványos (`@<` általi) rendezése, ismétlődéseket nem szűr.

Kifejezések kiírása

- `write(@X)`: Kiírja `X`-et, ha szükséges operátorokat, zárójeleket használva.
- `writeq(@X)`: Mint `write(X)`, csak gondoskodik, hogy szükség esetén az névkonstansok idézőjelek közé legyenek téve.
- `write_canonical(@X)`: Mint `writeq(X)`, csak operátorok nélkül, minden struktúra szabványos alakban jelenik meg.
- `write_term(@X, +Opciók)`: Az `Opciók` opciólista szerint kiírja `X`-et.
- `format(@Formátum, @AdatLista)`: A `Formátum`-nak megfelelő módon kiírja `AdatLista`-t. A formázójelek alakja: `~`⟨`szám esetleg`⟩⟨`formázójel`⟩.

```
| ?- write('Helló világ').                          ⟹  Helló világ
| ?- writeq('Helló világ').                         ⟹  'Helló világ'
| ?- write_canonical('*' - '%').                    ⟹  -(*,'%')
| ?- write_canonical([1,2]).                        ⟹  '.'(1,'.'(2,[]))
| ?- write_term([1,2,3], [max_depth(2)]).           ⟹  [1,2|...]
| ?- format('X=~s -- ~3d s', [[0'j,0'ó],3245]).     ⟹  X=jó -- 3.245 s
```

Kifejezések kiírása – felhasználó vezérelte formázás

- `print(@X)`: Alapértelmezésben azonos `write`-tal. Ha a felhasználó definiál egy `portray/1` eljárást, akkor a rendszer minden a `print`-tel kinyomtatandó részkifejezésre meghívja `portray`-t. Ennek sikere esetén feltételezi, hogy a kiírás megtörtént, meghiúsulás esetén maga írja ki a részkifejezést.
  A rendszer a `print` eljárást használja a változó-behelyettesítések és a nyomkövetés kiírására!
- `portray(@Kif)` (felhasználó által definiálandó ún. *kampó eljárás*): Igaz, ha `Kif` kifejezést a Prolog rendszernek *nem* kell kiírnia (és ekkor maga a `portray` kell, hogy elvégezze a kiírást).
- Példa:

```
portray(Matrix) :-
     Matrix = [[_|_]|_],
     (   member(Row, Matrix),
         nl, print(Row), fail
     ;   true
     ).
```

```
| ?- X = [[1,2],[3,4],[5,6]].

X =
[1,2]
[3,4]
[5,6] ?
```


Karakterek kiírása és beolvasása

- `put_code(+Kód)`: Kiírja az adott kódú karaktert.
- `nl`: Kiír egy soremelést.
- `get_code(?Kód)`: Beolvas egy karaktert és (karakterkódját) egyesíti `Kód`-dal. (File végénél `Kód = -1`.)
- `peek_code(?Kód)`: A soronkövetkező karakter kódját egyesíti `Kód`-dal. A karaktert nem távolítja el a bemenetről. (File végénél `Kód = -1`.)
- Példa:

```
% rd_line(L): L a következő sor karakterkódjainak listája.
% read_line néven beépített eljárás SICStus 3.9.0-tól.
rd_line(L) :-
        peek_code(0'\n), !, get_code(_), L = [].
rd_line([C|L]) :-
        get_code(C), rd_line(L).
```

```
| ?- rd_line(L), member(X, L), put_code(X), write(' '), fail ; nl.
|: Hello world!
H e l l o   w o r l d !
```


Példa: számbeolvasás

```
% számbe(Szám): a Szám szám következik az input-folyamban.
számbe(Szám) :-
        számjegy(Érték), számbe(Érték, Szám).

% Az eddig beolvasott Szám0-val együtt az input-folyamban következő
% szám értéke Szám.
számbe(Szám0, Szám) :-
        számjegy(E), !,
        Szám1 is Szám0*10+E,
        számbe(Szám1, Szám).
számbe(Szám, Szám).

% Érték értékű számjegy következik.
számjegy(Érték) :-
        peek_code(Kar),
        Kar >= 0'0, Kar =< 0'9,
        get_code(_),
        Érték is Kar - 0'0.
```

```
| ?- számbe(X), get_code(_), számbe(Y).
|: 123 456                    ⟹    X = 123, Y = 456
```


Kifejezések beolvasása

- `read(?Kif)`: Beolvas egy ponttal lezárt kifejezést és egyesíti `Kif`-fel. (File végénél `Kif = end_of_file`.)
- `read_term(?Kif, +Opciók)`: Mint `read/1`, de az `Opciók` opciólistát is figyelembe veszi.
- Példa – botcsinálta programbeolvasó:

```
consult_body :-
    repeat,
        read(Term),
        (   Term = end_of_file -> true
        ;   assertz(Term), fail
        ),
    !.
```

```
| ?- consult_body.
|: p(X) :- q(X), r(X).
|: ^D
yes
```

```
| ?- listing([p/1]).
p(A) :-
        q(A),
        r(A).
yes
```

Be- és kiviteli csatornák

- Csatornák megnyitása és kezelése:
  - `open(@Filenév, @Mód, -Csatorna)`: Megnyitja a `Filenév` nevű állományt `Mód` módban (`read`, `write` vagy `append`). A `Csatorna` argumentumban visszaadja a megnyitott csatorna „nyelét”.
  - `set_input(@Csatorna)`, `set_output(@Csatorna)`: Az ezt követő beviteli/kiviteli eljárások `Csatorna`-t használják majd (jelenlegi csatorna).
  - `current_input(?Csatorna)`, `current_output(?Csatorna)`: A jelenlegi beviteli/kiviteli csatornát egyesíti `Csatorna`-val.
  - `close(@Csatorna)`: Lezárja a `Csatorna` csatornát.
- Explicit csatornamegadás be- és kiviteli eljárásokban
  - Az eddig ismertetett összes be- és kiviteli eljárásnak van egy eggyel több argumentumú változata, amelynek első argumentuma a csatorna. Ezek: `write/2`, `writeq/2`, `write_canonical/2`, `write_term/3`, `print/2`, `read/2`, `read_term/3`, `format/3`, `put_code/2`, `tab/2`, `nl/1`, `get_code/2`, `peek_code/2`.

Egy egyszerűbb be- és kiviteli szervezés: DEC10 I/O

- `see(@Filenév)`, `tell(@Filenév)`: Megnyitja a `Filenév` file-t olvasásra/írásra és a jelenlegi csatornává teszi. Újabb híváskor csak a jelenlegi csatornává teszi.
- `seeing(?Filenév)`, `telling(?Filenév)`: A jelenlegi beviteli/kiviteli csatorna állománynevét egyesíti `Filenév`-vel.
- `seen`, `told`: Lezárja a jelenlegi beviteli/kiviteli csatornát.
- Példák – nagyon egyszerű `consult` variánsok:

```
consult_dec10_style(File) :-
    seeing(Old), see(File),
    repeat,
        read(Term),
        (   Term = end_of_file
        ->  seen
        ;   assertz(Term), fail
        ),
    !,
    see(Old).
```

```
consult_with_streams(File) :-
    open(File, read, S),
    repeat,
        read(S, Term),
        (   Term = end_of_file
        ->  close(S)
        ;   assertz(Term), fail
        ),
    !.
```


Hibakezelési beépített eljárások

- Hibahelyzetet beépített eljárás rossz argumentumokkal való meghívása, vagy a `throw/1` (`raise_exception/1`) eljárás válthat ki.
- Minden hibahelyzetet egy Prolog kifejezés (ún. hiba-kifejezés) jellemez.
- Hiba „dobása”, azaz a `HibaKif` hibahelyzet kiváltása:
  `throw(@HibaKif)`,
  `raise_exception(@HibaKif)`
- Hiba „elkapása”:
  `catch(:+Cél, ?Minta, :+Hibaág)`,
  `on_exception(?Minta, :+Cél, :+Hibaág)`
  - Hatása: Futtatja a `Cél` hívást.
    - Ha `Cél` végrehajtása során hibahelyzet nem fordul elő, futása azonos `Cél`-lal.
    - Ha `Cél`-ban hiba van, a hiba-kifejezést egyesíti `Mintá`-val.
    - Ha ez sikeres, meghívja a `Hibaág`-at.
    - Ellenkező esetben továbbdobja a hiba-kifejezést, hogy a további körülvevő `catch` eljárások esetleg elkaphassák azt.

Programfejlesztési beépített eljárások (SICStus specifikusak)

- `set_prolog_flag(+Jelző, @Érték)`: `Jelző` értkékét `Érték`-re állítja.
- `current_prolog_flag(?Jelző, ?Érték)`: `Jelző` pillanatnyi értéke `Érték`.
- Néhány fontos Prolog jelző:
  - `argv`: csak olvasható, a parancssorbeli argumentumok listája.
  - `unknown`: viselkedés definiálatlan eljárás hívásakor (`trace`, `fail`, `error`).
  - `source_info`: forrásszintű nyomkövetés (`on`, `off`, `emacs`).
- `consult(:@Files)`, `[:@File,...]`: Betölti a `File`(ok)at, interpretált alakban.
- `compile(:@File)`: Betölti a `File`(ok)at, leforditott alakot hozva létre.
- `listing`: Kiírja az összes interpretált eljárást az aktuális kimenetre.
- `listing(:@EljárásSpec)`: Kiírja a megnevezett interpretált eljárásokat.
- Itt és később: `EljárásSpec` – név vagy funktor, eseteg modul-kvalifikációval ellátva, ill. ezek listája, pl. `listing(p)`, `listing([m:q,p/1])`.

Programfejlesztési eljárások (folytatás)

- `statistics`: Különféle statisztikákat ír ki az aktuális kimenetre.
- `statistics(?Fajta, ?Érték)`: `Érték` a `Fajta` fajtájú mennyiség értéke.
  - Példa: `statistics(runtime, E)` $\Longrightarrow$ `E=[Tdiff, T]`, `Tdiff` az előző lekérdezés óta, `T` a rendszerindítás óta eltelt idő, ezredmásodpercben.
- `break`: Egy új interakciós szintet hoz létre.
- `abort`, `halt`: Kilép a legkülső interakciós szintre ill. a Prolog rendszerből.
- `trace`: Elindítja az interaktív nyomkövetést.
- `debug`, `zip`: Elindítja a szelektív nyomkövetést, csak spion-pontoknál áll meg.
  (A `zip` mód gyorsabb, de nem gyűjt annyi információt mint a `debug` mód.)
- `nodebug`, `notrace`, `nozip`: Leállítja a nyomkövetést.
- `spy(:@EljárásSpec)`: Spion-pontot tesz a megadott eljárásokra.
- `nospy(:@EljárásSpec)`: Megszünteti a megadott spion-pontokat.
- `nospyall`: Az összes spion-pontot megszünteti.

Tartalom

5 Haladó Prolog
- Megoldásgyűjtő beépített eljárások
- Meta-logikai eljárások
- A keresési tér szűkítése
- Vezérlési eljárások
- A Prolog megvalósítási módszereiről
- Determinizmus és indexelés
- Jobbrekurzió és akkumulátorok
- Kényelmi eszközök: Definite Clause Grammars (DCG), ciklusok
- Listák és fák akkumulálása – példák
- Imperatív programok átírása Prologba
- Modularitás
- Magasabbrendű eljárások
- Dinamikus adatbáziskezelés
- Egy összetettebb példaprogram
- „Hagyományos” beépített eljárások
- Fejlettebb nyelvi és rendszerelemek

Külső nyelvi interfész

- Hagyományos (pl. C nyelvű) programrészek meghívásának módja:
  - A Prolog rendszer elvégzi az átalakítást a Prolog alak és a külső nyelvi alak között. Kényelmesebb, biztonságosabb mint a másik módszer, de kevésbé hatékony. Többnyire csak egyszerű adatokra (egész, valós, atom). (MProlog)
  - A külső nyelvi rutin pointereket kap Prolog adatstruktúrákra, valamint hozzáférési algoritmusokat ezek kezelésére. Nehézkesebb, veszélyesebb, de jóval hatékonyabb mint az előző megoldás. Összetett adatok adásvételére is jó. (SWI, SICStus)

Külső nyelvi interfész – példa

- A példa a `library(bdb)` megvalósításából származik.
- A C nyelven megirandó eljárás Prolog hívási alakja:
  `index_keys(+`*`Spec`*`, +`*`Kif`*`, -`*`Kulcs`*`, -`*`Szám`*`)`
- A megirandó eljárás jelentése:
  - Ha *`Spec`* és *`Kif`* különböző funktorú kifejezések, akkor *`Szám`* `= -1` és *`Kulcs`* `= []`.
  - Egyébként, ha *`Spec`* valamelyik argumentuma + és *`Kif`* megfelelő argumentuma változó, akkor *`Szám`* `= -2` és *`Kulcs`* `= []`.
  - Egyébként *`Szám`* a *`Spec`* argumentumaként előforduló + névkonstansok száma, *`Kulcs`* pedig *`Kif`* megfelelő argumentumainak *kivonatából* képzett lista. A kivonat lényegében az argumentum funktora, azzal az eltéréssel, hogy a konstansok kivonata maga a konstans, struktúrák esetén pedig a struktúra neve és az aritása külön elemként kerül a kivonat-listába.

Külső nyelvi interfész – példa (SICStus 3)

- A példaeljárás használata

```
| ?- [ixtest].
| ?- index_keys(f(+, -, +, +),
                f(12.3, _, s(1, _, z(2)), t),
                Kulcs, Szam).
Kulcs = [12.3,s,3,t], Szam = 3 ?
```

- Az `ixtest.pl` Prolog file tartalmazza az interfész specifikációját:

```
foreign(ixkeys, index_keys(+term, +term, -term, [-integer])).
                          % 1. arg: bemenő, általános kifejezés
                          % 2. arg: bemenő, általános kifejezés
                          % 3. arg: kimenő, általános kifejezés
                          % 4. arg: a C függvény értéke, egész (long)
foreign_resource(ixkeys, [ixkeys]).

:- load_foreign_resource(ixkeys).
```

- A C programot elő kell készíteni a Prolog számára az `splfr` (link foreign resource) eszköz segítségével:

```
splfr ixkeys ixtest.pl +c ixkeys.c
```


Külső nyelvi interfész – a C kód (`ixkeys.c` állomány)

```
#include <sicstus/sicstus.h>

#define NA -1  /* not applicable */
#define NI -2  /* instantiatedness */

long ixkeys(SP_term_ref spec,
      SP_term_ref term, SP_term_ref list)
{
  unsigned long sname, tname, plus;
  int sarity, tarity, i;
  long ret = 0;
  SP_term_ref arg = SP_new_term_ref(),
              tmp = SP_new_term_ref();

  SP_get_functor(spec, &sname, &sarity);
  SP_get_functor(term, &tname, &tarity);
  if (sname != tname || sarity != tarity)
    return NA;

  plus = SP_atom_from_string("+");

  for (i = sarity; i > 0; --i) {
   unsigned long t;
   SP_get_arg(i, spec, arg);
   SP_get_atom(arg, &t); /* no check */
   if (t != plus) continue;

   SP_get_arg(i, term, arg);
   switch (SP_term_type(arg)) {
   case SP_TYPE_VARIABLE:
     return NI;
   case SP_TYPE_COMPOUND:
     SP_get_functor(arg, &tname, &tarity);
     SP_put_integer(tmp, (long)tarity);
     SP_cons_list(list, tmp, list);
     SP_put_atom(arg, tname);
     break;
   }
   SP_cons_list(list, arg, list); ++ret;
  }
  return ret;
}
```


Hasznos lehetőségek SICStus Prolog-ban

- Tetszőleges nagyságú egész számok
  pl.:

```
| ?- fakt(40,F).

F = 815915283247897734345611269596115894272000000000 ?
```

- Globális változók (Blackboard)

`bb_put(Kulcs, Érték)`
A `Kulcs` kulcs alatt eltárolja `Érték`-et, az előző értéket, ha van, törölve. (`Kulcs` egy (kis) egész szám vagy névkonstans lehet.)

`bb_get(Kulcs, Érték)`
Előhívja `Érték`-be a `Kulcs` értékét.

`bb_delete(Kulcs, Érték)`
Előhívja `Érték`-be a `Kulcs` értékét, majd kitörli.

Hasznos lehetőségek SICStus Prolog-ban *(folytatás)*

- Visszaléptethető módon változtatható kifejezések

`create_mutable(Adat, ValtKif)`
`Adat` kezdőértékkel létrehoz egy új változtatható kifejezést, ez lesz `ValtKif`. `Adat` nem lehet üres változó.

`get_mutable(Adat, ValtKif)`
`Adat`-ba előveszi `ValtKif` pillanatnyi értékét.

`update_mutable(Adat, ValtKif)`
A `ValtKif` változtatható kifejezés új értéke `Adat` lesz. Ez a változtatás visszalépéskor visszacsinálódik. `Adat` nem lehet üres változó.

- Takarító eljárás

`call_cleanup(Hivas, Tiszito)`
Meghívja `call(Hivas)`-t és ha az véglegesen befejezte futását, meghívja `Tiszito`-t. Egy eljárás akkor fejezte be véglegesen a futását, ha további alternatívák nélkül sikerült, meghiúsult vagy kivételt dobott.

Fejlett vezérlési lehetőségek SICStusban: Blokk-deklarációk

- Példa:

  ```
  :- block p(-, ?, -, ?, ?).
  ```

  Jelentése: ha az első és a harmadik argumentum is behelyettesítetlen változó (blokkolási feltétel), akkor a `p` hívás felfüggesztődik.

  Ugyanarra az eljárásra több vagylagos feltétel is szerepelhet, pl.
  `:- block p(-, ?), p(?, -).`
- Végtelen választási pontok kiküszöbölése blokk-deklarációval

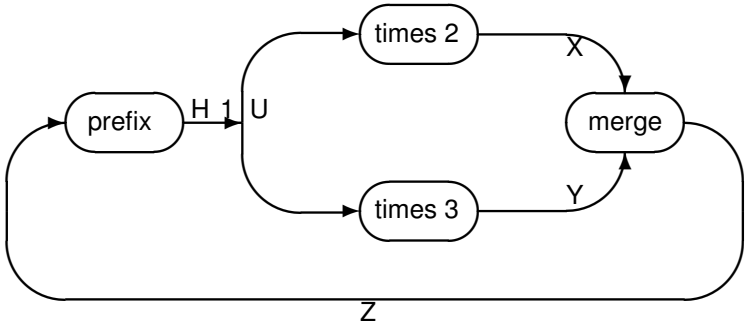
  ```
  :- block append(-, ?, -).

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


Blokk-deklarációk *(folytatás)*

- Generál-és-ellenőriz típusú programok gyorsítása
  - általában nem hatékonyak (pl megrajzolja_1), mert túl sok visszalépést használnak
  - korutinszervezéssel a generáló és ellenőrző rész „automatikusan” összefésülhető
  - ehhez az ellenőrző részt kell előre tenni és megfelelően blokkolni
- Korutinszervezésre épülő programok
  - Példa: egyszerűsített Hamming feladat
    - keressük a $2^i * 3^j (i \geq 1, j \geq 1)$ alakú számok közül az első `N` darabot nagyság szerint rendezve.
    - „stream-and-parallelism” közelítésmódot használva korutinszervezéssel egyszerűen lehet megoldani

Hamming probléma




```
% A H lista az első N, csak a 2 és 3 tényezőkből álló szám.
hamming(N, H) :-
     U = [1|H], times(U, 2, X), times(U, 3, Y),
     merge(X, Y, Z), prefix(N, Z, H).

% times(X, M, Z): A Z lista az X elemeinek M-szerese
:- block times(-, ?, ?).
times([A|X], M, Z) :- B is M*A, Z = [B|U], times(X, M, U).
times([], _, []).
```


Hamming probléma (folyt.)

```
% merge(X, Y, Z): Z az X és Y összefésülése.
:- block merge(-, ?, ?), merge(?, -, ?).
% Csak akkor fusson, ha az első két argumentum ismert
merge([A|X], [B|Y], V) :-
     A < B, !, V = [A|Z], merge(X, [B|Y], Z).
merge([A|X], [B|Y], V) :-
     B < A, !, V = [B|Z], merge([A|X], Y, Z).
merge([A|X], [A|Y], [A|Z]) :-
     merge(X, Y, Z).
merge([], X, X) :- !.
merge(_, [], []).

% prefix(N, X, Y): Az X lista első N eleme Y.
prefix(0, _, []) :- !.
prefix(N, [A|X], [A|Y]) :-
     N > 0, N1 is N-1, prefix(N1, X, Y).
```

Korutinszervező eljárások

- `freeze(X, Hivas)`
  `Hivas`t felfüggeszti mindaddig, amig `X` behelyettesítetlen változó.
- `frozen(X, Hivas)`
  Az `X` változó miatt felfüggesztett hívás(oka)t egyesíti `Hivas`-sal.
- `dif(X, Y)`
  `X` és `Y` nem egyesíthető. Mindaddig felfüggesztődik, amig ez el nem dönthető.
- `call_residue_vars(Hivas, Valtozok)`
  `Hivas`-t végrehajtja, és a `Valtozok` listában visszaadja mindazokat az új (a `Hivas` alatt létrejött) változókat, amelyekre vonatkoznak felfüggesztett hívások. Pl.

```
| ?- call_residue_vars((dif(X,f(Y)), X=f(Z)), Vars).
     X = f(Z),
     Vars = [Z,Y],
     prolog:dif(f(Z),f(Y)) ?
```


SICStus könyvtárak

- Könyvtár betöltése `:- use_module(library(`*könyvtárnév*`)).`
- A legfontosabb könyvtárak
  - `avl` – AVL fák segítségével megvalósított „asszociációs listák” (kiterjeszthető leképezések Prolog kifejezések között)
  - `atts` – tetszőleges attributumok hozzárendelése Prolog változókhoz, tárolórekeszként és az egyesítés módosítására is használható
  - `bdb` – Prolog kifejezések állományokban való tárolására szolgáló adatbázis, felhasználó által definiált többszörös indexelési lehetőséggel
  - `between` – számintervallumok felsorolása
  - `codesio` – karakterlistából olvasó ill. abba író be- és kiviteli eljárások
  - `file_systems` – állományok és könyvtárak kezelése
  - `heaps` – bináris kazal (heap), pl. prioritásos sorokhoz (priority queue)
  - `lists` – listakezelő alapműveletek
  - `logarr` – logaritmikus elérési idejű kiterjeszthető tömbök
  - `odbc` – ODBC adatbázis interfész
  - `ordsets` – halmazműveletek (halmaz $\equiv$ `@<` szerint rendezett lista)
  - `queues` – sorokra (queue, FIFO store) vonatkozó műveletek

Legfontosabb SICStus könyvtárak, folyt.

- `random` – véletlenszám-generátor
- `samsort` – általános rendezés (nem szűri az ismétlődő elemeket)
- `sockets` – socket interfész
- `system` – operációsrendszer-szolgáltatások elérése
- `terms` – általános Prolog kifejezések kezelése (pl. változók kigyűjtése)
- `timeout` – célok futási idejének korlátozása
- `trees` – az `arrays` könyvtárhoz hasonló, de nem-kiterjeszthető logaritmikus elérési idejű tömbfogalom, bináris fákkal
- `ugraphs` – élcimkék nélküli irányított és irányítatlan gráfok
- `wgraphs` – irányított és irányítatlan gráfok, egészértékű élcimkékkel
- `linda/client` és `linda/server` – Linda-szerű processzkommunikáció
- `clpb` – Boole-értékekre vonatkozó korlátmegoldó (constraint solver)
- `clpq` és `clpr` – korlátmegoldó a racionálisak és a valósok tartományán
- `clpfd` – véges tartományokra vonatkozó korlátmegoldó
- `tcltk` – A *Tcl/Tk* nyelv és eszközkészlet elérése
- `gauge` – Prolog programok a teljesítményvizsgálata `tcltk` grafikus felülettel

Új irányzatok a logikai programozásban – kitekintés

- Bevezetés a Logikai Programozásba c. jegyzet 6. fejezete:
  - Párhuzamos megvalósítások
  - Az Andorra-I rendszer rövid bemutatása
  - A Mercury nagyhatékonyságú LP megvalósítás
  - Korlát-logikai programozás (Constraint Logic Programming – CLP)
- Az utolsó két témával foglalkozik a **„Nagyhatékonyságú deklaratív programozás”** c. MSc szakirányos tárgy (őszi félévben)