Deklaratív programozás 10. előadás

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

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

2026. ősz

I. rész

Programozás Prologban

1 Programozás Prologban

2 Haladó Prolog

Az *univ* beépített eljárás

- Kiindulás: `| ?- K=F(A,B).` $\Rightarrow$ szintaxis-hiba, helyette: `K=..[F,A,B]`, pl.:

```
| ?- el(a,b,10) =.. L.        ⟹    L = [el,a,b,10]
| ?- Kif =.. [el,a,b,10].     ⟹    Kif = el(a,b,10)
| ?- alma =.. L.              ⟹    L = [alma]
```

- Az *univ* eljárás hívási mintái: `+Kif =..  ?Lista`
  `-Kif =..  +Lista` (`Lista` zárt végű!)
- 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 (szám- vagy név)konstans.
- További példák:

```
| ?- 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]]

          (SWI Prologban:)    ⟹    L=['[|]',a,[b,c]]
```

Struktúrák kezelése: a `functor/3` eljárás (kieg. anyag)

- `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 kezelése: az `arg/3` eljárás (kieg. anyag)

- `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)
```

Alkalmazás: részkifejezések keresése (kieg. anyag)

- A feladat: adott egy tetszőleges kifejezés, soroljuk fel a benne levő számokat, és minden szám esetén adjuk meg az ún. *kiválasztóját*!
- Egy részkifejezés kiválasztója egy olyan lista, amely megadja, hogy sorra mely argumentumpozíciók mentén juthatunk el hozzá.
- Az `[`$i_1, i_2, \ldots, i_k$`]` $k \geq 0$ lista egy `Kif`-ből az i_1-edik argumentum i_2-edik argumentumának, ... i_k-adik argumentumát választja ki.
  (Az `[]` kiválasztó `Kif`-ből `Kif`-et választja ki.)
- Pl. `a*b+f(5,8,7)/c`-ben `b` kiválasztója `[1,2]`, `7` 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),
        X =.. [_F|Args], nth1(I, Args, X1),                    % (*)
        kif_szám(X1, N, Kiv).
```

```
| ?- kif_szám(f(5,[8,b]), Sz, K).⟹   Sz = 5, K = [1]  ? ;
                                    Sz = 8, K = [2,1] ? ; no
```

- A `(*)` sor helyett ez is állhat:

```
        functor(X, _F, ArgNo), between(1, ArgNo, I), arg(I, X, X1),
```

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`.
- 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
```

- Az `atom_codes(Atom, KódLista)` beépített eljárás 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.

Atomok kezelése: példák (kieg. anyag)

- Keresés névkonstansokban

```
% Atom-ban a Rész nem üres részatom kétszer ismétlődik.
dadogó_rész(Atom, Rész) :-
    atom_codes(Atom, Cs),
    Ds = [_|_],
    append([_,Ds,Ds,_], Cs), % append/2, lásd library(lists)
    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).
```

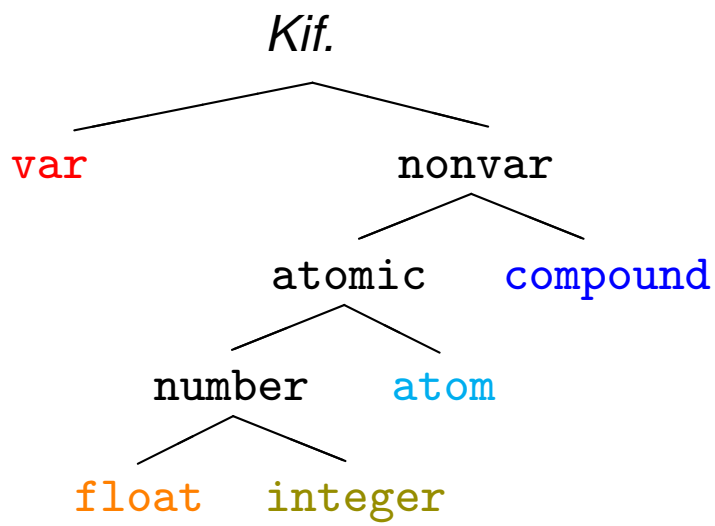
```
| ?- atom_concat(abra, kadabra, A). ⟹    A = abrakadabra ?
```

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.
- `| ?- number_codes(12, Cs).` $\Longrightarrow$ `Cs = [49,50]`
  `| ?- number_codes(0123, [0'1|L]).` $\Longrightarrow$ `L = [50,51]`
  `| ?- number_codes(N, " - 12.0e1").` $\Longrightarrow$ `N = -120.0`
  `| ?- number_codes(N, "12e1").` $\Longrightarrow$ **hiba (nincs `.0`)**
  `| ?- number_codes(120.0, "12e1").` $\Longrightarrow$ `no` **(mert a szám adott! :-)**
- A `number_codes(Szám, KódLista)` beépített eljárás végrehajtása:
  - Ha `Szám` adott (bemenő), és a $c_1 c_2 \ldots c_n$ karakterekből áll, akkor `KódLista`-t egyesíti a $[k_1, k_2, \ldots, k_n]$ 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.

Prolog kifejezések általános rendezése: a $\prec$ reláció



A különböző kif.-fajták sorrendje:

$$\texttt{var} \prec \texttt{float} \prec \texttt{integer} \prec \texttt{atom} \prec \texttt{compound}$$

Egy kifejezésfajtán belüli sorrendezés szabályai:

- Változók: rendszerfüggő (pl. memóriacím alapján)
- Egész és lebegőpontos számok: szokásosan ($x \prec y \Leftrightarrow x < y$)
- Atomok: lexikografikus sorrend ($\texttt{abc} \prec \texttt{abcd}$, $\texttt{abcv} \prec \texttt{abcz}$)
- Összetett kif.-ek: $\texttt{név}_a(a_1, \ldots, a_n) \prec \texttt{név}_b(b_1, \ldots, b_m) \Leftrightarrow$
  1. $n < m$, pl. $\texttt{p(x,s(u,v,w))} \prec \texttt{a(b,c,d)}$, vagy
  2. $n = m$, és $\texttt{név}_a \prec \texttt{név}_b$ (lexikografikusan), pl. $\texttt{a(x,y)} \prec \texttt{p(b,c)}$, vagy
  3. $n = m$, $\texttt{név}_a = \texttt{név}_b$, és az első olyan i-re melyre $a_i \neq b_i$, $a_i \prec b_i$, pl. $\texttt{r(1,u+v,3,x)} \prec \texttt{r(1,u+v,5,a)}$

Kifejezések összehasonlítása – beépített eljárások

- Beépített eljárások tetszőleges kifejezések összehasonlítására:

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

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

  `| ?- [1, 2, 3, 4] @< struktúra(1, 2, 3).` $\Longrightarrow$ **yes**

- Beépített elj. tetszőleges lista rendezésére: `sort(+L, ?S)`
  Jelentése: az `L` lista `@<` szerinti rendezése `S`,
  `==/2` szerint azonos elemek ismétlődését kiszűrve.

```
| ?- sort([a,c,a,b,b,c,c,b,d,a(2,3),c(1),2.0,1,X], S).
S = [X,2.0,1,a,b,c,d,c(1),a(2,3)] ? ; no
(SWI):S = [X,1,2.0,a,b,c,d,c(1),a(2,3)]. :-(
```

Összefoglalás: a Prolog egyenlőség-szerű beépített eljárásai

- `U = V`: `U` egyesítendő `V`-vel.
  Soha sem jelez hibát.
- `U == V`: `U` azonos `V`-vel.
  Soha sem jelez hibát és soha sem helyettesít be.
- `U =:= V`: Az `U` és `V` aritmetikai kifejezések értéke megegyezik.
  Hibát jelez, ha `U` vagy `V` nem (tömör) aritmetikai kifejezés.
- `U is V`: `U` egyesítendő a `V` aritmetikai kifejezés értékével.
  Hiba, ha `V` nem (tömör) aritmetikai kifejezés.
- (`U =.. V`: `U` „szétszedettje” a `V` lista)

```
| ?- X = 1+2.       ⟹   X = 1+2
| ?- 3 = 1+2.       ⟹   no
| ?- X == 1+2.      ⟹   no
| ?- 3 == 1+2.      ⟹   no
| ?- +(1,2)==1+2    ⟹   yes
| ?- X =:= 1+2.     ⟹   hiba
| ?- 1+2 =:= X.     ⟹   hiba
| ?- 2+1 =:= 1+2.   ⟹   yes
| ?- 2.0 =:= 1+1.   ⟹   yes
| ?- 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
| ?- 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.

Az egyesítés kiegészítése: előfordulás-ellenőrzés, *occurs check*

- Kérdés: `X` és `s(X)` egyesíthető-e?
  - A matematikai válasz: *nem*, egy változó nem egyesíthető egy olyan struktúrával, amelyben előfordul (ez az előfordulás-ellenőrzés).
  - Az ellenőrzés költséges, ezért alaphelyzetben nem alkalmazzák (emiatt ún. ciklikus kifejezések keletkezhetnek)
  - Szabványos eljárásként rendelkezésre áll:
    `unify_with_occurs_check/2`
  - Kiterjesztés (pl. SICStus): az előfordulás-ellenőrzés elhagyása miatt keletkező ciklikus kifejezések tisztességes kezelése.
- Példák:

```
| ?- X = s(1,X).
        X = s(1,s(1,s(1,s(1,s(...))))) ?
| ?- unify_with_occurs_check(X, s(1,X)).
        no
| ?- X = s(X), Y = s(s(Y)), X = Y.
        X = s(s(s(s(s(...))))), Y = s(s(s(s(s(...))))) ?
```

II. rész

Haladó Prolog

1 Programozás Prologban

2 Haladó Prolog

Tartalom

2 Haladó Prolog
- A keresési tér szűkítése
- Determinizmus és indexelés
- Jobbrekurzió és akkumulátorok
- Imperatív programok átírása Prologba

Nyelvi eszközök a keresési tér szűkítésére

- 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.
- A vágó beépített eljárás – procedurális szemantika (ismétlés)
  - mindig sikerül; de mellékhatásként megszünteti a választási pontokat egészen a szülő célig, azt is beleértve. (Egy *C* cél szülője az a cél, amelyet, a *C*-t tartalmazó klóz fejével illesztettünk.)
- A vágó „színe”:
  - zöld, ha csak olyan ágakat vág le, amelyek nem vezetnek megoldáshoz (nem módosítja a megoldások halmazát);
  - piros, ha megoldást tartalmazó ágat is levág (módosítja a megoldáshalmazt)

Példa: `első_poz_elem(+L, ?P)`: `P` az `L` lista első pozitív eleme

- Rekurzív megoldás (mérnöki)

```
első_poz_elem1([X|L], EP) :- (   X > 0 -> EP = X
                              ;   első_poz_elem1(L, EP)
                              ).
```

- Nem-rekurzív, deklaratív, de lassú megoldás (matematikusi):

```
első_poz_elem2(L, EP) :-                % L első pozitív eleme EP ha
    append(Pref, [EP|_], L),            % Pref ++ [EP] ++ _ = L,
    EP>0, \+ ( member(X, Pref), X>0 ). % EP>0, Pref-nek nincs pozitív eleme
```

- Itt amikor `EP>0` sikerül, a felsorolás leállítható, hiszen egy hátrábbi `EP` nem lehet `L` első pozitív eleme:

```
első_poz_elem3(L, EP) :-
    append(Pref, [EP|_], L), EP>0, !, \+ ( member(X, Pref), X>0 ).
```

- Ha `EP` kimenő (`(+,-)` mód), akkor a vágó elérésekor `Pref`-nek nyilván nem lehet pozitív eleme. További egyszerűsítések:

```
első_poz_elem4(L, EP) :- append(_Pref, [EP|_], L), EP>0, !.
első_poz_elem5(L, EP) :- member(EP, L), EP>0, !.
```

- A 4.-5. változat `(+,+)` módban hibás: `első_poz_elem4([1,2], 2)` $\Longrightarrow$ `yes`
- A vágás **alapszabálya**: kimenő paraméter a vágó után kapjon értéket:

```
első_poz_elem6(L, EP) :- member(X, L), X>0, !, EP=X. % (+,+) módban is jó!
```

A 3.–6, megoldás épít az `append/2`, ill. `member/2` felsorolási sorrendjére!

A vágás alapszabálya a mérnöki megoldásban.

- Az első megoldásban a felt. szerkezetet írjuk át többklózos formára:

```
első_poz_elem7([X|_], X) :- X > 0.
első_poz_elem7([X|L], EP) :- X =< 0, első_poz_elem7(L, EP).
```

- Szúrjunk be egy (zöld) vágót!

```
első_poz_elem8([X|_], X) :- X > 0, !.
első_poz_elem8([X|L], EP) :- X =< 0, első_poz_elem8(L, EP).
```

- A második klózból hagyjuk el a „felesleges” negált vizsgálatot

```
első_poz_elem9([X|_], X) :- X > 0, !.
első_poz_elem9([X|L], EP) :- első_poz_elem9(L, EP).
```

- Ez a változat is hibás `(+,+)` módban: `első_poz_elem9([1,2], 2)` $\Longrightarrow$ `yes`
- A **vágás alapszabályának** betartásával kijavítható a hiba:

```
első_poz_elem10([X|_], EP) :- X > 0, !, EP = X.
első_poz_elem10([X|L], EP) :- első_poz_elem10(L, EP).
```

- Ez nemcsak általánosabban használható, hanem hatékonyabb kód is: csak akkor helyettesíti be a kimenő paramétert, ha tudja, hogy pozitív
- Az alapszabály betartásakor az ún. indexelés is hatékonyabb lesz

További példák a vágás alapszabályának betartására

```
% 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).

% páros_elemei(L, Pk): Pk az L egészlista páros elemeinek listája.
páros_elemei([], []).
páros_elemei([X|L], Pk) :-
    X mod 2 =:= 0, !, Pk = [X|Pk1], páros_elemei(L, Pk1).
páros_elemei([_X|L], Pk) :-
    % _X mod 2 =:= 1,
    páros_elemei(L, Pk).
```

- Kezdő Prolog programozóknak **a diszjunktív feltételes szerkezet használatát javasoljuk** a vágó helyett.

Példa: `max(X, Y, Z)`: `X` és `Y` maximuma `Z` (kieg. anyag)

- 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 */.
```

Tartalom

2 Haladó Prolog
- A keresési tér szűkítése
- Determinizmus és indexelés
- Jobbrekurzió és akkumulátorok
- Imperatív programok átírása Prologba

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) alkalmazhatók.
- Hogyan ismerheti fel a fordító a választásmentességet
  - egyszerű feltételes szerkezet (vö. Elixir őrfeltétel)
  - indexelés (indexing)
  - vágó és indexelés kölcsönhatása
- Az alábbi definíciók esetén SICStusban 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).
```

Indexelés

- Mi az indexelés?
  - egy adott hívásra illeszthető klózok gyors kiválasztása,
  - egy eljárás klózainak fordítási idejű csoportosításával.
- A legtöbb Prolog rendszeraz 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 $\Rightarrow$ 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 klózok:
  - ha `A` változó, akkor `(1) (2) (3) (4) (5)`
  - ha `A = 0`, akkor `(1) (2)`
  - ha `A` fő funktora `s/1`, akkor `(2) (3) (4)`
  - ha `A = 9`, akkor `(2) (5)`
  - minden más esetben `(2)`

- 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.
```

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 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).
- Ez egy újabb érv a vágás alapszabálya mellett:

A kimenő paraméterek értékadását mindig a vágó után végezzük!

A vágó és az indexelés hatékonysága (kieg. anyag)

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

```
% determ. xx=''          % determ. lefut. xx='c'      % választásmentes, xx='ci'
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

2 Haladó Prolog
- A keresési tér szűkítése
- Determinizmus és indexelés
- Jobbrekurzió és akkumulátorok
- Imperatív programok átírása Prologba

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 a mutábilis (angolul *mutable* – 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 mut. változó kezdőértéke;
    - `S` az összeg a `sum/3` lefutása után: a mut. 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 `sum/3` második klóza ilyen alakra hozva:

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

További akkumulátoros példák (kieg. anyag)

- Többszörös akkumulálás – lista összege és négyzetösszege

```
% sum2(+L, +S0, ?S, +Q0, ?Q): S-S0 =Σ Li, Q-Q0 = Σ Li^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 Li, Q-Q0 = Σ Li^2
sum3([], SQ, SQ).
sum3([X|L], SQ0, SQ) :-
        plus3(X, SQ0, SQ1), sum3(L, SQ1, SQ).
        % teljesen analóg a "sima" összegzővel

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

Tartalom

2 Haladó Prolog
- A keresési tér szűkítése
- Determinizmus és indexelés
- Jobbrekurzió és akkumulátorok
- Imperatív programok átírása Prologba

Hogyan írjunk át imperatív nyelvű algoritmust Prolog programmá?

- Példafeladat: Hatékony hatványozási algoritmus
  - Alaplépés: a kitevő felezése, az alap négyzetre emelése.
  - 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 ciklusban három változót használunk: `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).

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

  ism:  if (h > 0)
        {  if (h & 1)

             e *= a;
 /* else e is unchanged; */

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

```
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).
```

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, vagy if-then-else 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 (kieg. anyag)

- 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 (kieg. anyag)

- 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 */
}
```