DETERMINIZMUS ÉS INDEXELÉS

Determinizmus

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

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


A determinisztikus lefutás

- Mi a determinisztikus lefutás haszna?
  - a futás gyorsabb lesz,
  - a tárigény csökken,
  - más optimalizálások (pl. jobbrekurzió) alkalmazható.
- Hogyan ismeri fel a fordító azt, hogy nem kell választási pont?
  - indexelés (indexing)
  - vágók és feltételes szerkezetek
- Az alábbi definíciók esetén a `p(`*Nonvar*`, Y)` hívás nem hoz létre választási pontot (a 2. definíció esetén a `p(`*Var*`, Y)` sem):

```
p(1, a).      | p(1, Y) :- !,     | p(X, Y) :-
p(2, b).      |     Y = a.        |     (   X > 1 -> Y = a
              | p(_, b).          |     ;   Y = b
              |                   |     ).
```


Indexelés

- Mi az indexelés?
  - egy hívásra illeszthető klózok gyors kiválasztása,
  - egy eljárás klózainak fordítási idejű csoportosításával,
- A legtöbb Prolog rendszer, így a SICStus Prolog is, az első fej-argumentum alapján indexel (first argument indexing).
- Az indexelés alapja az első fejargumentum külső funktora:
  - `C` szám vagy névkonstans esetén `C/0`;
  - `R` nevű és `N` argumentumú struktúra esetén `R/N`;
  - változó esetén nem értelmezett.
- Az indexelés megvalósítása:
  - Fordításkor a funktorokhoz elkészítjük az illeszthető klózok részhalmazát.
  - Futáskor lényegében konstans idő alatt választunk a részhalmazok közül.
  - ***Fontos:*** ha egyelemű a részhalmaz, nem hozunk létre választási pontot!

Példa indexelésre

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

- 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(3, Y)` nem hoz létre választási pontot.
  - `p(s(1), Y)` létrehoz választási pontot, de determinisztikusan fut le.
  - `p(s(0), Y)` nemdeterminisztikusan fut le.

Struktúrák, változók a fejargumentumban

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

```
p(0, a).        | q(0, a).          | q_seged(0, b).
p(s(0), b).     | q(s(X), Y) :-     | q_seged(1, c).
p(s(1), c).     |     q_seged(X, Y).|
p(9, z).        | q(9, z).          |
```

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

```
rhossz([], 0).
rhossz(L, RH) :- L = [_|_], length(L, H), RH is 1/H.
% rhossz([X|L], RH) :-  length([X|L], H), RH is 1/H. % nem hatékony!
```


Indexelés és aritmetika, indexelés és listák

- Aritmetikai elágazások
  - Az indexelés nem foglalkozik aritmetikai vizsgálatokkal.
  - Pl. az `N = 0` és `N > 0` feltételek nem „zárják ki” egymást.
  - Az alábbi `fakt(N, F)` lefutása nem-determinisztikus:

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

- Listakezelő eljárások
  - 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 (összefűzésre).

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

- Csak akkor tudjuk fordításkor kizárni a vágót követő klózokat, ha garantált, hogy az adott fő funktor esetén a vágót elérjük. Ennek feltételei:
  - az első argumentumban konstans, vagy legáltalánosabb struktúra legyen,
  - a további argumentumok változók legyenek,
  - a fejben az összes változóelőfordulás különböző legyen,
  - a törzs első hívása a vágó (megengedve a fejillesztést kiváltó =-t).
- Példa: `p(s(A, B, C), D, E) :- !, ....`
- Ez egy újabb érv a vágás alapszabálya mellett:
  A kimenő paraméterek értékadását mindig a vágó után végezzük!

A vágó és az indexelés hatékonysága

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

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

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

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

- Futási idők $N = 2000$ esetén

	`fib`	`fibc`	`fibci`
futási idő	4410 ms	4060 ms	3820 ms
meghiúsulási idő	730 ms	0 ms	0 ms
összesen	5140 ms	4060 ms	3820 ms

Választás-mentesség diszjunktív feltételes szerkezetek esetén

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

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

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


JOBBREKURZIÓ ÉS AKKUMULÁTOROK

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 tagjának utolsó helyén, stb., és
  - a rekurzív hívás pillanatában nincs választási pont a predikátumban, tehát a rekurzív hívást megelőző célok determinisztikusan futottak le.
- Jobbrekurzió optimalizálás: az utolsó hívás végrehajtása ***előtt*** a predikátum által lefoglalt hely felszabadul ill. szemétgyüjtésre alkalmassá válik.
- Ez az optimalizálás nemcsak rekurzív hívás esetén, hanem minden ***utolsó*** hívás esetén megvalósul — 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:

```
% sum(+L, ?S): Az L számlista elemeinek összege S.
sum([], 0).
sum([X|L], S):- sum(L,S0), S is S0+X.
```

- Első jobbrekurzív változat, csak ellenőrzésre használható:

```
% sum1(+L, +S): Az L számlista elemeinek összege S.
sum1([], 0).
sum1([X|L], S) :- /* S is S0+X  helyett: */ S0 is S-X, sum1(L, S0).
```

- Második jobbrekurzív változat, csak kiírni tudja az eredményt:

```
% sum2(+L): Az L számlista elemeinek összegét kiírja.
sum2(L):- sum2(L, 0).

% sum2(+L, +S0): Az L lista S0-lal növelt összegét kiírja.
sum2([], S) :-    write(S), nl.
sum2([X|L], S0):- S1 is S0+X, sum2(L, S1).
```


Jobbrekurzív listaösszeg — akkumulátorpár segítségével

- Harmadik változat: teljes értékű jobbrekurzív lista-összegző:

```
% sum3(+L, ?S): Az L számlista elemeinek összege S.
sum3(L, S):- sum3(L, 0, S).

% sum3(+L, +S0, ?S): Az L lista elemeit hozzáadva S0-hoz kapjuk S-et.
sum3([], S, S).
sum3([X|L], S0, S):-
        S1 is S0+X, sum3(L, S1, S).
```

- Az ***akkumulátor*** fogalma:
  - A `sum3(L, S0, S)` predikátumban az `S0` és `S` argumentumok egy akkumulátorpárt alkotnak.
  - Az akkumulátorpár két része egy változó mennyiség (a példában az összeg) különböző időpontokban vett értékeit mutatja:
    - `S0` az összeg értéke a `sum3/3` ***meghívásakor***;
    - `S` az összeg értéke a `sum3/3` ***lefutása után***.

Akkumulátorok használata

- Az akkumulátorpárok a hagyományos, „változtatható” változók megfelelői.
- Az általános séma:

```
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: *Vált0*; közbülső értékek: *Vált1*, ..., *Váltn*; végérték: *Vált*.
- A Prolog akkumulátorpár nem más mint a funkcionális programozásból ismert gyűjtőargumentum és a függvény eredményének együttese.

Akkumulátorok használata — folytatás

- **Három lista összege**

```
% sum_3_lists(+L, +LL, +LLL, +S0, ?S): Az L, LL, LLL számlisták
% összegeinek összege S-S0
sum_3_lists(L, LL, LLL, S0, S) :-
        sum3(L, S0, S1), sum3(LL, S1, S2), sum3(LLL, S2, S).
```

- **Többszörös akkumulálás — listák összege és négyzetösszege**

```
% sum12(+L, +S0, ?S, +Q0, ?Q): S-S0 =Σ Li, Q-Q0 = Σ Li*Li
sum12([], S, S, Q, Q).
sum12([X|L], S0, S, Q0, Q):-
        S1 is S0+X, Q1 is Q0+X*X, sum12(L, S1, S, Q1, Q).
```

- **Többszörös akkumulátorok összevonása**

```
% sum12(+L, +S0/Q0, ?S/Q): S-S0 =Σ Li, Q-Q0 = Σ Li*Li
sum12([], SQ, SQ).
sum12([X|L], S0/Q0, SQ):-
        S1 is S0+X, Q1 is Q0+X*X, sum12(L, S1/Q1, SQ).
```


Korábbi listakezelő predikátumok

- **A `revapp` mint akkumuláló eljárás**

```
% revapp(Xs, L0, L): Xs megfordítását L0 elé fűzve 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 (előfeltétel: `L0` szuffixuma `L`-nek).**
- **Az `append` is tekinthető akkumuláló eljárásnak (a 2. és 3. arg. felcserélt). A változtatás: az `L0` elejéről sorra elhagyjuk `Xs` elemeit, végül marad `L`.**

```
% 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).
```


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, [A|L0], 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
```

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 akkumulátor-pár, 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
```

ALGORITMUSOK PROLOGBAN

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;
}
```


A `hatv` C függvénynek megfelelő Prolog eljárás

- A fügvény eredménye a reláció utolsó arg.-a: `hatv(+A, +H, ?E)`: $A^H = E$.
- A ciklusnak segédeljárás felel meg: `hatv(+A, +H, +E0, ?E)`: $A^H * E0 = E$.
- Az »`a`« és »`h`« C változóknak az »`+A`« és »`+H`« bemenő *paraméterek*, az »`e`« C változónak az »`+E0, ?E`« ***akkumulátor-pár*** felel meg.

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

hatv(H, A, E0, E) :- H > 0, !,
        (   H /\ 1 =:= 1
        ->  E1 is E0*A
        ;   E1 = E0
        ),
        H1 is H >> 1,
        A1 is A*A,
        hatv(H1, A1, 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:
  - 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 ezutáni 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 megfeleltetett Prolog változóval.
- A Prolog eljárás fejkommentje a C ciklus ***ciklus-invariánsa***.

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 > 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-invariáns*-sal, 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 (h > 0)
  { /* ciklus-invariáns: a0^h0 == e*a^h */
    /* 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 */
}
```


Második példa: Fibonacci sorozat tagjainak hatékony számítása

- A C függvény

```
unsigned fib(unsigned n)
{ unsigned f = 0, fnxt = 1, t;
  while (n > 0) t = fnxt, fnxt += f, f = t, --n;  /* (1) */
  return f;
}
```

- Az (1) ciklusnak bemenő változói: `n`, `f`, `fnxt`, kimenő változója: `f`.
- A ciklusnak megfeleltetett Prolog eljárás: `fib(N, F0, FNXT, F)`: az `F0` és `FNXT` kezdőértékű Fibonacci sorozat `N`-edik tagja `F`.

```
% "betű szerinti" Prolog átírás:
fib(N, F0, FNXT, F) :- N > 0, !,
    T = FNXT, FNXT1 is FNXT+F0,
    F1 = T, N1 is N-1,
    fib(N1, F1, FNXT1, F).
fib(_, F0, _, F0).
```

```
% Leegyszerűsített alak:
fib(N, F0, FNXT, F) :- N > 0, !,
    FNXT1 is FNXT+F0,
    N1 is N-1,
    fib(N1, FNXT, FNXT1, F).
fib(_, F0, _, F0).
```


Fibonacci sorozat — Prolog stílusban

- A Fibonacci sorozat teljes Prolog megvalósítása, és az ennek megfeleltethető C kód:

```
fib(N, F) :-                    % unsigned fib(unsigned N)
        fib(N, 0, 1, F).        % { unsigned F0 = 0, F1 = 1, F2;
                                %
fib(N, F0, F1, F) :-            % ism:
        N > 0, !,               %   if (N > 0)
        N1 is N-1,              %   {  --N;
        F2 is F0+F1,            %      F2 = F0+F1;
        fib(N1, F1, F2, F).     %      F0 = F1; F1 = F2; goto ism;
                                %   }
fib(_, F0, _, F0).              %   return F0;
                                % }
```

MEGOLDÁSOK GYŰJTÉSE ÉS FELSOROLÁSA

Keresési feladat Prologban — felsorolás vagy gyűjtés?

- Keresési feladat: bizonyos 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 visszalépés esetén sorra előáll minden megoldás.
- Egyszerű példa: egy lista páros elemeinek megkeresése:

```
% 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]) :-
    páros_elemei(L, Pk).
```

```
% páros_eleme(L, P): P egy páros
% eleme az L listának.
páros_eleme([P|L], P) :-
    P mod 2 =:= 0.
páros_eleme([_|L], P) :-
    páros_eleme(L, P).

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


Mi a közös a felsoroló és gyűjtő megoldásokban?

- Keressük meg a közös részt a `páros_eleme[i]` eljárásokban!
- Mindkettőben át kell lépni a páratlan elemeket:

```
% köv_páros(L0, P, L) :- Az L0 első páros eleme P, a maradék L.
köv_páros([X|L0], P, L) :-
    X mod 2 =\= 0, !, köv_páros(L0, P, L).
köv_páros([P|L], P, L).
```

- A `köv_páros` eljárásra épülő `páros_eleme[i]` eljárások:

```
% páros_elemei(L, Pk): Pk az L
% lista páros elemeinek listája.
páros_elemei(L0, Pk) :-
    köv_páros(L0, P, L1), !,
    Pk = [P|Pk1],
    páros_elemei(L1, Pk1).
páros_elemei(_, []).
```

```
% páros_eleme(L, P): P egy páros
% eleme az L listának.
páros_eleme(L0, P) :-
    köv_páros(L0, P0, L1),
    (   P = P0
    ;   páros_eleme(L1, P)
    ).
```


A gyűjtő és felsoroló sémák összehasonlítása

- A páros elemeket gyűjtő ill. felsoroló eljárások alapján adjunk meg egy általános sémát a kétféle eljárástípusra!
- Az általános esetben a keresésnek lehet egy vagy több `Param` paramétere. Például, kereshetjük az `Param`-mal osztható elemeket.
- A közös építőelem: `következő(V0, Param, E, V1)`: A `V0` kifejezéssel jellemzett keresési térben az első megoldás `E`, és a fennmaradó keresési tér `V1`, a `Param` paraméter-érték mellett.

A gyűjtő séma:

```
% A V0 keresési térben a Param
% paraméterű megoldások listája L.
megoldások(V0, Param, L) :-
  következő(V0, Param, E, V1), !,
  L = [E|L1],
  megoldások(V1, Param, L1).
megoldások(_, _, []).
```

A felsoroló séma:

```
% A V0 keresési térben E egy
% Param paraméterű megoldás.
megoldás(V0, Param, E) :-
  következő(V0, Param, E0, V1),
  (   E = E0
  ;   megoldás(V1, Param, E)
  ).
```

Egy összetettebb példa: fennsíkok felsorolása

- Egy listában fennsíknak nevezünk:
  - egy csupa azonos elemből álló, legalább kételemű, folytonos részlistát;
  - amely az ilyenek között maximális (egyik irányba sem kiterjeszthető).
- A feladat: felsorolandók egy lista fennsíkjai és kezdőpozíciójuk.
- Egy gyorsprogramozási módszerrel készült megoldás:

```
% Az L listában az F pozíción egy H hosszú fennsík van.
fennsík0(L, F, H) :-
    Teste = [E,E|_],
    append(Eleje, Teste, L),
    \+ last(Eleje, E),
    length(Eleje, F0), F is F0+1,
    kezdethossz(Teste, H).
    % kezdethossz/2 definícióját
    % lásd korábban
```

```
fennsík1(L, F, H) :-
    Teste = [E,E|_],
    append(Eleje, Teste, L),
    \+ last(Eleje, E),
    length(Eleje, F0), F is F0+1,
    (   append(Ek, Farok, Teste),
        \+ Farok = [E|_] ->
        length(Ek, H)
    ).
```


Fennsíkok felsorolása — 2., hatékony megoldás

- Használjuk a megoldás-felsoroló sémát: `megoldás(V0, Param, E)`!
  - *V0*: »`L, P`«, a bejárandó lista és első elemének pozíciója;
  - *Param* üres
  - *E* »`F, H`«, a megoldás-fennsík kezdőpozíciója és hossza.

```
% Az L listában az F pozíción egy H hosszú fennsík van.
fennsík(L, F, H) :-
        fennsík(L, 1, F, H).

% A P0-tól számozott L0 listában az F pozíción egy H hosszú fennsík van.
fennsík(L0, P0, F, H) :-
        % az első fennsík jellemzői F0 és H0, az utána levő maradék L1:
        első_fennsík(L0, P0, F0, H0, L1),
        (   F = F0, H = H0
        ;   % L1 kezdőpoz.ja, P1 = előző megoldás kezdőpoz.ja+hossza:
            P1 is F0+H0, fennsík(L1, P1, F, H)
        ).
```


Fennsíkok felsorolása — 2., hatékony megoldás (folyt.)

```
% első_fennsík(+L0, +P0, -F, -H, -L): A P0-tól számozott L0 listában az
% első fennsík az F. pozíción van és hossza H, a fennsík után fennmaradó
% rész pedig az L lista.
első_fennsík([E,E|L1], P0, F, H, L) :-
        !, F = P0, azonosak(L1, E, 2, H, L).
első_fennsík([_|L1], P0, F, H, L) :-
        P1 is P0+1,
        első_fennsík(L1, P1, F, H, L).

% azonosak(+L0, +E, +H0, -H, -L): Az L0 lista elejéről a maximális számú
% E-vel azonos elemet lehagyva marad L, a lehagyott elemek száma H-H0.
azonosak([X|L0], E, H0, H, L) :-
        E = X, !,
        H1 is H0+1,
        azonosak(L0, E, H1, H, L).
azonosak(L, _, H, H, L).
```