Nagyhatékonyságú deklaratív programozás (labor)

Szeredi Péter, Kabódi László

BME Számítástudományi és Információelméleti Tanszék

2025 tavasz

Témakörök

- Prolog alapok
- Haladó Prolog ismeretek
- A CLP (Constraint Logic Programming) irányzat áttekintése
- A SICStus `clpq/r` könyvtárai
- A SICStus `clpb` könyvtára
- A SICStus `clpfd` könyvtára
- A SICStus `chr` könyvtára
- A Mercury programozási nyelv

Háttéranyagok

- Információk a korlát-logikai programozásról
  - „Az első alapkönyv”: Pascal Van Hentenryck: Constraint Satisfaction in Logic Programming, MIT Press, 1989
  - Kim Mariott, Peter J. Stuckey, Programming with Constraints: an Introduction, MIT Press 1998
  - On-line Guide to Constraint Programming, by Roman Barták (`http://kti.ms.mff.cuni.cz/~bartak/constraints/`)
  - Krzysztof R. Apt, Mark G. Wallace, Constraint logic programming using ECLiPSe `https://www.researchgate.net/publication/220693610_Constraint_logic_programming_using_ECLiPSe`
- Információk a Mercury nyelvről
  - Honlap: `http://mercurylang.org`

I. rész

Prolog alapok

Tartalom

A Prolog alapelemei: a családi kapcsolatok példája

- Adatok
  - Adottak személyekre vonatkozó állítások, pl.

„gyerek–szülő” tábla

gyerek	szülő
Imre	István
Imre	Gizella
István	Géza
István	Sarolta
Gizella	Civakodó Henrik
Gizella	Burgundi Gizella

„férfiak” tábla

férfi
Imre
István
Géza
Civakodó Henrik

- A feladat:
  - Definiálandó az unoka–nagyszülő kapcsolat

A nagyszülő feladat — Prolog megoldás

```
% szuloje(Gy, Sz):Gy szülője Sz.
% Tényállításokból álló predikátum
szuloje('Imre', 'Gizella').  % (sz1)
szuloje('Imre', 'István').   % (sz2)
szuloje('István', 'Sarolt'). % (sz3)
szuloje('István', 'Géza').   % (sz4)
szuloje('Gizella',
        'Burgundi Gizella'). % (sz5)
szuloje('Gizella',
        'Civakodó Henrik').  % (sz6)
% ffi(Szemely): Szemely férfi.
ffi('Imre'). ffi('István').  % (f1)-(f2)
ffi('Géza').                 % (f3)
ffi('Civakodó Henrik').      % (f4)
% Gyerek nagyszülője Nagyszulo.
% Egyetlen szabályból álló predikátum
nagyszuloje(Gyerek, Nagyszulo) :-
  szuloje(Gyerek, Szulo),
  szuloje(Szulo, Nagyszulo). % (nsz)
```

```
% Ki Imre nagyapja?
| ?- nagyszuloje('Imre', NA),
     ffi(NA).
NA = 'Civakodó Henrik' ? ;
NA = 'Géza' ? ;
no
% Ki Géza unokája?
| ?- nagyszuloje(U, 'Géza').
U = 'Imre' ? ;
no
% Ki Imre nagyszülője?
| ?- nagyszuloje('Imre', NSz).
NSz = 'Burgundi Gizella' ? ;
NSz = 'Civakodó Henrik' ? ;
NSz = 'Sarolt' ? ;
NSz = 'Géza' ? ;
no
```


Deklaratív szemantika – klózok logikai alakja

- A **szabály** jelentése implikáció: a törzsbeli célok **konjunkciójából** következik a fej.
  - Példa: `nagyszuloje(U,N) :- szuloje(U,Sz), szuloje(Sz,N).`
  - Logikai alak:
    $\forall U, N, Sz(\texttt{nagyszuloje}(U, N) \leftarrow \texttt{szuloje}(U, Sz) \wedge \texttt{szuloje}(Sz, N))$
  - Ekvivalens alak:
    $\forall U, N \quad (\texttt{nagyszuloje}(U, N) \leftarrow \exists Sz(\texttt{szuloje}(U, Sz) \wedge \texttt{szuloje}(Sz, N)))$
- A **tényállítás** feltétel nélküli állítás, pl.
  - Példa: `szuloje('Imre', 'István').`
  - Logikai alakja változatlan
  - Ebben is lehetnek változók, ezeket is univerzálisan kell kvantálni
- A **célsorozat** jelentése: keressük azokat a változó-behelyettesítéseket amelyek esetén a célok konjunkciója igaz
- Egy célsorozatra kapott válasz **helyes**, ha az adott behelyettesítésekkel a célsorozat következménye a program logikai alakjának
- A Prolog garantálja a helyességet, de a **teljességet** nem: nem biztos, hogy minden megoldást megkapunk –
  kaphatunk hibajelzést, végtelen ciklust, végtelen keresési teret stb.

A Prolog végrehajtás alaplépése, az ún. redukciós lépés

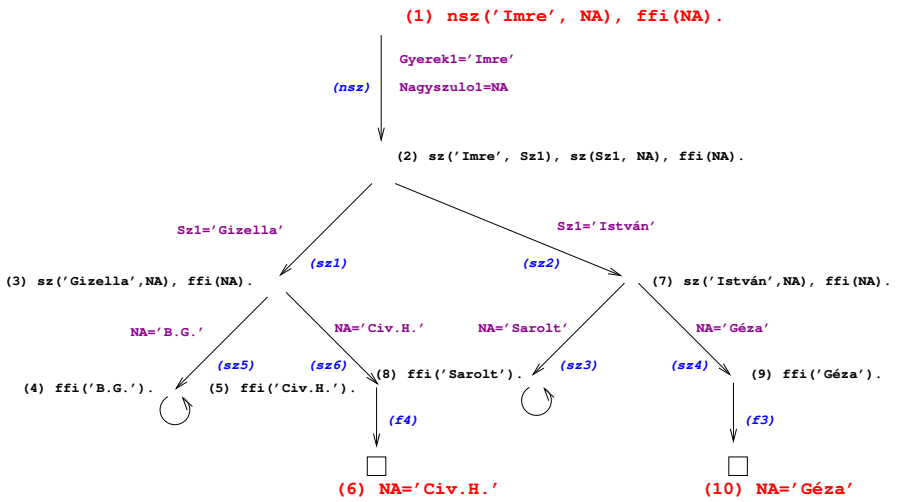
- Redukciós lépés: egy célsorozat redukálása egy újabb célsorozattá, egy programklóz (pl. az `(nsz)`) segítségével:

  `| ?- nsz('Imre', NA), ffi(NA).` `(kc)` *kezdeti célsorozat*
  `| ?- sz('Imre', Sz1), sz(Sz1, NA), ffi(NA).` `(rc)` *redukált célsorozat*

  (A fenti esetben `(kc)` $\equiv$ `(rc)`, általánosan `(kc)` $\Leftarrow$ `(rc)`.)
- Részletesen: a klózt **lemásoljuk**, a változókat újakra cserélve, pl.
  `nsz(Gy1, Nsz1) :- sz(Gy1, Sz1), sz(Sz1, Nsz1).`
- A célsorozatot (pl. `(kc)`), szétbontjuk az első hívásra és a maradékra, pl. első hívás: `nsz('Imre', NA)`, maradék: `ffi(NA)`.
- Az első hívást **egyesítjük** a klózfejjel, azaz a két kifejezést azonos alakra hozzuk (mintaillesztés):
  behelyettesítés: `Gy1 = 'Imre'`, `Nsz1 = NA`, közös alak: `nsz('Imre', NA)`
- Ha az egyesítés nem sikerül, akkor a redukciós lépés is meghiúsul.
- Sikeres egyesítés esetén az ehhez szükséges behelyettesítéseket elvégezzük a klóz **törzsén** és a **célsorozat** maradékán is
  törzs: `sz('Imre', Sz1), sz(Sz1, NA)`, maradék célsorozat: `ffi(NA)`
- Az új célsorozat: a klóztörzs és utána a maradék célsorozat, ld. fent `(rc)`

A nagyszülő példa végrehajtása – keresési tér

```
nsz(Gyerek, Nagyszulo) :-
  sz(Gyerek, Szulo),
  sz(Szulo, Nagyszulo).                    % (nsz)
```



A Prolog végrehajtási algoritmus – megjegyzések

- A keresési fában a nyilak a redukció (visszavezetés) irányát mutatják, de ...
- Az implikáció alulról felfelé irányul, pl. `(3)` $\Rightarrow$ `(2)` és `(7)` $\Rightarrow$ `(2)`.
- A végrehajtás nem „intelligens”
  - Pl. `| ?- nagyszuloje(U, 'Géza').` hatékonyabb lenne ha a klóz törzsét jobbról balra hajtanánk végre
  - DE: így a végrehajtás átlátható; a Prolog nem tételbizonyító, hanem programozási nyelv

A redukciós modell alapfogalmai

- A végrehajtás bemenete:
  - egy Prolog program (klózok sorozata), pl. a `nagyszuloje` program, és
  - egy célsorozat, pl. `:- nsz(im, Sz).`
    a megoldás értelmezése érdekében ezt egy utolsó, `answer(Megoldás)` fiktív céllal bővítjük ki, pl.

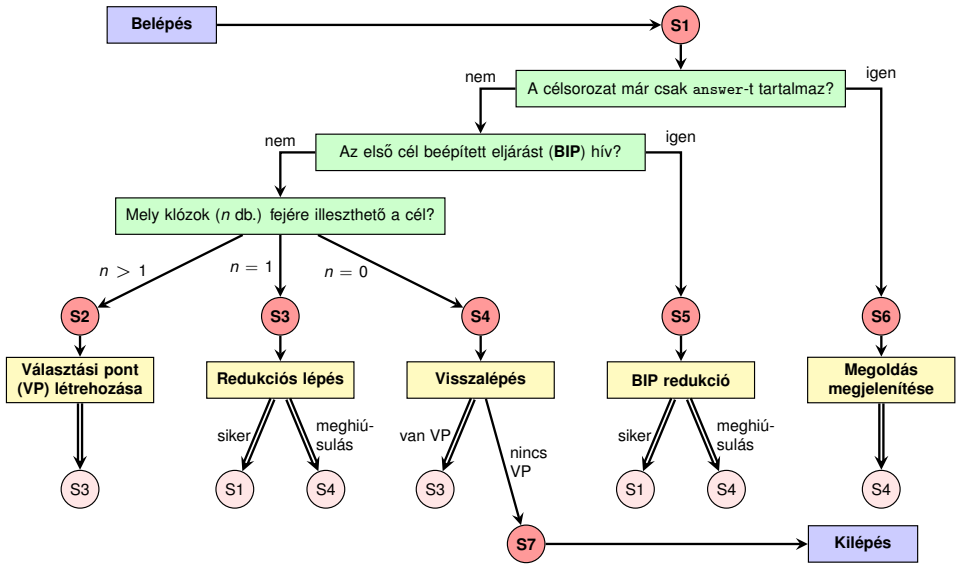
    ```
    :- nsz(im, NSz), answer(NSz).        % Kik Imre nagyszülei?
    :- sz(Gy, Sz),   answer(Gy-Sz).      % Mik a gyerek-szülő párok?
    ```
- Az `answer(...)` cél segítségével követhetjük a megoldás felépülését
- Ha a célsorozat már csak az `answer` célt tartalmazza, akkor eljutottunk egy megoldáshoz (ezt a szerepet korábban az üres célsorozat játszotta)
- Az `answer` csak egy elméleti eszköz, nem beépített elj., de definálhatjuk, így: `answer(M) :- write(M), nl, fail.`
- A végrehajtásnak többféle kimenetele lehetséges:
  - Hiba (kivétel, exception), pl. `:- Y = alma, X is Y+1.`
    (Ezzel most nem foglalkozunk részletesebben.)
  - Meghiúsulás (nincs megoldás), pl. `:- sz(ge, Sz), answer(Sz).`
  - Siker (1 vagy több megoldás), pl. `:- sz(im, Sz), answer(Sz).`

A redukciós végrehajtás alapfogalmai (folyt.)

- A végrehajtás által használt (imperatív!) adatstruktúrák:
  - a jelenlegi célsorozatot tartalmazó változó (Goal)
  - a választási pontokat (VP) tartalmazó verem (Choice point stack)
- A VP verem akkor mélyül, ha 2 vagy több klózzal lehet redukálni
  - a redukció előtt a veremre elmentjük a célsorozatot és a redukcióban használható klózok listáját, majd folytatjuk a végrehajtást
  - ennek meghiúsulása esetén
    - a veremben tárolt klózlistából elhagyjuk az első elemet,
    - ha ezután már csak egyelemű a klózlista, megszűntetjük a VP-t,
    - a klózlistában most első klózzal folytatjuk a redukciót.
  - ha meghiúsuláskor üres a VP-verem ⇒ kimerítettük a keresési teret
- Például a `nsz(im, NA), ffi(NA), answer(NA)` célsorozat végrehajtásakor az alábbi VP verem jön léte:

ChPoint name	Clause list	Goal	
CHP2	[p3,p4]	(4)	`hasP(d,Y),answer(b-Y).`
CHP1	[p2,p3,p4]	(2)	`hasP(X,P),hasP(P,Y),answer(X-Y).`

A redukciós modell folyamatábrája (összes megoldás előállítása)



(A kettős nyilak jelentése: ugrás a rózsaszínű körben megadott lépésre, azaz folytatás az adott piros körnél.)

Megjegyzések a folyamatábrához

- Hétféle végrehajtási lépésünk van: **S1**–**S7**, ahol **S1** a kiindulási pont (de közbülső is), **S7** a végállapot.
- **S1** alapvető feladata az elágaztatás **S2**–**S6** egyikére
  - ha `Goal` már csak az answer elemet tartalmazza ⇒ **S6**;
  - ha az első cél beépített eljárást hív ⇒ **S5**;
  - egyébként az első cél felhasználói eljárást hív. Ekkor megvizsgáljuk (esetleg csak közelítően), hogy az eljárás mely klózainak fejére illeszthető az első cél, és ezek száma (*n*) szerint ⇒ **S2**, **S3** vagy **S4**.
- **S2** létrehoz egy VP-t, majd az első klózzal redukál (⇒ **S3**).
- **S3** meghiúsulhat, ha **S1**-ben *n* csak közelítés volt, ilyenkor ⇒ **S4**.
- **S4**-ben a VP-ban eltárolt következő klózzal redukálunk, ha van ilyen (⇒ **S3**), egyébként befejezzük a végrehajtást (⇒ **S7**).
- **S5** az **S3** lépéssel analóg módon vagy ⇒ **S1**, vagy ⇒ **S4**.
- **S6**-ban a megoldás megjelenítése után visszalépéssel folytatjuk (⇒ **S4**, további megoldások keresése).

Tartalom

Aritmetikai beépített eljárások

Aritmetikai beépített eljárások (predikátumok)

- `X is Kif`: A `Kif` **aritmetikai** kif.-t **kiértékeli** és értékét **egyesíti** `X`-szel.
- `Kif1>Kif2`: `Kif1` **aritmetikai értéke** nagyobb `Kif2` értékénél.
- Hasonlóan: `Kif1=<Kif2`, `Kif1>Kif2`, `Kif1>=Kif2`, `Kif1=:=Kif2` (aritmetikailag egyenlő), `Kif1=\=Kif2` (aritmetikailag nem egyenlő)
- Fontos aritmetikai operátorok: `+, -, *, /, rem, //` (egész-osztás)

A faktoriális függvény definíciója Prologban

- funkc. nyelven a faktoriális 1-argumentumú függvény: `Ered = fakt(N)`
- Prologban ennek egy kétargumentumú reláció felel meg: `fakt(N, Ered)`
- Konvenció: az utolsó argumentum(ok) a kimenő pararaméter(ek)

```
% fakt(N, F): F = N!.
fakt(0, 1).           % 0! = 1.
fakt(N, F) :-         % N! = F ha létezik olyan N1, F1, hogy
        N > 0,        %    N > 0, és
        N1 is N-1,    %    N1 = N-1. és
        fakt(N1, F1), %    N1! = F1, és
        F is F1*N.    %    F = F1*N.
```


Néhány további beépített eljárás

- Kifejezések egyesítése
  - `X = Y`: az `X` és `Y` **szimbolikus** kifejezések egyesítése ≡ azonos alakra hozása változók esetleges behelyettesítésével, a lehető legáltalánosabb módon
  - `X \= Y`: az `X` és `Y` kifejezések nem egyesíthetőek (nem hozhatók azonos alakra)
- Típusvizsgálatot végző beépített predikátumok
  - `var(X)`: `X` változó
  - `nonvar(X)`: `X` nem változó
    - `atomic(X)`: `X` konstans; `atom(X)`: `X` névkonstans, `number(X)`: `X` szám `integer(X)`: `X` egész szám, `float(X)`: `X` lebegőpontos szám
    - `compound(X)`: `X` összetett kifejezés
- További hasznos predikátumok
  - `true, fail`: Mindig sikerül ill. mindig meghiúsul.
  - `write(X)`: Az `X` Prolog kifejezést kiírja.
  - `write_canonical(X)`: `X` kanonikus (alapstruktúra) alakját írja ki.
  - `nl`: Kiír egy újsort.

Programfejlesztési beépített eljárások

- `consult(File)`: A `File` állományban levő programot beolvassa és értelmezendő alakban eltárolja. (`File = user` ⇒ terminálról olvas.)
- `compile(File)`: mint `consult`, csak kompilált alakban tárol (gyorsabb kód, de egyes eljárások nem nyomkövethetők)
- `trace, notrace`: A (teljes) nyomkövetést be- ill. kikapcsolja.
- `listing` vagy `listing(Predikátum)`: Az értelmezendő alakban eltárolt összes ill. adott nevű predikátumokat kilistázza.
- `halt`: A Prolog rendszer befejezi működését.

```
> sicstus
SICStus 4.4.1 (x86_64-linux-glibc2.12) ...
| ?- consult(fakt).
% consulted /home/user/fakt.pl in module user, 10 msec 91776 bytes
yes
| ?- fakt(4, F).
F = 24 ? ;
no
| ?- listing(fakt).
(...)
yes
| ?- halt.
>
```


Tartalom

A Prolog adatfogalma, a Prolog kifejezés (term)

- konstans (`atomic`)
  - számkonstans (`number`) – egész vagy lebegőp, pl. `1`, `-2.3`, `3.0e10`
  - névkonstans (`atom`), pl. `'István'`, `szuloje`, `+`, `-` `tree_sum`
  - egy *C* konstans funktora *C*/0
- összetett- vagy struktúra-kifejezés (`compound`)
  - ún. kanonikus alak: ⟨ struktúranév ⟩(⟨ arg_1 ⟩, . . . , ⟨ arg_n ⟩)
    - a ⟨ struktúranév ⟩ egy névkonstans, az ⟨ arg_i ⟩ argumentumok tetszőleges Prolog kifejezések
    - a kifejezés funktora: ⟨ struktúranév ⟩/n
    - példák: `person(ian,smith,2003)`, `<(X,Y)`, `is(X, +(Y,1))`
  - szintaktikus „édesítőszerek”, pl. operátorok: `X is Y+1` ≡ `is(X, +(Y,1))`
- változó (`var`)
  - pl. `X`, `Szulo`, `X2`, `_valt`, `_`, `_123`
  - a változó alaphelyzetben behelyettesítetlen, értékkel nem bír, egyesítés során egy tetszőleges Prolog kifejezést (akár egy másik változót) vehet fel értékül – dinamikus típusfogalom

Adatstruktúrák Prologban – a bináris fák példája

- A bináris fa adatstruktúra
  - vagy egy csomópont (`node`), amelynek két részfája van (`left`,`right`)
  - vagy egy levél (`leaf`), amely egy egészt tartalmaz

Binárisfa-struktúra C-ben

```
enum treetype {Node, Leaf};
struct tree {
 enum treetype type;
 union {
   struct { struct tree *left;
            struct tree *right;
          } nd;
   struct { int value;
          } lf;
 } u;
};
```

A Prolog dinamikusan típusos nyelv – nincs szükség explicit típusdefinícióra

- Mercury típusleírás (komment)

```
% :- type tree --->
%         node(tree, tree)
%       | leaf(int).
```

- A típushoz tartozás ellenőrzése

```
% is_tree(T): T egy bináris fa
is_tree(leaf(V)) :- integer(V).
is_tree(node(Left,Right)) :-
    is_tree(Left),
    is_tree(Right).
```


Bináris fák összegzése

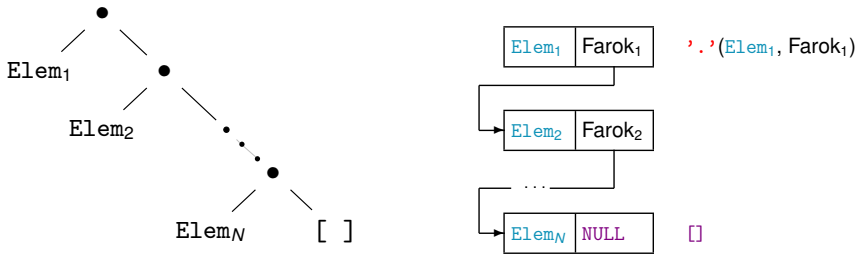
- Egy bináris fa levélösszegének kiszámítása:
  - levél esetén a levélben tárolt egész
  - csomópont esetén a két részfa levélösszegének összege

```
% S = tsum(T): T levélösszege S
int tsum(struct tree *tree)
{
 switch(tree->type) {
 case Leaf:
  return tree->u.lf.value;
 case Node:
  return tsum(tree->u.nd.left) +
         tsum(tree->u.nd.right);
   }
}
```

```
% tree_sum(Tree, S): Σ Tree = S.
tree_sum(leaf(Value), Value).
tree_sum(node(Left,Right), S) :-
        tree_sum(Left, S1),
        tree_sum(Right, S2),
        S is S1+S2.
| ?- tree_sum(node(leaf(5),
                   node(leaf(3),
                        leaf(2))),S).
S = 10 ? ;
no
| ?- tree_sum(T, 3).
T = leaf(3) ? ;
! Inst. error in argument 2 of is/2
! goal:  3 is _73+_74
```


A Prolog lista-fogalma

- A Prolog lista
  - Az üres lista a `[]` névkonstans.
  - A nem-üres lista a `'.'(Fej,Farok)` (SWI Prologban `'[|]'(Fej,Farok)`) struktúra:
    - `Fej` a lista feje (első eleme), míg
    - `Farok` a lista farka, azaz a fennmaradó elemekből álló lista.
  - A listákat egyszerűsítve is leírhatjuk („szintaktikus édesítés”).
  - Megvalósításuk optimalizált, időben és helyben is hatékonyabb.
- A listák fastruktúra alakja és megvalósítása



Listák jelölése – szintaktikus „édesítőszerek”

- Az alapvető édesítés:
  `.(Fej,Farok)` helyett a `[Fej|Farok]` kifejezést írjuk
- Kiterjesztés N darab „fej”-elemre, a skatulyázás kiküszöbölése:
  $[Elem_1|[...|[Elem_N|Farok]...]] \Longrightarrow [Elem_1,...,Elem_N|Farok]$
- Ha a farok `[]`, a „`|[]`” jelsorozat elhagyható:
  $[Elem_1,...,Elem_N|[]] \Longrightarrow [Elem_1,...,Elem_N]$

```
| ?- [1,2] = [X|Y].          ⟹   X = 1, Y = [2] ?
| ?- [1,2] = [X,Y].          ⟹   X = 1, Y = 2 ?
| ?- [1,2,3] = [X|Y].        ⟹   X = 1, Y = [2,3] ?
| ?- [1,2,3] = [X,Y].        ⟹   no
| ?- [1,2,3,4] = [X,Y|Z].    ⟹   X = 1, Y = 2, Z = [3,4] ?
| ?- L = [1|_], L = [_,2|_]. ⟹   L = [1,2|_A] ? % nyílt végű
| ?- L = .(1,[2,3|[]]).      ⟹   L = [1,2,3] ?
| ?- L = [1,2|.(3,[])].      ⟹   L = [1,2,3] ?
```


Listák összefűzése – az `append/3` eljárás

- Egy funkcionális (Erlang) megoldás:

```
append([], B) -> B;
append([X|A], B) -> [X|append(A, B)].
```

- Írjuk át a kétargumentumú `append` *függvényt* `app0/3` Prolog *eljárássá*!

```
% app0(A, B, C): A és B listák összefűzése a C lista.
app0([], B, Ret) :- Ret = B.
app0([X|A], B, Ret) :-
    app0(A, B, C), Ret = [X|C].
```

- Logikailag tiszta Prolog programokban a `Vált = Kif` alakú hívások kiküszöbölhetőek, ha `Vált` minden előfordulását `Kif`-re cseréljük.

```
app([], B, B).
app([X|A], B, [X|C]) :-
    app(A, B, C).
```

- Mindkét eljárásban a (max) futási idő arányos az 1. arg. hosszával
- Miért jobb az `app/3` mint az `app0/3`?
  - `app/3` **jobbrekurzív**, ciklussal ekvivalens (nem fogyaszt vermet)
  - `app([1,...,1000],[0],[2,...])` 1, `app0(...)` 1000 lépésben hiúsul meg.
  - `app/3` használható szétszedésre is (lásd később), míg `app0/3` nem.

Lista építése *elölről* – nyílt végű listákkal

- Egy `X` Prolog kifejezés nyílt végű lista, ha `X` változó, vagy `X = [_|Farok]` ahol `Farok` nyílt végű lista.

```
| ?- L = [1|_], L = [_,2|_].   ⟹   L = [1,2|_A] ?
```

- A beépített `append/3` azonos az `app/3`-mal:

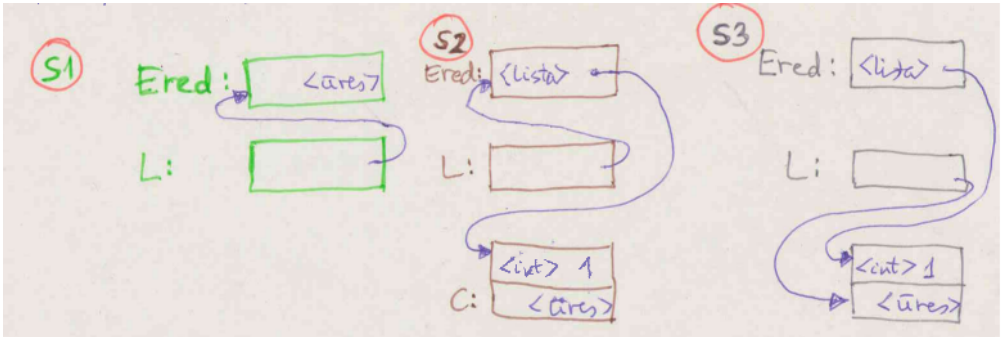
```
append([], B, B).
append([X|A], B, [X|C]) :-
    append(A, B, C).
```

- Az `append` eljárás már az első redukciónál felépíti az eredmény fejét!
  - Példa-célsorozat: `append([1,2,3], [4,5], Ered), answer(Ered).`
  - Fej: `append([X|A], B, [X|C])`
  - Behelyettesítés: `X = 1, A = [2,3], B = [4,5], Ered = [1|C]`
  - Új célsorozat: `append([2,3], [4,5], C), answer([1|C]).`
    (`Ered` nyílt végű lista, farka még behelyettesítetlen.)
  - A további redukciós lépések behelyettesítése és eredménye:

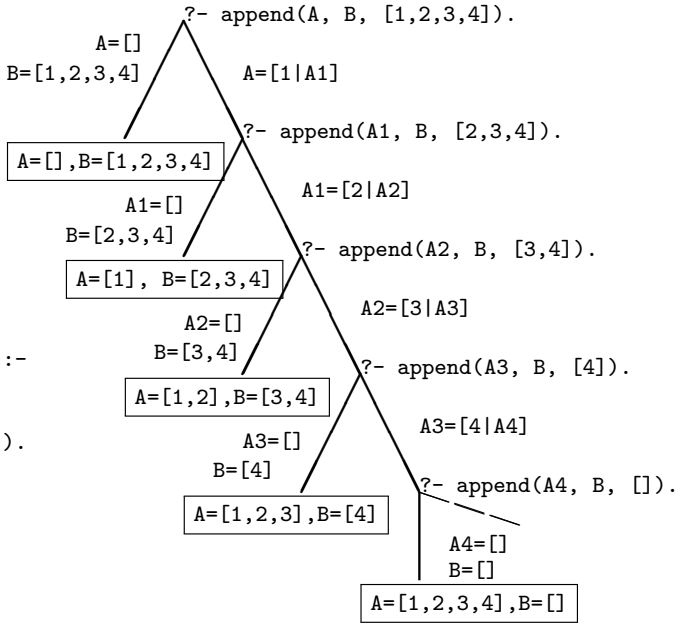
```
C  = [2|C1]   append([3], [4,5], C1),   answer([1|[2|C1]]).
C1 = [3|C2]   append([], [4,5], C2),    answer([1,2|[3|C2]]).
C2 = [4,5]                              answer([1,2,3|[4,5]]).
```


Lista építése *elölről* – a megvalósítás részletei

```
app1([], B, B).
app1([X|A], B, L /*S1 (1. hívás), S3 (2., rekurzív hívás*/) :-
    L = [X|C], /*S2*/
    app1(A, B, C).

:- app1([1,2,3], [4,5], Ered).
```



Listák szétbontása az append/3 segítségével




```
% append(L1, L2, L3):
% Az L3 lista az L1 és L2
% listák elemeinek egymás
% után fűzésével áll elő.
append([], L, L).
append([X|L1], L2, [X|L3]) :-
    append(L1, L2, L3).

| ?- append(A, B, [1,2,3,4]).
A = [], B = [1,2,3,4] ? ;
A = [1], B = [2,3,4] ? ;
A = [1,2], B = [3,4] ? ;
A = [1,2,3], B = [4] ? ;
A = [1,2,3,4], B = [] ? ;
no
```


Tartalom

Predikátumok, klózok

- Példa:

```
% két klózból álló predikátum definíciója, funktora: tree_sum/2
tree_sum(leaf(Val), Val).        %                  1. klóz, tényáll.
tree_sum(node(Left,Right), S) :- %          fej    \
        tree_sum(Left, S1),      % cél     \       |
        tree_sum(Right, S2),     % cél     | törzs | 2. klóz, szabály
        S is S1+S2.              % cél     /       /
```

- Szintaxis:

⟨ Prolog program ⟩	::=	⟨ predikátum ⟩ . . .	
⟨ predikátum ⟩	::=	⟨ klóz ⟩ . . .	{azonos funktorú}
⟨ klóz ⟩	::=	⟨ tényállítás ⟩.␣ |	
		⟨ szabály ⟩.␣	{klóz funktora = fej funktora}
⟨ tényállítás ⟩	::=	⟨ fej ⟩	
⟨ szabály ⟩	::=	⟨ fej ⟩ :- ⟨ törzs ⟩	
⟨ törzs ⟩	::=	⟨ cél ⟩, . . .	
⟨ cél ⟩	::=	⟨ kifejezés ⟩	
⟨ fej ⟩	::=	⟨ kifejezés ⟩	

Prolog kifejezések

- Példa – egy klózfej mint kifejezés:

```
% tree_sum(node(Left,Right), S)    % összetett kif., funktora
% --------- ----------------  _                    tree_sum/2
%      |         |           |
% struktúranév    \          argumentum, változó
%                  \- argumentum, összetett kif.
```

- Szintaxis:

⟨ kifejezés ⟩	::=	⟨ változó ⟩ |	{Nincs funktora}
		⟨ konstans ⟩ |	{Funktora: ⟨ konstans ⟩/0}
		⟨ összetett kif. ⟩ |	{Funktor: ⟨ struktúranév ⟩/⟨ arg.sz. ⟩}
		⟨ egyéb kifejezés ⟩	{Operátoros, lista, stb.}
⟨ konstans ⟩	::=	⟨ névkonstans ⟩ |	
		⟨ számkonstans ⟩	
⟨ számkonstans ⟩	::=	⟨ egész szám ⟩ |	
		⟨ lebegőp. szám ⟩	
⟨ összetett kif. ⟩	::=	⟨ struktúranév ⟩ (⟨ argumentum ⟩, . . .)	
⟨ struktúranév ⟩	::=	⟨ névkonstans ⟩	
⟨ argumentum ⟩	::=	⟨ kifejezés ⟩	

Lexikai elemek: példák és szintaxis

```
% változó:            Fakt FAKT _fakt X2 _2 _
% névkonstans:        fakt ≡ 'fakt' 'István' [] ; ',' += ** \= ≡ '\\='
% számkonstans:       0 -123 10.0 -12.1e8
% nem névkonstans:    !=, Istvan
% nem számkonstans:   1e8 1.e2
```

⟨ változó ⟩ ::= ⟨ nagybetű ⟩⟨ alfanum. jel ⟩... |
_ ⟨ alfanum. jel ⟩...

⟨ névkonstans ⟩ ::= ’⟨ idézett kar. ⟩...’ |
⟨ kisbetű ⟩⟨ alfanum. jel ⟩... |
⟨ tapadó jel ⟩... | ! | ; | [] | {}

⟨ egész szám ⟩ ::= {előjeles vagy előjeltelen számjegysorozat}

⟨ lebegőp.szám ⟩ ::= {belsejében tizedespontot tartalmazó számjegysorozat esetleges exponenssel}

⟨ idézett kar. ⟩ ::= {tetszőleges nem ' és nem \ karakter} |
\ ⟨ escape szekvencia ⟩

⟨ alfanum. jel ⟩ ::= ⟨ kisbetű ⟩ | ⟨ nagybetű ⟩ | ⟨ számjegy ⟩ | _

⟨ tapadó jel ⟩ ::= + | - | * | / | \ | $ | ^ | < | > | = | ` | ~ | : | . | ? | @ | # | &

Prolog programok formázása

- Megjegyzések (comment)
  - A % százalékjeltől a sor végéig
  - A /* jelpártól a legközelebbi */ jelpárig.
- Formázó elemek (komment, szóköz, újsor, tabulátor stb.) szabadon használhatók
  - kivétel: összetett kifejezésben a struktúranév után tilos formázó elemet tenni (operátorok miatt);
  - prefix operátor (ld. később) és „(” között kötelező a formázó elem;
  - klózt lezáró pont (.␣): önmagában álló pont (előtte nem tapadó jel áll) amit egy formázó elem követ
- Programok javasolt formázása:
  - Az egy predikátumhoz tartozó klózok legyenek egymás mellett a programban, közéjük ne tegyünk üres sort.
  - A predikátum elé tegyünk egy üres sort és egy fejkommentet:
    ```
    % predikátumnév(A1, ..., An): A1, ..., An közötti
    % összefüggést leíró kijelentő mondat.
    ```
  - A klózfejet írjuk sor elejére, minden célt lehetőleg külön sorba, néhány szóközzel beljebb kezdve

Tartalom

Operátoros kifejezések

- Példa: `S is -S1+S2` ekvivalens az `is(S, +(-(S1),S2))` kifejezéssel
- Szintaxis:

⟨ összetett kif. ⟩ ::=
⟨ struktúranév ⟩ (⟨ argumentum ⟩, ...) {eddig csak ez volt}
| ⟨ argumentum ⟩ ⟨ operátornév ⟩ ⟨ argumentum ⟩ {infix kifejezés}
| ⟨ operátornév ⟩ ⟨ argumentum ⟩ {prefix kifejezés}
| ⟨ argumentum ⟩ ⟨ operátornév ⟩ {posztfix kifejezés}
| (⟨ kifejezés ⟩) {zárójeles kif.}

⟨ operátornév ⟩ ::= ⟨ struktúranév ⟩ {ha operátorként lett definiálva}

- Operátor(ok) definiálása
  `op(Prio, Fajta, OpNév)` vagy `op(Prio, Fajta, [OpNév`$_1$`,...OpNév`$_n$`])`, ahol
  - `Prio` (prioritás): 1–1200 közötti egész
  - `Fajta`: az `yfx`, `xfy`, `xfx`, `fy`, `fx`, `yf`, `xf` névkonstansok egyike
  - `OpNév`$_i$ (az operátor neve): tetszőleges névkonstans
- Az `op/3` beépített predikátum meghívását általában a programot tartalmazó fájl elején, *direktívában* helyezzük el:
  ```
  :- op(800, xfx, [szuloje,nagyszuloje]).  'Imre' szuloje 'István'.
  ```
- A direktívák a programfájl *betöltésekor* azonnal végrehajtódnak.

Operátorok jellemzői

- Egy operátort jellemez a fajtája és prioritása
- A fajta az asszociatívitás irányát és az írásmódot határozza meg:

Fajta			Írásmód	Értelmezés
bal-assz.	jobb-assz.	nem-assz.		
`yfx`	`xfy`	`xfx`	infix	`A f B` ≡ `f(A, B)`
	`fy`	`fx`	prefix	`f A` ≡ `f(A)`
`yf`		`xf`	posztfix	`A f` ≡ `f(A)`

- A zárójelezést a prioritás és az asszociatívitás együtt határozza meg, pl.
  - `a/b+c*d` ≡ `(a/b)+(c*d)` mert `/` és `*` prioritása 400 < 500 (`+` prioritása) (kisebb prioritás = erősebb kötés)
  - `a-b-c` ≡ `(a-b)-c` mert a `-` operátor fajtája `yfx`, azaz bal-asszociatív – balra köt, balról jobbra zárójelez (a fajtanévben az `y` betű mutatja az asszociatívitás irányát)
  - `a^b^c` ≡ `a^(b^c)` mert a `^` operátor fajtája `xfy`, azaz jobb-asszociatív (jobbra köt, jobbról balra zárójelez)
  - `a=b=c` szintaktikusan hibás, mert az `=` operátor fajtája `xfx`, azaz nem-asszociatív

Szabványos, beépített operátorok

Szabványos operátorok
Színkód: már ismert, új aritmetikai

1200	`xfx`	`:- -->`	
1200	`fx`	`:- ?-`	
1100	`xfy`	`;`	diszjunkció
1050	`xfy`	`->`	if-then
1000	`xfy`	`','`	
900	`fy`	`\+`	negáció
700	`xfx`	`= \=`	
		`< =< > >= =:= =\= is`	
		`@< @=< @> @>= == \== =..`	
500	`yfx`	`+ - \/ /\`	bitműveletek
400	`yfx`	`* / // rem`	
		`mod`	modulus
		`<< >>`	léptetések
200	`xfx`	`**`	hatványozás
200	`xfy`	`^`	
200	`fy`	`- \`	bitenkénti negáció

További beépített operátorok SICStus Prologban

1150	`fx`	`mode public`
		`dynamic block`
		`volatile`
		`discontiguous`
		`initialization`
		`multifile`
		`meta_predicate`
1100	`xfy`	`do`
900	`fy`	`spy nospy`
550	`xfy`	`:`
500	`yfx`	`\`
200	`fy`	`+`

Operátorok zárójelezése

- Egy `X` $\texttt{op}_1$ `Y` $\texttt{op}_2$ `Z` zárójelezése, ahol $\texttt{op}_1$ és $\texttt{op}_2$ prioritása n_1 és n_2:
  - ha $n_1 > n_2$ akkor `X` $\texttt{op}_1$ `(Y` $\texttt{op}_2$ `Z)`;
  - ha $n_1 < n_2$ akkor `(X` $\texttt{op}_1$ `Y)` $\texttt{op}_2$ `Z`; (kisebb prio. ⇒ erősebb kötés)
  - ha $n_1 = n_2$ és $\texttt{op}_1$ jobb-asszociatív (`xfy`), akkor `X` $\texttt{op}_1$ `(Y` $\texttt{op}_2$ `Z)`;
  - egyébként, ha $n_1 = n_2$ és $\texttt{op}_2$ bal-assz. (`yfx`), akkor `(X` $\texttt{op}_1$ `Y)` $\texttt{op}_2$ `Z`;
  - egyébként szintaktikus hiba
- Érdekes példa: `:- op(500, xfy, +^).  % :- op(500, yfx, +).`

```
| ?- :- write((1 +^ 2) + 3), nl.  ⇒ (1+^2)+3
| ?- :- write(1 +^ (2 + 3)), nl.  ⇒ 1+^2+3
```

  tehát: konfliktus esetén az első operátor asszociativitása „győz”.
- Alapszabály: egy n prioritású operátor zárójelezetlen operandusaként
  - legfeljebb $n-1$ prioritású operátort fogadunk el az `x` oldalon
  - legfeljebb n prioritású operátort fogadunk el az `y` oldalon
- A zárójelezett kifejezéseket és az alapstruktúra-alakú kifejezéseket feltétel nélkül elfogadjuk operandusként
- Az alapszabály a prefix és posztfix operátorokra is alkalmazandó

Operátorok – további megjegyzések

- Ugyanaz a névkonstans használható többféle fajtájú operátorként is, pl. a ‘-’ és ‘+’ atomok prefix és infix beépített operátorként is definiálva vannak a Prolog ISO szabványában
- A „vessző” jel három szintaktikus helyzetben is használható:
  - összetett kifejezés (struktúra) argumentumait határoló jel pl. `szuloje('István', 'Gizella')`
  - listaelemeket határoló jel, pl. `[1,2,3|T]`
  - 1000 prioritású `xfy` op. pl.: `(p:-a,b,c)`≡`:-(p,',',(a,',',(b,c)))`
- A vessző atomként csak a `','`, határolóként csak a `,`, operátorként mindkét formában – `','` vagy `,` – használható.
- `:-(p, a,b,c)` többértelmű: $\stackrel{?}{=}$ `:-(p, (a,b,c))`, ... $\stackrel{?}{=}$ `:-(p,a,b,c)`...
- Egyértelműsítés: argumentumban vagy listaelemben az 1000-nél $\geq$ prioritású operátort tartalmazó kifejezést *zárójelezni kell*:

```
| ?- write_canonical((a,b,c)).  ⟹  ','(a,','(b,c))
| ?- write_canonical(a,b,c).    ⟹  ! write_canonical/3 does not exist
```

Operátorok törlése, lekérdezése

- Egy vagy több operátor törlésére az `op/3` beépített eljárást használhatjuk, ha első argumentumként (prioritásként) 0-t adunk meg.

```
| ?- X = a+b, op(0, yfx, +). ⟹   X = +(a,b) ? ; no
| ?- X = a+b.                ⟹   ! Syntax error
                                 ! op. expected after expression
                                 ! X = a <<here>> + b .
| ?- op(500, yfx, +).       ⟹   yes
| ?- X = +(a,b).            ⟹   X = a+b ? ; no
```

- Az adott pillanatban érvényes operátorok lekérdezése:

```
current_op(Prioritás, Fajta, OpNév)
```

```
| ?- current_op(P, F, +).
⟹   F = fy, P = 200 ? ;
     F = yfx, P = 500 ? ;
no
| ?- current_op(_, xfy, Op), write_canonical(Op), write(' '), fail.
; do -> ',' : ^
no
```


Operátorok felhasználása

- Mire jók az operátorok?
  - aritmetikai eljárások kényelmes irására, pl. `X is (Y+3) mod 4`
  - szimbolikus kifejezések kezelésére (pl. szimbolikus deriválás)
  - klózok leírására (`:-` és `','` is operátor), és meta-eljárásoknak való átadására, pl `asserta( (p(X):-q(X),r(X)) )`
  - eljárásfejek, eljáráshívások olvashatóbbá tételére:

```
:- op(800, xfx, [nagyszülője, szülője]).
Gy nagyszülője N :-  Gy szülője Sz, Sz szülője N.
```

  - adatstruktúrák olvashatóbbá tételére, pl.

```
sav(kén, h*2-s-o*4).
```


Operátoros példa: polinom behelyettesítési értéke

- Polinom: az 'x' atomból és számokból a '+' és '*' op.-okkal felépülő kif.
- A feladat: egy polinom értékének kiszámolása egy adott `x` érték esetén.

```
% value_of0(P, X, V): A P polinom x=X helyen vett értéke V.
value_of0(x, X, V) :-
    V = X.
value_of0(N, _, V) :-
    number(N), V = N.
value_of0(P1+P2, X, V) :-
    value_of0(P1, X, V1),
    value_of0(P2, X, V2),
    V is V1+V2.
value_of0(P1*P2, X, V) :-
    value_of0(P1, X, V1),
    value_of0(P2, X, V2),
    V is V1*V2.
```

```
| ?- value_of0((x+1)*x+x+2*(x+x+3), 2, V).
V = 22 ? ; no
```


Klasszikus szimbolikuskifejezés-feldolgozás: deriválás

- Írjunk olyan Prolog predikátumot, amely az `x` névkonstansból és számokból a `+`, `-`, `*` műveletekkel képzett kifejezések deriválását elvégzi!

```
% deriv(Kif, D):  Kif-nek az x szerinti deriváltja D.
deriv(x, D) :-                  D = 1.
deriv(C, D) :-                  number(C), D = 0.
deriv(U+V, DU+DV) :-            deriv(U, DU), deriv(V, DV).
deriv(U-V, DU-DV) :-            deriv(U, DU), deriv(V, DV).
deriv(U*V, DU*V + U*DV) :-      deriv(U, DU), deriv(V, DV).
```

```
| ?- deriv(x*x+x, D).
  ⟹   D = 1*x+x*1+1 ? ; no
| ?- deriv((x+1)*(x+1), D).
  ⟹   D = (1+0)*(x+1)+(x+1)*(1+0) ? ; no
| ?- deriv(I, 1*x+x*1+1).
  ⟹   I = x*x+x ? ; no
| ?- deriv(I, 0).
  ⟹   no
```

Tartalom

Example 1: checking if an integer is a prime

- A Prolog program consists of predicates (Boolean functions)
- Let's write a predicate `prime(P)` to determine if `P` is a prime
- A non-recursive executable specification, first in English, then in Prolog:

```
prime(P) :-                 % P is a prime if
  integer(P), P > 1,        %   P is an integer and P > 1 and
  P1 is P-1,                %   P1 = P-1 and
  \+                        %   it is not the case that
    (                       %     (   there exists an I such that:
      between(2, P1, I),    %         2 =< I =< P1  and
      P mod I =:= 0         %         P modulo I equals 0
    ).                      %     ).
```

- `X is Expr` is a built-in predicate (BIP): it evaluates the arithmetic expression `Expr`, and assigns its value to `X`
- The BIP `Expr1 =:= Expr2` evaluates both `Expr1` and `Expr1`, and succeeds iff the values are equal; analogously for `Expr1 > Expr2`, etc.
- Library predicate `between(From, To, Int)` enumerates in `Int` all integers between given integers `From` and `To`.

Example 2: append - multiple uses of a single predicate

- `app(L1, L2, L3)` is true if `L3` is the concatenation of lists `L1` and `L2`.

```
app([], L, L).                % appending an empty list with L gives L.
app([H|L1], L2, [H|L3]) :-    % appending a list composed of
                              % head H and tail L1 with a list L2
                              % gives a list with head H and tail L3 if
   app(L1, L2, L3).           %     appending L1 and L2 gives L3.
```

- app can be used, for example,
  - to check whether the relation holds:
    `| ?- app([1,2], [3], [1,2,3]).` ⟹ `yes`
  - to append two lists:
    `| ?- app([1,2], [3,4], L).` ⟹ `L = [1,2,3,4] ? ; no`
  - to split a list into two:
    `| ?- app(L1, L2, [1,2,3]).` ⟹
    ```
    L1 = [], L2 = [1,2,3] ? ;
    L1 = [1],  L2 = [2,3] ? ;
    L1 = [1,2],  L2 = [3] ? ;
    L1 = [1,2,3], L2 = [] ? ; no
    ```
- Predicate `app` is available as a built-in: `append/3` (`append` with 3 args)

The logic variable

- A variable in Prolog is a „first class citizen" data structure
- The 2nd clause of `app` sets its 3rd arg. to a list whose tail is yet unknown:

```
app([], L, L).
app([H|L1], L2, [H|L3]) :-
            % Here L3 is still unbound, [H|L3] is an open ended list
    app(L1, L2, L3).                                        (*)
```

- In the goal `(*) L3` can be viewed as a pointer to the location where the output list is to be deposited
- Multiple occurences of yet uninstantiated variables are allowed:

```
% X appears in List on two subsequent positions
double_member(X, List) :- append(_, [X,X|_], List).
```

`| ?- double_member(X, [a,b,b,a,a]).` ⟹ `X = b ? ; X = a ?`

- A single underline (_) is a so called *void* variable, each occurence of which represents a new variable
- The data structure `[X,X|_]` is actually implemented by the first list element cell pointing to the second one (or the other way round)

Example 3: Higher order predicates

- Let's write a predicate `double(N, D)`, which expects a number in `N` and returns its double in `D`:

```
% double(N, D): D = 2*N
double(N, D) :- D is 2*N.
```

- Higher order predicate `maplist(Pred, L1, L2)` takes the name of a 2-argument predicate `Pred`, and two lists `L1` and `L2`. It executes `Pred` on pairs of corresponding elements of `L1` and `L2`, e.g.

```
| ?- maplist(double, [1,3,4,6], L).  ⟹    L = [2,6,8,12] ?
```

- A nondeterministic example:

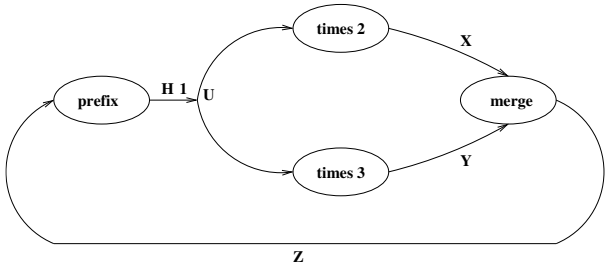
```
% neighbor(I, J): J differs by 1 from I.
neighbor(I, J) :- J is I+1.
neighbor(I, J) :- J is I-1.
```

- Example runs:

```
| ?- maplist(neighbor, [5,8], L).
L = [6,9] ? ; L = [6,7] ? ; L = [4,9] ? ; L = [4,7] ? ; no
| ?- L = [_N1,_N2], neighbor(5, _N1), neighbor(8, _N2).
L = [6,9] ? ; L = [6,7] ? ; L = [4,9] ? ; L = [4,7] ? ; no
```


Prolog extensions: coroutining (Prolog II)

- Wikipedia: Coroutines are computer program components that allow execution to be suspended and resumed, generalizing subroutines for cooperative multitasking. Coroutines are well-suited for implementing familiar program components such as cooperative tasks, exceptions, event loops, iterators, infinite lists and pipes.
- A typical use of coroutining is in the Hamming problem: Generate, in increasing order, the sequence of all i positive integers whose prime divisors are a subset of $\{2, 3, 5\}$, i.e. $i = 2^j * 3^k * 5^l$, $j, k, l \geq 0$
- A simplified version is presented: the only divisors allowed are 2 and 3
- The solution is based on the dataflow diagram below:



Example 6: Solving the Hamming problem via coroutining

```
% times(As, M, Bs): List Bs is obtained by multiplying each element of As by M
:- block times(-, ?, ?).                % blocks if the 1st arg is a variable.
times([A|As], M, Bs) :-
    B is M*A, Bs = [B|BBs], times(As, M, BBs).
times([], _, []).

% merge(As, Bs, Cs): Sorted list Cs is obtained by
% collating sorted lists As and Bs, removing duplicates
:- block merge(-, ?, ?), merge(?, -, ?). % blocks if 1st or 2nd arg is a var.
merge([A|As], [B|Bs], Cs) :-
    ( /*if*/    A < B -> /*then*/ Cs = [A|Ds], merge(As, [B|Bs], Ds)
    ; /*elif*/  A > B -> /*then*/ Cs = [B|Ds], merge([A|As], Bs, Ds)
    ; /*else*/                    Cs = [A|Ds], merge(As,     Bs, Ds)
    ).
merge([], Bs, Bs).
merge(As, [], As).

% U is the list of the first N (2,3)-Hamming numbers
hamming(N, U) :-
    U = [1|_H], times(U, 2, X), times(U, 3, Y), merge(X, Y, Z),
    prefix_length([1|Z], U, N).          % A predicate from library(lists)
                                         % prefix_length(L, P, N): L has a prefix P of length N
```


II. rész

Haladó Prolog

Tartalom

Blokkolás, korutinszervezés

- Blokk-deklarációk SICStusban
  - Egy eljárásra előírhatjuk, hogy mindaddig, amíg egy ún. blokkolási feltétel fennáll, az eljárás függesztődjék fel.
  - Példa:

    `:- block p(-, ?, -, ?, ?).`
  - Jelentése: ha az első és a harmadik argumentum is behelyettesítetlen változó (blokkolási feltétel), akkor a `p/5` hívás felfüggesztődik.
  - Ugyanarra az eljárásra több vagylagos feltétel is szerepelhet, pl.
    `:- block p(-, ?), p(?, -).`
    (`p/2` felfüggesztődik, ha bármelyik argumentuma behelyettesítetlen.)
- Blokk-deklarációk haszna
  - Adatfolyam-programozás (lásd Hamming probléma, Prolog jegyzet)
  - Generál és ellenőriz programok gyorsítása
  - Végtelen választási pontok kiküszöbölése

Listák biztonságos összefűzése blokk-deklaráció segítségével

```
:- block app(-, ?, -).
% blokkol, ha az első és a harmadik argumentum
% egyaránt behelyettesítetlen
app([], L, L).
app([X|L1], L2, [X|L3]) :-
    app(L1, L2, L3).


| ?- app(L1, L2, L3).
user:app(L1,L2,L3) ? ;
no
| ?- app(L1, L2, L3), L3 = [a|L4].
L1 = [], L2 = [a|L4], L3 = [a|L4] ? ;
L1 = [a|_A], L3 = [a|L4], user:app(_A,L2,L4) ? ;
no
```


Listák biztonságos összefűzése, nyomkövetés

```
| ?- trace, app(L1, L2, L3), L3 = [a|L4], L4 = [].
% The debugger will first creep -- showing everything (trace)
        -      - Block: app(_1012,_532,_1018)
        1      1 Call: _1018=[a|_622] ?
        -      - Unblock: app(_1012,_532,[a|_622])
        2      2 Call: app(_1012,_532,[a|_622]) ?
?       2      2 Exit: app([],[a|_622],[a|_622]) ?
?       1      1 Exit: [a|_622]=[a|_622] ?
        3      1 Call: _622=[] ?
        3      1 Exit: []=[] ?
L1 = [], L2 = [a], L3 = [a], L4 = [] ? ;
        1      1 Redo: [a|_622]=[a|_622] ?
        2      2 Redo: app([],[a|_622],[a|_622]) ?
        -      - Block: app(_2098,_532,_2104)
        2      2 Exit: app([a|_2098],_532,[a|_2104]) ? &
Blocked goals:
1 (_2098): user:app(_2098,_532,_2104)
2 (_2104): user:app(_2098,_532,_2104)
        2      2 Exit: app([a|_2098],_532,[a|_2104]) ?
        1      1 Exit: [a|_2104]=[a|_2104] ?
        4      1 Call: _2104=[] ?
        -      - Unblock: app(_2098,_532,[])
        5      2 Call: app(_2098,_532,[]) ?
?       5      2 Exit: app([],[],[]) ?
?       4      1 Exit: []=[] ?
L1 = [a], L2 = [], L3 = [a], L4 = [] ? ;
        4      1 Redo: []=[] ?
        5      2 Redo: app([],[],[]) ?
        5      2 Fail: app(_2098,_532,[]) ?
        4      1 Fail: _2104=[] ?
no
```

Példa korutinszervezésre: többirányú összeadás

```
% plusz(X, Y, Z): X+Y=Z,  ahol X, Y és Z természetes számok.
% Bármelyik argumentum lehet behelyettesítetlen.
plusz(X, Y, Z) :-
        app(A, B, C), len(A, X), len(B, Y), len(C, Z).

% L hossza Len.
len(L, Len) :- len(L, 0, Len).

:- block len(-, ?, -).
% L lista hossza Len-Len0. Len0 mindig ismert.
len(L, Len0, Len) :-
        nonvar(Len), !, Len1 is Len-Len0, Len1 >= 0, length(L, Len1).
len([_|L], Len0, Len) :-
        Len1 is Len0+1, len(L, Len1, Len).
len([], Len, Len).
```


Példa korutinszervezésre: többirányú összeadás

```
| ?- plusz(X, Y, 2).
X = 0, Y = 2 ? ;
X = 1, Y = 1 ? ;
X = 2, Y = 0 ? ;
no
| ?- plusz(X, X, 8).
X = 4 ? ;
no
| ?- plusz(X, 1, Y), plusz(X, Y, 22).
no
| ?- plusz(2, A, B).
user:len(_A,0,A),          % van egy _A lista, melynek hossza A
user:len(_A,2,B) ?         % és      _A                 hossza B-2
                           % VAGYIS: A = B-2
                     ;
no
```


Korutinszervezés – hívások késleltetése

- `freeze(X, Hivas)`
  `Hivas`t felfüggeszti mindaddig, amíg `X` behelyettesítetlen változó.
- `dif(X, Y)`
  `X` és `Y` nem egyesíthető. Mindaddig felfüggesztődik, amíg ez el nem dönthető.
- `when(Feltétel, Hívás)`
  Blokkolja a `Hívás`t mindaddig, amíg a `Feltétel` igazzá nem válik. Itt a `Feltétel` egy (nagyon) leegyszerűsített Prolog cél, amelynek szintaxisa:

```
CONDITION ::=  nonvar(X) | ground(X) | ?=(X,Y) |
               CONDITION, CONDITION |
               CONDITION; CONDITION
```

  `ground(X)` jelentése: `X` tömör – nincs benne (behelyettesítetlen) változó
  `?=(X,Y)` jelentése: `X` és `Y` egyesíthetősége eldönthető

Korutinszervezés – hívások késleltetése

- Példa (`process` csak akkor hívódik meg, ha `T` tömör, és vagy `X` nem változó, vagy `X` és `Y` egyesíthetősége eldönthető):

```
| ?- when( ((ground(T),nonvar(X);?=(X,Y))),
           process(X,Y,T)).
```

- A `dif` eljárás a `when` segítségével definiálható:

```
dif(X, Y) :- when(?=(X,Y), X\=Y).
```

Korutinszervezés – késleltetett hívások lekérdezése

- `frozen(X, Hivas)`
  Az `X` változó miatt felfüggesztett hívás(oka)t egyesíti `Hivas`-sal.
- `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)) ?
```


Többirányú összeadás `when` segítségével

```
:- use_module(library(between)).

% app(L1, L2, L3): L1 és L2 összefűzöttje L3.
% ahol L1, L2 és L3 1-es számokból álló listák.
app([], L, L).
app([1|L1], L2, [1|L3]) :-
    when((nonvar(L1);nonvar(L3)),
         app(L1, L2, L3)).

len(L, Len) :-
        when(ground(L), length(L, Len)),
        when(nonvar(Len), findall(1, between(1, Len, _), L)).

% X+Y=Z, ahol X, Y és Z természetes számok.
% Bármelyik argumentum lehet behelyettesítetlen.
plusz(X, Y, Z) :-
        app(A, B, C),
        len(A, X),
        len(B, Y),
        len(C, Z).
```


Többirányú összeadás `when` segítségével

```
| ?- plusz(X, Y, 2).
X = 0, Y = 2 ? ;
X = 1, Y = 1 ? ;
X = 2, Y = 0 ? ;
no
| ?- plusz(X, X, 8).
X = 4 ? ;
no
| ?- plusz(X, 1, Y), plusz(X, Y, 20).
no
| ?- plusz(2, A, B).
prolog:trig_ground(_A,[],[_A],_B,_B),
prolog:when(_B,ground(_A),user:length(_A,A)),
prolog:when(A,nonvar(A),user:findall(1,between(1,A,_C),_A)),
prolog:trig_ground(_A,[],[_A],_D,_D),
prolog:when(_D,ground([1,1|_A]),user:length([1,1|_A],B)),
prolog:when(B,nonvar(B),user:findall(1,between(1,B,_E),[1,1|_A])) ?
no
```


Tartalom

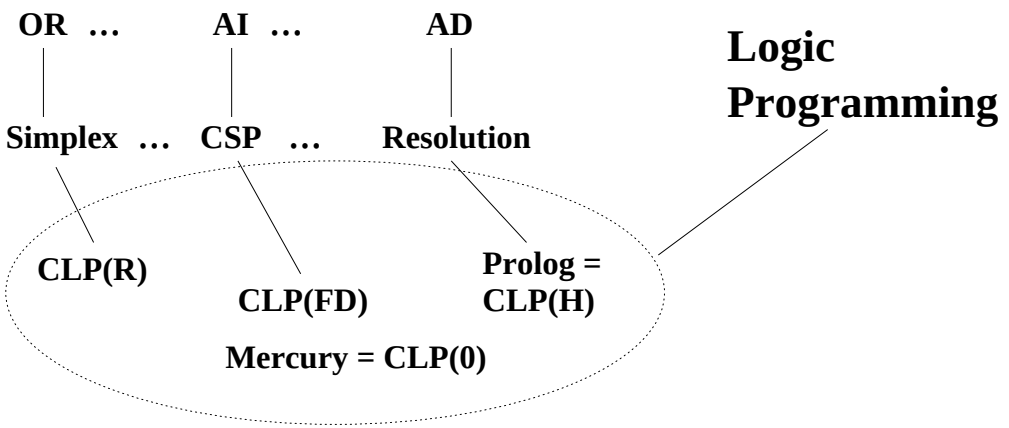
A CLP alapgondolata

- A CLP($\mathcal{X}$) séma

  Prolog + egy valamilyen $\mathcal{X}$ adattartományra és azon értelmezett korlátokra (relációkra) vonatkozó „erős” következtetési mechanizmus

- Példák az $\mathcal{X}$ tartomány megválasztására
  - $\mathcal{X} =$ Q vagy R (a racionális vagy valós számok)
    korlátok: lineáris egyenlőségek és egyenlőtlenségek
    következtetési mechanizmus: Gauß elimináció, szimplex módszer
  - $\mathcal{X} =$ FD (egész számok Véges Tartománya, FD — Finite Domain)
    korlátok: különféle aritmetikai és kombinatorikus relációk
    következtetési mechanizmus: MI CSP–módszerek (CSP = Korlát-Kielégítési Probléma)
  - $\mathcal{X} =$ B (0 és 1 Boole értékek)
    korlátok: ítéletkalkulusbeli relációk
    következtetési mechanizmus: MI SAT-módszerek (SAT — Boole kielégíthetőség)

A CLP mint integrációs paradigma



Példa: CLP(MiniNat)

- Egy miniatűr kvázi-CLP nyelv természetes számokra
  (Motiváció: a CLP alapelvek és egyben a haladó Prolog lehetőségek bemutatása.)
  - Tartomány: Nem negatív egészek
  - Függvények:

    `+ - *`
  - Korlát-relációk:

    `= < > =< >=`
  - Korlát-megoldó algoritmus:

    a Prolog korutin-kiterjesztésén alapul
- A Prologba ágyazás szintaxisa:

  {*Korlát*} a *Korlát* felvétele
  (`{X}` szintaktikus édesítőszer, ekvivalens a `'{}'(X)` kifejezéssel.)

Példa: CLP(MiniNat)

Példafutás

```
| ?- {X+Y = 2}.
X = 2, Y = 0 ? ;
X = 1, Y = 1 ? ;
X = 0, Y = 2 ? ;
no
| ?- {2*X+3*Y=8}.
X = 4, Y = 0 ? ;
X = 1, Y = 2 ? ;
no
| ?- {X*2+1=28}.
no
| ?- {X*X+Y*Y=25, X > Y}.
X = 5, Y = 0 ? ;
X = 4, Y = 3 ? ;
no
```

CLP(MiniNat) megvalósítása – számábrázolás

- A korábbi `plusz/3` eljárásokban egy *N* elemű listával ábrázoltuk az *N* számot (a listaelemek érdektelenek, behelyettesítetlen változók vagy 1-esek)
- Példa: a 2 szám ábrázolása: `[_,_]` ≡ `.(_,.(_,[])).`
- Hagyjuk el a felesleges listaelemeket, akkor a 2 szám ábrázolása: `.(.([])).`
- Itt a `[]` jelenti a 0 számot, a `.(X)` struktúra az `X` szám rákövetkezőjét (a nála 1-gyel nagyobb számot).
- Ez tulajdonképpen a Peano féle számábrázolás, ha a `. /1` helyett az `s/1` funktort, a `[]` helyett a `0` konstanst használjuk.
- A CLP(MiniNat) megvalósításában a Peano számábrázolást használjuk, tehát; `0 = 0; 1 = s(0); 3 = s(s(s(0)))` stb.

CLP(MiniNat) megvalósítása – összeadás és kivonás

```
% plusz(X, Y, Z): X+Y=Z (Peano számokkal).
:- block plusz(-, ?, -).
plusz(0, Y, Y).
plusz(s(X), Y, s(Z)) :-
      plusz(X, Y, Z).

% +(X, Y, Z): X+Y=Z (Peano számokkal). Hatékonyabb, mert
% továbblép, ha bármelyik argumentum behelyettesített.
:- block +(-, -, -).
+(X, Y, Z) :-
        nonvar(Y), !, plusz(Y, X, Z).
+(X, Y, Z) :-
        /* var(Y), */ plusz(X, Y, Z). % \+((var(X),var(Z)))

% X-Y=Z (Peano számokkal).
-(X, Y, Z) :-
       +(Y, Z, X).
```


CLP(MiniNat) – a szorzás művelet megvalósítási elvei

- Felfüggesztjük mindaddig, míg legalább egy tényező vagy a szorzat ismertté nem válik.
- Ha az egyik tényező ismert, visszavezetjük ismételt összeadásra.
- Ha a szorzat ismert (*N*), az egyik tényezőre végigpróbáljuk az 1,2,. . . *N* értékeket, ezáltal ismételt összeadásra visszavezethetővé tesszük.

CLP(MiniNat) megvalósítása – szorzás

```
% X*Y=Z. Blokkol, ha nincs tömör argumentuma.
*(X, Y, Z) :-
       when( (ground(X);ground(Y);ground(Z)),
             szorzat(X, Y, Z)).

% X*Y=Z, ahol legalább az egyik argumentum tömör.
szorzat(X, Y, Z) :-
       (   ground(X) -> szor(X, Y, Z)
       ;   ground(Y) -> szor(Y, X, Z)
       ;   /* Itt már Z biztosan tömör a fenti when miatt! */
           Z == 0 -> szorzatuk_nulla(X, Y)
       ;   X = s(_), +(X, _, Z),
         % X >= 1,     X =< Z,      %  vö. between(1, Z, X)
           szor(X, Y, Z)
       ).

% X*Y=0.
szorzatuk_nulla(X, Y) :-
       (   X = 0
       ;   dif(X, 0), Y = 0
       ).

% szor(X, Y, Z): X*Y=Z, X tömör.
% Y-nak az (ismert) X-szeres összeadása adja ki Z-t.
szor(0, _Y, 0).
szor(s(X), Y, Z) :-
       szor(X, Y, Z1),
       +(Z1, Y, Z).
```

CLP(MiniNat) – példa

- Tekintsük az `X*Y =< 2` egyenlőtlenséget (a természetes számokon)
- Szorzásra használjuk a `* /3` predikátumot: `*(X, Y, XszorY)`
- Az összehasonlításra használható a `+ /3` predikátum: `+(XszorY, _, s(s(0)))`
- Hány megoldása van az egyenletnek?
- Hány megoldást kapunk?

```
| ?- *(X, Y, _XszorY),+(_XszorY, _, s(s(0))).
X = 0 ? ;
Y = 0, prolog:dif(X,0) ? ;
X = 1, Y = 1 ? ;
X = 1, Y = 2 ? ;
X = 2, Y = 1 ? ;
no
```


CLP(MiniNat) korlátok fordítása Prolog célokká

- A funkcionális alakban megadott korlátokat a `+ /3, - /3, * /3` hívásokból álló célsorozattá alakítjuk, majd ezt a célsorozatot meghívjuk.
- Például a `2*Z = X*Y+2` korlát fordítása:
  - a bal oldal értéke az `A` változóban: `*(s(s(0)),Z,A)`
  - a jobb oldal értéke, szintén az `A` változóban: `*(X,Y,B),+(B,s(s(0)),A)`
  - a teljes korlát fordítása: `*(s(s(0)),Z,A), *(X,Y,B),+(B,s(s(0)),A)`
- Egyenlőtlenségek (<, =<, >, >=) esetén kiszámoljuk a bal és jobb oldal értékét (`B` és `J`), majd
  - a `{B =< J}` korlátot a `{B+_ = J}` korlátra,
  - a `{B < J}` korlátot a `{B+s(_) = J}` korlátra vezetjük vissza
- A Prolog listához hasonló szintaktikus „édesítőszer”: `{Kif}` ≡ `'{}'(Kif)`
- A beépített eljárásoktól (pl. egyesítés: `A = B`) való megkülönböztetés miatt a korlátokat kapcsos zárójelbe tesszük pl. `{Z=X+1,T=Z*Z}`

CLP(MiniNat) megvalósítása – korlátok fordítása

```
% {Korlat}: Korlat fennáll.
{Korlat} :-
        korlat_cel(Korlat, Cel), call(Cel).


% korlat_cel(Korlat, Cel): Korlat végrehajtható
% alakja a Cel célsorozat.
korlat_cel(Kif1=Kif2, (C1,C2)) :-
        kiertekel(Kif1, E, C1), % Kif1 értékét E-ben
                                % előállító cél C1
        kiertekel(Kif2, E, C2).
korlat_cel(Kif1 =< Kif2, Cel) :- korlat_cel(Kif1+_ =  Kif2, Cel).
korlat_cel(Kif1 < Kif2, Cel)  :- korlat_cel(Kif1+1 =< Kif2, Cel).
korlat_cel(Kif1 >= Kif2, Cel) :- korlat_cel(Kif2   =< Kif1, Cel).
korlat_cel(Kif1 > Kif2, Cel)  :- korlat_cel(Kif2   <  Kif1, Cel).
korlat_cel((K1,K2), (C1,C2)) :-
        korlat_cel(K1, C1), korlat_cel(K2, C2).
```


CLP(MiniNat) megvalósítása – kifejezések fordítása

```
% kiertekel(Kif, E, Cel): A Kif aritmetikai kifejezés
% értékét E-ben előállító cél Cel.
% Kif egészekből és változókból
% a +, -, és * operátorokkal épül fel.
```

- Egy `Kif1` *Op* `Kif2` kifejezés lefordított alakja egy max. három részből álló célsorozat, amely az `E` változóba helyezi a kifejezés eredményét, pl.:

```
| ?- kiertekel((X+1)*(X-1), E, Cel).
        ⟹   Cel = (+(X,1,E1),-(X,1,E2),*(E1,E2,E)) ? ; no
```

  - első, opcionális rész: `Kif1` értékét pl. `E1`-ben előállító cél(sororzat).
  - második, opc. rész: `Kif2` értékét pl. `E2`-ben előállító cél(sororzat).
  - harmadik rész: *Op*`(E1, E2, E)` (ahol *Op* `+`, `-`, vagy `*`).
- Egy szám lefordított formája a szám Peano alakja.
- Minden egyéb (változó, vagy már Peano alakú szám) változatlan marad a fordításkor.

CLP(MiniNat) megvalósítása – kifejezések fordítása

```
% kiertekel(Kif, E, Cel): A Kif aritmetikai kifejezés
% értékét E-ben előállító cél Cel.
% Kif egészekből a +, -, és * operátorokkal épül fel.
kiertekel(Kif, E, Cel) :-
        (   compound(Kif), Kif =.. [Op,Kif1,Kif2]
        ->  Cel = (C1,C2,Rel),
            Rel =.. [Op,E1,E2,E],
            kiertekel(Kif1, E1, C1),
            kiertekel(Kif2, E2, C2)
        ;   integer(Kif)
        ->  Cel = true, int_to_peano(Kif, E)
        ;   Cel = true, E = Kif
        ).

% int_to_peano(N, P): N természetes szám Peano alakja P.
int_to_peano(N, P) :-
        (   N > 0 -> N1 is N-1, P = s(P1),
            int_to_peano(N1, P1)
        ;   N = 0, P = 0
        ).
```


Prolog háttér: kifejezések testreszabott kiírása

A `print(Kif)` beépített eljárás

Alapértelmezésben azonos `write(Kif)`-fel. Ha a felhasználó definiál egy `portray/1` eljárást, akkor a `print` a kinyomtatandó kifejezésre meghívja a `portray` callback eljárást. Ennek sikere esetén feltételezi, hogy a kiírás megtörtént, meghiúsulás esetén maga kezdi el kiírni a kifejezést, de ennek részkifejezéseire rekurzívan újra meghívja a `portray` callback eljárást.
A rendszer a `print` eljárást használja a változó-behelyettesítések és a nyomkövetés kiírására is!

Prolog háttér: kifejezések testreszabott kiírása, folyt.

Példa: mátrixok kiíratása

```
portray(Matrix) :-
     % Durva közelítés: mátrixnak tekintünk egy kifejezést,
     % ha olyan lista, melynek első eleme nem-üres lista:
     Matrix = [[_|_]|_],
     (   member(Row, Matrix), nl, print(Row), fail
     ;   true
     ).

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


Prolog háttér: kifejezések testreszabott kiírása, folyt. 2

A `portray(Kif)` ún. *kampó* eljárás (callback/hook predicate)

Értelemszerűen a `portray/1` klózait a felhasználónak kell definiálnia.
Ha az adott klóz „érintve érzi magát”, akkor kiírja a `Kif` kifejezést, és sikerrel tér vissza, ezzel jelezve, hogy a Prolog rendszernek nem kell kiírnia.
A `portray/1` ún. `multifile` eljárás, azaz klózai több fájlban elszórva lehetnek, meghiúsulás esetén az összes `portray` klózt végigpróbálja a rendszer, az első sikeres lefutásig.
Ha a `portray` hívás meghiúsul, akkor a `print/1`

- egyszerű kifejezés (szám, atom vagy változó) esetén kiírja a kif.-t;
- összetett kifejezés esetén kírja a struktúranevet, zárójeleket, vesszőket, de az argumentumokra megint meghívja a `portray`-t.
- ha operátoros összetett kifejezésről van szó, akkor az operátort a print írja ki, de az argumentumok esetén ismét próbálkozik a `portray`-val.
- analóg a helyzet a listák esetén is: itt a listaelemekre újra meghívódik a `portray`.

Példa testreszabott kiíratásra: Peano számok

```
% Peano számok kiírásának formázása
user:portray(Peano) :-
        peano_to_int(Peano, 0, N), write(N).

% A Peano Peano-szám értéke N-N0.
peano_to_int(Peano, N0, N) :- nonvar(Peano),
        (   Peano == 0 -> N = N0
        ;   Peano = s(P),
            N1 is N0+1,
            peano_to_int(P, N1, N)
        ).

% Célok kiíratásának formázása
user:portray(mininat:Rel) :-
        Rel =.. [Pred,A,B,C],
        predikatum_operator(Pred, Op),
        Fun =.. [Op,A,B],
        print({Fun=C}).

predikatum_operator(plusz, +).
predikatum_operator(+, +).
predikatum_operator(-, -).
predikatum_operator(*, *).
```


CLP(MiniNat) használata — példák

```
:- block fact0(-,-). % csak akkor fut ha ismert N vagy F.
fact0(N, F) :-
        {N = 0, F = 1}.
fact0(N, F) :-
        {N1 = N-1},
        fact0(N1, F1),
        {F = N*F1}.

| ?- {X*X+Y*Y=25, X>Y}.
X = 4, Y = 3 ? ;
X = 5, Y = 0 ? ; no

| ?- fact0(6, F).
F = 720 ? ; no

| ?- fact0(8, F).
F = 40320 ? ; no

| ?- fact0(N, 6).
N = 3 ? ; no

| ?- fact0(N, 24).
N = 4 ? ;
! Resource error: insufficient memory

| ?- fact0(N, 11).
no

| ?- fact0(N, 17).
! Resource error: insufficient memory
```


Prolog háttér: programok előfeldolgozása

Kampó (Hook, callback) eljárások a fordítási idejű átalakításhoz:

- `user:term_expansion(+Kif, ..., -Klózok, ...)`: (közelítő leírás:) Minden betöltő eljárás (`consult`, `compile` stb.) által beolvasott kifejezésre a rendszer meghívja. A kimenő paraméterben várja a transzformált alakot (lehet lista is). Meghiúsulás esetén változtatás nélkül veszi fel a kifejezést klózként.
- `M:goal_expansion(+Cél, +Layout, +Modul, -ÚjCél, -ÚjLayout)`: Minden a beolvasott programban (vagy feltett kérdésben) előforduló részcélra meghívja a rendszer. A kimenő paraméterekben várja a transzformált alakot (lehet konjunkció). Meghiúsulás esetén változtatás nélkül hagyja a célt. (Ha a forrásszintű nyomkövetés nem fontos, `ÚjLayout` lehet `[]`.)

CLP(MiniNat) továbbfejlesztése `goal_expansion` használatával

- A funkcionális alak átalakítása a betöltés alatt is elvégezhető (kompilálás):

```
goal_expansion({Korlat}, _L0, _M, Cel, /*ÚjLayout:*/ []) :-
        korlat_cel(Korlat, Cel).
```

- Célszerű a generált célsorozatból a `true` hívásokat kihagyni.

```
% osszetett(C1, C2, C): C a C1 és C2 célok konjunkciója.
osszetett(true, Cel0, Cel) :-    !, Cel = Cel0.
osszetett(Cel0, true, Cel) :-    !, Cel = Cel0.
osszetett(Cel1, Cel2, (Cel1,Cel2)).
```

- A fenti eljárást használjuk a konjunkciók helyett, pl:

```
korlat_cel((K1,K2), C12) :-
         korlat_cel(K1, C1), korlat_cel(K2, C2),
         osszetett(C1, C2, C12).
```

Megjegyzés: a faktoriális példában ez a kompilálás 6-7% gyorsulást jelent

Előfeldolgozás a faktoriális példa esetén

- A faktoriális példa betöltött alakja :

```
fact(0, s(0)).
fact(N, F) :-
        +(s(0), _, N),    % N >= 1
        -(N, s(0), N1),   % N1 = N-1
        *(N, F1, F),      % F = N*F1
        fact(N1, F1).
```

- Vigyázat! Az így előálló kód már nem foglalkozik a számok Peano-alakra hozásával:

```
| ?- fact(N, 6).          -->  no
| ?- {F=6}, fact(N, F).   -->  F = 6, N = 3 ? ; no
```


Tartalom

1. kis házi feladat: CLP(MiniB) megvalósítása

CLP(MiniB) jellemzése

- **Tartomány:** logikai értékek (`1` és `0`, igaz és hamis)
- **Függvények** (egyben korlát-relációk):

`~ P`	`P` hamis (*negáció*).
`P * Q`	`P` és `Q` mindegyike igaz (*konjunkció*).
`P + Q`	`P` és `Q` legalább egyike igaz (*diszjunkció*).
`P # Q`	`P` és `Q` pontosan egyike igaz (*kizáró vagy*).
`P =\= Q`	Ugyanaz mint `P # Q`.
`P =:= Q`	Ugyanaz mint `~(P # Q)`.

- A fenti függvényjelek többsége szabványos beépített operátor (ezek prioritását nem célszerű módosítani), a `~` és `#` operátorokat – a CLP(B) könyvtárral megegyezően – az alábbi módon javasoljuk deklarálni:

```
:- op(300, fy, ~).
:- op(500, yfx, #).
```


1. kis házi feladat: CLP(MiniB) megvalósítása

A megvalósítandó eljárások

- `sat(`*Kif*`)`, ahol *Kif* változókból, a `0`, `1` konstansokból a fenti müveletekkel felépített logikai kifejezés. Jelentése: A *Kif* logikai kifejezés igaz. A `sat/1` eljárás ne hozzon létre választási pontot! A benne szereplő változók behelyettesítése esetén minél előbb ébredjen fel, és végezze el a megfelelő következtetéseket (lásd a példákat alább)!
- `count(`*Es*`, `*N*`)`, ahol *Es* egy (változó-)lista, *N* adott természetes szám. Jelentése: Az *Es* listában pontosan *N* olyan elem van, amelynek értéke 1.
- `labeling(`*Változók*`)`. Behelyettesíti a *Változók*at `0`, `1` értekekre. Visszalépés esetén felsorolja az összes lehetséges értéket.

1. kis házi feladat: egy kis segítség

Mikor érdemes a korlátokat felébreszteni?

- Ha egy változó (pl. `A`) behelyettesített: `nonvar(A)` ébresztési feltétel
- Ha van két változó (pl. `A` és `Res`) amelyek azonossága eldönthető: `?=(A,Res)` ébresztési feltétel

```
~(A, Res) :-
        when( (nonvar(A); nonvar(Res); ?=(A,Res)),
              not(A,Res)
            ).
not(A, Res) :-
        (   nonvar(A) -> Res is 1-A       % nonvar(A) ébresztés
        ;   nonvar(Res) -> A is 1-Res     % nonvar(Res) ébresztés
        ;   A == Res -> fail              % ?=(A,Res) ébresztés
        ).
```

- A kétargumentumú műveletekből generált korlátok esetén (pl. `#(A,B,Res)`), a `when` részben 3 `nonvar` és 3 `?=` feltétel szükséges.
- **Fontos:** A `count` eljárás esetében nem várjuk el az argumentumlistában levő változók azonosság-vizsgálatát (csak `nonvar` feltételek kellenek.)

1. kis házi feladat

Futási példák

```
| ?- sat(A*B =:= (~A)+B).
                          --->   <...felfüggesztett célok...> ? ; no
| ?- sat(A*B =:= (~A)+B), labeling([A,B]).
                          --->   A = 1, B = 0 ? ; A = 1, B = 1 ? ; no
| ?- sat((A+B)*C=\=A*C+B), sat(A*B).
                          --->   A = 1, B = 1, C = 0 ? ; no
| ?- sat(~A =:= A).       --->   no

| ?- count([A,A,B], 2). --->   <...felfüggesztett célok...> ? ; no
| ?- count([A,A,B], 2), labeling([A]).
                          --->   A = 1, B = 0 ? ; no
| ?- count([A,A,B,B], 3), labeling([A,B]).
                          --->   no
```


1. kis házi feladat, példák (folyt.)

```
| ?- trace, ~(A, A).
 1 1 Call: ~(A,A) ?
 2 2 Call: when((nonvar(A);nonvar(A);?=(A,A)),not(A,A))?
 3 3 Call: not(A,A) ?
 4 4 Call: nonvar(A) ?
 4 4 Fail: nonvar(A) ?
 5 4 Call: nonvar(A) ?
 5 4 Fail: nonvar(A) ?
 6 4 Call: A==A ?
 6 4 Exit: A==A ?
 3 3 Fail: not(A,A) ?
 2 2 Fail: when((nonvar(A);nonvar(A);?=(A,A)),not(A,A))?
 1 1 Fail: ~(A,A) ?
no
| ?- sat(A*A=:=B).
                              B = A ? ; no
| ?- sat(A#A=:=B).
                              B = 0 ? ; no
| ?- sat(A+B=:=C), A=B.
                              B = A, C = A ? ; no
```


III. rész

A SICStus clp(Q,R) könyvtárai

A `clpq`/`clpr` könyvtárak

- Tartomány:
  - `clpr`: lebegőpontos számok
  - `clpq`: racionális számok
- Függvények:
  `+ - * / min max pow exp` (kétargumentumúak, `pow` $\equiv$ `exp`),
  `+ - abs sin cos tan` (egyargumentumúak).
- Korlát-relációk:
  `= =:= < > =< >= =\=` (`=` $\equiv$ `=:=`)
- Primitív korlátok (korlát tár elemei):
  lineáris kifejezéseket tartalmazó relációk
- Korlát-megoldó algoritmus:
  lineáris programozási módszerek: Gauss elimináció, szimplex módszer

A `clpq`/`clpr` könyvtárak

A könyvtár betöltése:

- `use_module(library(clpq))`, vagy
- `use_module(library(clpr))`

A fő beépített eljárás:

- `{` *Korlát* `}` , ahol *Korlát* változókból és (egész vagy lebegőpontos) számokból a fenti műveletekkel felépített reláció, vagy ilyen relációknak a vessző (`,`) operátorral képzett konjunkciója.

A korlát-tár

- A CLP(X) séma általános adatstruktúrája
- A futás adott pillanatáig beérkezett ún. primitív korlátokat tárolja
- Ha a tárbeli korlátok ellentmondásosak, visszalépés történik (azaz előremenő végrehajtás esetén garantált a tár konzisztenciája)
- Az ún. összetett korlátok nem kerülnek be a tárba

Példafutás a SICStus `clpq` könyvtárával

```
| ?- use_module(library(clpq)).
{loading .../library/clpq.ql...}
...

| ?- {X=Y+4, Y=Z-1, Z=2*X-9}.
X = 6, Y = 2, Z = 3 ?    % lineáris egyenlet

| ?- {X+Y+9<4*Z, 2*X=Y+2, 2*X+4*Z=36}.
                         % lineáris egyenlőtlenség
{X<29/5}, {Y= -2+2*X}, {Z=9-1/2*X} ?
                         % az eredmény: ekvivalens alak,
                         % de látható, hogy ellentmondásmentes
| ?- {(Y+X)*(X+Y)/X = Y*Y/X+100}.
{X=100-2*Y} ?            % lineárissá egyszerűsíthető

| ?- {(Y+X)*(X+Y) = Y*Y+100*X}.
                         % így már nem lineáris
clpq:{2*(X*Y)-100*X+X^2=0} ?
                         % a clpq modul-prefix jelzi,
                         % hogy felfüggesztett összetett
                         % hívásról van szó
```


Példafutás a SICStus `clpq` könyvtárával

```
| ?- {exp(X+Y+1,2) = 3*X*X+Y*Y}.
                         % nem lineáris...
clpq:{1+2*X+2*(Y*X)-2*X^2+2*Y=0} ?

| ?- {exp(X+Y+1,2) = 3*X*X+Y*Y}, X=Y.
X = -1/4, Y = -1/4 ?     % így már igen...

| ?- {2 = exp(8, X)}.     % nem-lineárisak is
                         % megoldhatók
X = 1/3 ?
```

Összetett korlátok kezelése CLP(Q)-ban

Példa várakozó ágensre

```
| ?- {X =< Y}, {X*(Y+1) > X*X+Z},
       (   Z = X*(Y-X), {Y < 0}
       ;   Y = X
       ).                          Y = X,  {X-Z>0} ? ; no
```

A végrehajtás lépései

```
| ?- {X =< Y}, {X*(Y+1) > X*X+Z}.
                         {X-Y=<0}, clpq:{Z-X-Y*X+X^2<0} ?

| ?- {X =< Y}, {X*(Y+1) > X*X+Z}, Z = X*(Y-X).
                         Z = X*(Y-X), {X-Y=<0}, {X>0} ?

| ?- {X =< Y}, {X*(Y+1) > X*X+Z}, Z = X*(Y-X), {Y < 0}.
                         no

| ?- {X =< Y}, {X*(Y+1) > X*X+Z}, Y = X.
                         Y = X, {X-Z>0} ?
```


Példa egy *lehetséges* erősítési lépésre

- A tár tartalma: `X > 3`.
- A végrehajtandó összetett korlát: `Y > X*X`.
- A korlátot a CLP megoldó nem tudja felvenni a tárba, de egy *következményét*, pl. az `Y > 9` korlátot felvehetné!
- Az erősítés után az eredeti összetett korlát továbbra is démonként kell lebegjen!
- **Fontos megjegyzés:** a CLP(Q/R) rendszer **nem** hajtja végre a fenti következtetést, és semmiféle erősítést nem végez.

Egy összetettebb példa: hiteltörlesztés

```
% Hiteltörlesztés számítása: P összegű hitelt
% Time hónapon át évi IntRate kamat mellett havi MP
% részletekben törlesztve Bal a maradványösszeg.
mortgage(P, Time, IntRate, Bal, MP):-
     {Time > 0, Time =< 1,
     Bal = P*(1+Time*IntRate/1200)-Time*MP}.
mortgage(P, Time, IntRate, Bal, MP):-
     {Time > 1},
     mortgage(P*(1+IntRate/1200)-MP,
              Time-1, IntRate, Bal, MP).


| ?- mortgage(100000,180,12,0,MP).
                         % 100000 Ft hitelt 180
                         % hónap alatt törleszt 12%-os
                         % kamatra, mi a havi részlet?
MP = 1200.1681 ?
```


Egy összetettebb példa: hiteltörlesztés

```
| ?- mortgage(P,180,12,0,1200).
                         % ugyanez visszafelé
P = 99985.9968 ?

| ?- mortgage(100000,Time,12,0,1300).
                         % 1300 Ft a törlesztőrészlet,
                         % mi a törlesztési idő?
Time = 147.3645 ?

| ?- mortgage(P,180,12,Bal,MP).

{MP=0.0120*P-0.0020*Bal} ?

| ?- mortgage(P,180,12,Bal,MP), ordering([P,Bal,MP]).

{P=0.1668*Bal+83.3217*MP} ?
```

További könyvtári eljárások

- `entailed(Korlát)` — `Korlát` levezethető a jelenlegi tárból.
- `inf(Kif, Inf)` ill. `sup(Kif, Sup)` — kiszámolja `Kif` infímumát ill. szuprémumát, és egyesíti `Inf`-fel ill. `Sup`-pal. Példa:

```
| ?- { 2*X+Y =< 16, X+2*Y =< 11, X+3*Y =< 15 },
     sup(30*X+50*Y, Sup).

Sup = 310, {....}
```

- `minimize(Kif)` ill. `maximize(Kif)` — kiszámolja `Kif` infimumát ill. szuprémumát, és egyenlővé teszi `Kif`-fel. Példa:

```
| ?- { 2*X+Y =< 16, X+2*Y =< 11, X+3*Y =< 15,
       Z = 30*X+50*Y
     }, maximize(Z).

X = 7, Y = 2, Z = 310
```


További könyvtári eljárások

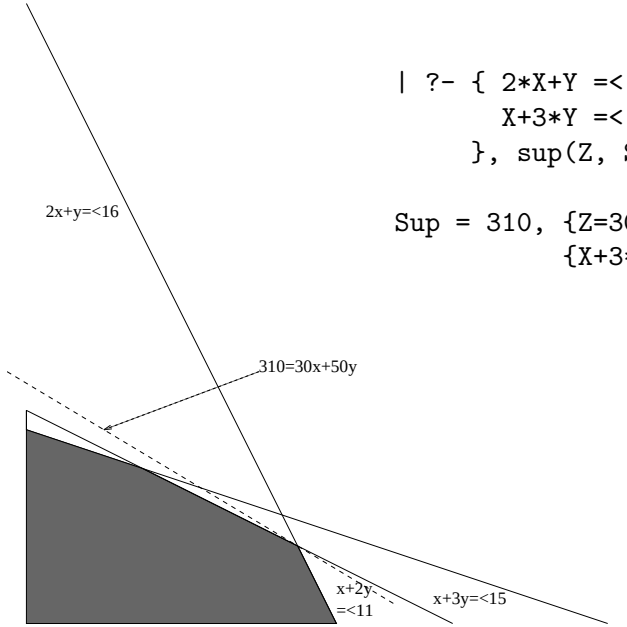
- `bb_inf(Egészek, Kif, Inf)` — kiszámolja `Kif` infimumát, azzal a további feltétellel, hogy az Egészek listában levő minden változó egész (ún. „Mixed Integer Optimisation Problem”).

```
| ?- {X >= 0.5, Y >= 0.5}, inf(X+Y, I).
I = 1, {Y>=1/2}, {X>=1/2} ?
| ?- {X >= 0.5, Y >= 0.5}, bb_inf([X,Y], X+Y, I).
I = 2, {X>=1/2}, {Y>=1/2} ?
```

- `ordering(V1 < V2)` — A `V1` változó előbb szerepeljen az eredmény-korlátban mint a `V2` változó.
- `ordering([V1,V2,...])` — `V1, V2, ...` ebben a sorrendben szerepeljen az eredmény-korlátban.

További eljárások (lásd kézikönyv):
`bb_inf/5`, `dump/3`, `projecting_assert/1`,

Szélsőérték-számítás grafikus illusztrálása




```
| ?- { 2*X+Y =< 16, X+2*Y =< 11,
       X+3*Y =< 15, Z = 30*X+50*Y
     }, sup(Z, Sup).

Sup = 310, {Z=30*X+50*Y}, {X+1/2*Y=<8},
           {X+3*Y=<15}, {X+2*Y=<11}
```


További részletek

Projekció

```
% Az (X,Y) pont az (1,2) (1,4) (2,4) pontok
% által kifeszített háromszögben van.
hszogben(X, Y) :-
        { X=1*L1+1*L2+2*L3,
          Y=2*L1+4*L2+4*L3,
          L1+L2+L3=1, L1>=0, L2>=0, L3>=0 }.

| ?- hszogben(X, Y).
                      {Y=<4}, {X>=1}, {X-1/2*Y=<0} ?

| ?- hszogben(_, Y).
                      {Y=<4}, {Y>=2} ?

| ?- hszogben(X, _).
                      {X>=1}, {X=<2} ?
```

További részletek

Belső ábrázolás

`clpr` — lebegőpontos szám; `clpq` — `rat(`*Számláló*`,` *Nevező*`)`, ahol *Számláló* és *Nevező* relatív prímek. Például `clpq`-ban:

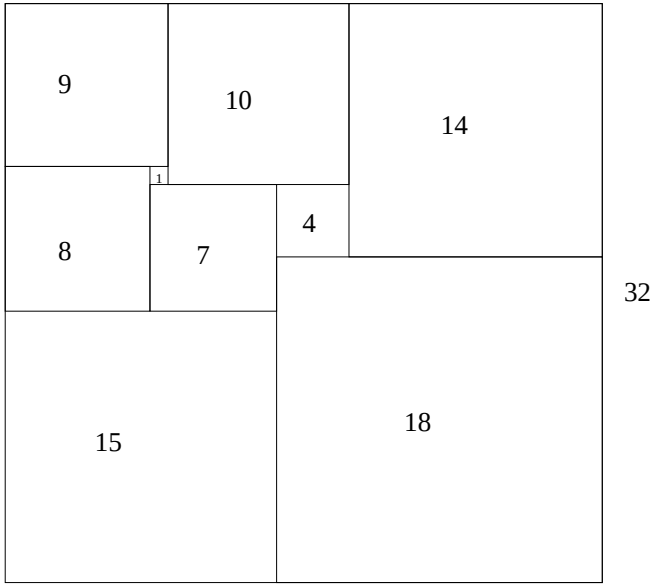
```
| ?- {X=0.5}, X=0.5.
no
| ?- {X=0.5}, X=1/2.
no
| ?- {X=0.5}, X=rat(2,4).
no
| ?- {X=0.5}, X=rat(1,2).
X = 1/2 ?                  % portray jelentíti meg
| ?- {X=5}, X=5.
no
| ?- {X=5}, X=rat(5,1).
X = 5 ?
```


Egy nagyobb CLP(Q) feladat: Tökéletes téglalapok

A feladat

- egy olyan téglalap keresése
- amely kirakható páronként különböző oldalú négyzetekből

Egy megoldás (a legkevesebb, 9 darab négyzet felhasználásával)



Tökéletes téglalapok — CLP(Q) megoldás

```
% Colmerauer A.: An Introduction to Prolog III,
% Communications of the ACM, 33(7), 69-90, 1990.

% Rectangle 1 x Width is covered by distinct
% squares with sizes Ss.
filled_rectangle(Width, Ss) :-
       { Width >= 1 }, distinct_squares(Ss),
       filled_hole([-1,Width,1], _, Ss, []).

% distinct_squares(Ss): All elements of Ss are distinct.
distinct_squares([]).
distinct_squares([S|Ss]) :-
       { S > 0 }, outof(Ss, S), distinct_squares(Ss).

outof([],     _).
outof([S|Ss], S0) :- { S =\= S0 }, outof(Ss, S0).
```

Tökéletes téglalapok — CLP(Q) megoldás

```
% filled_hole(L0, L, Ss0, Ss): Hole in line L0
% filled with squares Ss0-Ss (diff list) gives line L.
% Def: h(L): sum of lengths of vertical segments in L.
% Pre: All elements of L0 except the first >= 0.
% Post: All elems in L >=0, h(L0) = h(L).
filled_hole(L, L, Ss, Ss) :-
       L = [V|_], {V >= 0}.
filled_hole([V|HL], L, [S|Ss0], Ss) :-
       { V < 0 }, placed_square(S, HL, L1),
       filled_hole(L1, L2, Ss0, Ss1), { V1=V+S },
       filled_hole([V1,S|L2], L, Ss1, Ss).

% placed_square(S, HL, L): placing a square size S on
% horizontal line HL gives (vertical) line L.
% Pre: all elems in HL >=0
% Post: all in L except first >=0, h(L) = h(HL)-S.
placed_square(S, [H,V,H1|L], L1) :-
       { S > H, V=0, H2=H+H1 }, placed_square(S, [H2|L], L1).
placed_square(S, [S,V|L], [X|L]) :- { X=V-S }.
placed_square(S, [H|L], [X,Y|L]) :-
       { S < H, X= -S, Y=H-S }.
```


Tökéletes téglalapok: példafutás

```
% pentium i5, bogomips: 5187.85
| ?-    length(Ss, N), N > 1, statistics(runtime, _),
        filled_rectangle(Width, Ss),
        statistics(runtime, [_,MSec]).

N = 9, MSec = 840, Width = 33/32,
Ss = [15/32,9/16,1/4,7/32,1/8,7/16,1/32,5/16,9/32] ? ;

N = 9, MSec = 110, Width = 69/61,
Ss = [33/61,36/61,28/61,5/61,2/61,9/61,25/61,7/61,16/61] ? ;

N = 9, MSec = 1130, Width = 33/32,
Ss = [9/16,15/32,7/32,1/4,7/16,1/8,5/16,1/32,9/32] ?
```

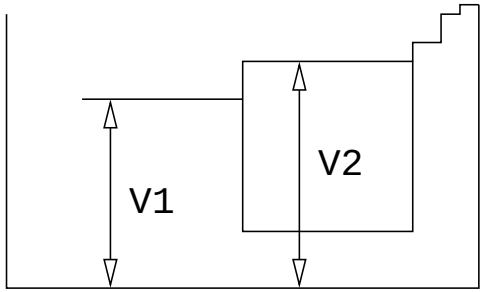

Az `outof` hívás kihagyásával végzett futtatás

Kommentként közöljük a generált korlátokat, a redundánsak elhagyásával.

```
| ?- filled_rectangle(W, [S1,S2,S3], [eqsq]).

S1 = 1/2, S2 = 1, S3 = 1/2, W = 3/2 ? ;    % 3 3 2 2 2 2
                                           % 3 3 2 2 2 2
% {W=S1+S2}, {S2=<1}, {S1=S3},             % 1 1 2 2 2 2
% {S2>=S1+S3}, {S1+S3>=1}.                 % 1 1 2 2 2 2

S1 = 1, S2 = 1/2, S3 = 1/2, W = 3/2 ? ;    % 1 1 1 1 3 3
                                           % 1 1 1 1 3 3
% {W=S1+S2}, {S2=S3}, {S2+S3=<1},          % 1 1 1 1 2 2
% {S2+S3>=S1}, {S1>=1}.                    % 1 1 1 1 2 2

S1 = 1, S2 = 1, S3 = 1, W = 3 ? ; no
% {W=S1+S2+S3}, {S3=<1}, {S3>=S2},         % 1 1 2 2 3 3
% {S2>=S1}, {S1>=1}.                       % 1 1 2 2 3 3

| ?- test_rectangle(3, [eqsq], _Cl), portray_clause(_Cl), fail.
filled_rectangle1(Width, [S1,S2,S3]) :-
  {S1>0}, {S2>0}, {S3>0}, {Width>=1}, {S1<Width}, {S1>0}, {Width=S1+S2},
  {S2=<1}, {S2>=S1}, {S1<1}, {S1=S3}, {S2>=S1+S3}, {S1+S3>=1}.
...
```


Tökéletes téglalapok: választási pontok

Függőleges

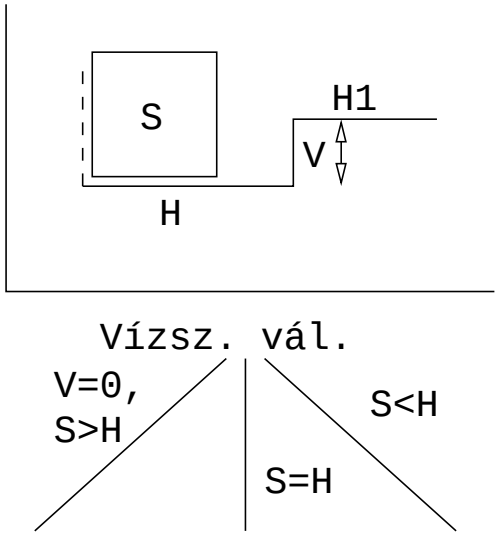


Függ.vál.

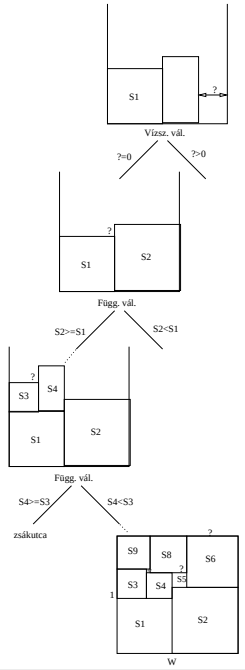
V1=<V2 V1>V2

Tökéletes téglalapok: választási pontok

Vízszintes



Tökéletes téglalapok: a keresési tér szerkezete



IV. rész

A CLP elméleti háttere

A CLP($\mathcal{X}$) séma

Egy adott CLP($\mathcal{X}$) meghatározásakor meg kell adni

- a korlát-következtetés tartományát,
- a korlátok szintaxisát és jelentését (függvények, relációk),
- a korlát-megoldó algoritmust.

A korlátok osztályozása

- *egyszerű korlátok* — a korlát-megoldó azonnal tudja kezelni őket;
- *összetett korlátok* — felfüggesztve, démonként várnak arra, hogy a korlát-megoldónak segíthessenek.

A CLP($\mathcal{X}$) korlát-megoldók közös vonása: a *korlát tár*

- A korlát tár *konzisztens* korlátok halmaza (konjunkciója).
- A korlát tár elemei egyszerű korlátok.
- A közönséges Prolog végrehajtás során a célsorozat mellett a CLP($\mathcal{X}$) rendszer nyilvántartja a korlát tár állapotát:
  - amikor a végrehajtás egy egyszerű korláthoz ér, akkor azt a megoldó megpróbálja hozzávenni a tárhoz;
  - ha az új korlát hozzávételével a tár konzisztens marad, akkor ez a redukciós lépés sikeres és a tár kibővül az új korláttal;
  - ha az új korlát hozzávételével a tár inkonzisztenssé válna, akkor (nem kerül be a tárba és) meghiúsulást, azaz visszalépést okoz;
  - visszalépés esetén a korlát tár is visszaáll a korábbi állapotába.
- Az összetett korlátok démonként (ágensként) várakoznak arra, hogy:
  a. egyszerű korláttá váljanak
  b. a tárat egy egyszerű következményükkel bővíthessék (az ún. erősítés)

A korlát logikai programozás elmélete

Egy CLP rendszer

- $\langle \mathcal{D}, \mathcal{F}, \mathcal{R}, \mathcal{S} \rangle$
- $\mathcal{D}$: egy tartomány (domain), pl. egészek (N), valósak (R), racionálisak(Q), Boole értékek (B), listák, füzérek (stringek) (+ a Prolog-fastruktúrák (Herbrand — H) tartománya)
- $\mathcal{F}$: $\mathcal{D}$-ben definiált függvényjelek egy halmaza, pl. $+$, $-$, $*$, $\vee$, $\wedge$
- $\mathcal{R}$: $\mathcal{D}$-ben definiált relációjelek (korlátok) egy halmaza pl. $=$, $\neq$, $<$, $\in$
- $\mathcal{S}$: egy korlát-megoldó algoritmus $\langle \mathcal{D}, \mathcal{F}, \mathcal{R} \rangle$-re, azaz a $\mathcal{D}$ tartományban az $\mathcal{F} \cup \mathcal{R}$ halmazbeli jelekből felépített korlátokra

CLP szintaxis és deklaratív szemantika

program

- klózok halmaza.

klóz

- szintaxis: `P :- G`$_1$`, ..., G`$_n$, ahol mindegyik `G`$_i$ vagy eljáráshívás, vagy korlát.
- deklaratív olvasat: `P` igaz, ha `G`$_1$`, ..., G`$_n$ mind igaz.

kérdés

- szintaxis: `?- G`$_1$`, ..., G`$_n$
- válasz egy `Q` kérdésre: korlátoknak egy olyan konjunkciója, amelyből a kérdés következik.

CLP procedurális szemantika

Végrehajtási állapot

- $\langle$`G`, $s\rangle$
- `G` — cél/korlát sorozat
- s — korlát-tár: az eddig felhalmozott egyszerű korlátok konjunkciója (kezdetben üres)

Szükséges megkülönböztetés

- egyszerű korlát (`c`): amit a korlát-tár közvetlenül befogad ($\mathcal{F} \cup \mathcal{R}$-től függ)
- összetett korlát (`C`): a tár nem tudja befogadni, de hathat a tárra

Klózok procedurális olvasata

- `P :- G`$_1$`, ..., G`$_n$ jelentése: `P` megoldásához megoldandó `G`$_1$`, ..., G`$_n$.

CLP procedurális szemantika

Végrehajtási invariánsok

- s konzisztens
- $\mathtt{G} \wedge s \rightarrow \mathtt{Q}$ ($\mathtt{Q}$ a kezdő kérdés)

Végrehajtás vége

- $\langle \mathtt{G}_e, s_e \rangle$, ahol $\mathtt{G}_e$-re nem alkalmazható egyetlen következtetési lépés sem.

A végrehajtás eredménye

- Az s_e korlát-tár, vagy annak a kérdésben szereplő változókra való „vetítése” (a többi változó egzisztenciális kvantálásával).
- A $\mathtt{G}_e$ fennmaradó (összetett) korlátok.

A CLP következtetés folyamata

Következtetési lépések

- rezolúció:
  $\langle \mathtt{P} \,\&\, \mathtt{G}, s \rangle \Rightarrow \langle \mathtt{G}_1 \,\&\, \ldots \,\&\, \mathtt{G}_n \,\&\, \mathtt{G}, (\mathtt{P} = \mathtt{P}') \wedge s \rangle$,
  feltéve, hogy a programban van egy $\mathtt{P}' \ \mathtt{:-}\ \mathtt{G}_1, \ldots, \mathtt{G}_n$ klóz.

  Itt $(\mathtt{P} = \mathtt{P}')$ a klózfej és a hívás egyesítését, illetve az ehhez szükséges behelyettesítések elvégzését jelenti.
- korlát-megoldás:
  $\langle \mathtt{c} \,\&\, \mathtt{G}, s \rangle \Rightarrow \langle \mathtt{G}, s \wedge \mathtt{c} \rangle$
- korlát-erősítés:
  $\langle \mathtt{C} \,\&\, \mathtt{G}, s \rangle \Rightarrow \langle \mathtt{C}' \,\&\, \mathtt{G}, s \wedge \mathtt{c} \rangle$
  ha s-ből következik, hogy $\mathtt{C}$ ekvivalens $(\mathtt{C}' \wedge \mathtt{c})$-vel. ($\mathtt{C}' = \mathtt{C}$ is lehet.)

Ha a tár inkonzisztensé válna, visszalépés történik.

A CLP következtetés folyamata

Példa erősítésre

- $\langle \mathtt{X} > \mathtt{Y*Y} \,\&\, \ldots, \mathtt{Y} > 3 \rangle \Rightarrow \langle \mathtt{X} > \mathtt{Y*Y} \,\&\, \ldots, \mathtt{Y} > 3 \wedge \mathtt{X} > 9 \rangle$
  hiszen $\mathtt{X} > \mathtt{Y*Y} \wedge \mathtt{Y} > 3 \Rightarrow \mathtt{X} > 9$
- clp(R)-ben nincs ilyen, de clp(FD)-ben van!

Követelmények a korlát megoldó algoritmussal szemben

- teljesség (egyszerű korlátok konjunkciójáról mindig döntse el, hogy konzisztens-e),
- inkrementalitás (az s tár konzisztenciáját ne bizonyítsa újra),
- a visszalépés támogatása,
- hatékonyság.

V. rész

A SICStus clp(FD) könyvtára

A SICStus `clpfd` könyvtár

Tartomány

Egészek (negatívak is) véges (esetleg végtelen) halmaza

Korlátok

- aritmetikai
- halmaz (halmazba tartozás)
- tükrözött
- logikai
- kombinatorikai
- felhasználó által definiált

Egyszerű korlátok

csak a halmaz-korlátok: `X ∈ Halmaz`

A SICStus `clpfd` könyvtár

Korlát-megoldó algoritmus

- egyszerű korlátok kezelése triviális;
- a lényeg az összetett korlátok **erősítő** tevékenysége, ez a Mesterséges Intelligencia CSP (Constraint Satisfaction Problems) ágának módszerein alapul.

Miről lesz szó?

- CSP, mint háttér
- Alapvető (aritmetikai és halmaz-) korlátok
- Tükrözött és logikai korlátok
- Címkéző eljárások
- Kombinatorikai korlátok
- Felhasználó által definiált korlátok: indexikálisok és globális korlátok
- Az FDBG nyomkövető csomag
- Esettanulmányok: négyzetdarabolás, torpedó-, ill. dominó-feladvány

Példa: SEND MORE MONEY – Prologban és CLPFD-ben

Prolog: generál és ellenőriz

```
:- use_module(library(between)).
send0(SEND, MORE, MONEY) :-
    Ds = [S,E,N,D,M,O,R,Y],
    maplist(between(0, 9), Ds),
    alldiff(Ds),
    S =\= 0, M =\= 0,
    SEND is 1000*S+100*E+10*N+D,
    MORE is 1000*M+100*O+10*R+E,
    MONEY is
      10000*M+1000*O+100*N+10*E+Y,
    SEND+MORE =:= MONEY.

% alldiff(+L):
% L elemei páronként különbözőek.
alldiff([]).
alldiff([D|Ds]) :-
    nonmember(D, Ds), alldiff(Ds).
```

Futási idő: 13.1 sec

CLPFD: (ellenőriz) korlátoz és generál

```
:- use_module(library(clpfd)).
send_clpfd(SEND, MORE, MONEY) :-
    Ds = [S,E,N,D,M,O,R,Y],
    domain(Ds, 0, 9),                (1)
    all_different(Ds),               (2)
    S #\= 0, M #\= 0,                (3)
    SEND #= 1000*S+100*E+10*N+D,     (4)
    MORE #= 1000*M+100*O+10*R+E,     (5)
    MONEY #=
      10000*M+1000*O+100*N+10*E+Y,   (6)
    SEND+MORE #= MONEY,              (7)
    labeling([], Ds).                (8)
```

Új nyelvi elemek:

- tartományt rendelhetünk a változókhoz
- a korlátok szűkítik a vált.-k tartományát.

Futási idő: 0.00011 sec

A program futtatása az FDBG nyomkövetővel

```
| ?- send(SEND, MORE, MONEY).
domain([S,E,N,D,M,O,R,Y],0,9)
all_different([S,E,N,D,M,O,R,Y])
S in(inf.. -1)\/(1..sup)
M in(inf.. -1)\/(1..sup)
SEND#=1000*S+100*E+10*N+D
MORE#=E+1000*M+100*O+10*R
MONEY#=10*E+100*N+10000*M+1000*O+Y
SEND+MORE#=MONEY
MONEY#=10*E+100*N+10000*M+1000*O+Y
all_different([S,E,N,D,M,O,R,Y])

MONEY#=10*E+100*N+10000*M+1000*O+Y
MORE#=E+1000*M+100*O+10*R
SEND#=1000*S+100*E+10*N+D
SEND+MORE#=MONEY

MONEY#=10*E+100*N+10000*M+1000*O+Y
all_different([S,E,N,D,M,O,R,Y])

SEND#=1000*S+100*E+10*N+D
MONEY#=10*E+100*N+10000*M+1000*O+Y
MORE#=E+1000*M+100*O+10*R
SEND#=1000*S+100*E+10*N+D
SEND+MORE#=MONEY
SEND#=1000*S+100*E+10*N+D
```

```
(1) % for each var S, E, N, ...: inf..sup -> 0..9
(2) % no pruning possible
(3) S = 0..9 -> 1..9              Constraint exited.
    M = 0..9 -> 1..9              Constraint exited.
(4) SEND = inf..sup -> 1000..9999
(5) MORE = inf..sup -> 1000..9999
(6) MONEY = inf..sup -> 10000..99999
(7) MONEY = 10000..99999 -> 10000..19998
M = 1..9 -> {1}
S = 1..9 -> 2..9
E = 0..9 -> {0}\/(2..9) etc.
% no pruning
MORE = 1000..9999 -> 1000..1999
SEND = 1000..9999 -> 2000..9999
SEND = 2000..9999 -> 8001..9999
MONEY = 10000..19998 -> 10000..11998
O = {0}\/(2..9) -> {0}
E = {0}\/(2..9) -> 2..9
N = {0}\/(2..9) -> 2..9 etc.
S = 2..9 -> 8..9
MONEY = 10000..11998 -> 10222..10999
MORE = 1000..1999 -> 1022..1099
SEND = 8001..9999 -> 8222..9999
SEND = 8222..9999 -> 9123..9977
S = 8..9 -> {9}
```

A program futtatása az FDBG nyomkövetővel (folyt.)

```
all_different([S,E,N,D,M,O,R,Y])

SEND#=1000*S+100*E+10*N+D
MONEY#=10*E+100*N+10000*M+1000*O+Y
MORE#=E+1000*M+100*O+10*R
SEND+MORE#=MONEY

Labeling [43, E]: in range 2..8.
Labeling [43, E]: indomain_up: E = 2
E in 2..2
all_different([9,E,N,D,1,0,R,Y])

MONEY#=10*E+100*N+10000*1+1000*0+Y
MORE#=E+1000*1+100*0+10*R
SEND#=1000*9+100*E+10*N+D
SEND+MORE#=MONEY


MONEY#=10*E+100*N+10000*1+1000*0+Y
all_different([9,E,N,D,1,0,R,Y])


MORE#=E+1000*1+100*0+10*R
SEND#=1000*9+100*E+10*N+D
```

```
E = 2..9 -> 2..8
N = 2..9 -> 2..8 etc.
SEND = 9123..9977 -> 9222..9888
MONEY = 10222..10999 -> 10222..10888
MORE = 1022..1099 -> 1022..1088
SEND = 9222..9888 -> 9222..9866
MONEY = 10222..10888 -> 10244..10888
(8)

E = 2..8 -> {2}                    Constraint exited.
N = 2..8 -> 3..8
D = 2..8 -> 3..8 etc.
MONEY = 10244..10888 -> 10323..10828
MORE = 1022..1088 -> 1032..1082
SEND = 9222..9866 -> 9233..9288
SEND = 9233..9288 -> 9241..9288
MORE = 1032..1082 -> 1035..1082
MONEY = 10323..10828 -> 10323..10370
N = 3..8 -> {3}
D = 3..8 -> 4..8
R = 3..8 -> 4..8
Y = 3..8 -> 4..8
MORE = 1035..1082 -> 1042..1082
E = {2}, N = {3}, D = 4..8,
SEND = 9241..9288                  Constraint failed.
```

(Labeling for E = 3, 4 also fails, not shown here)

A program futtatása az FDBG nyomkövetővel (folyt. 2)

```
Labeling [43, E]: indomain_up: E = 5
E in 5..5
all_different([9,E,N,D,1,0,R,Y])

MONEY#=10*E+100*N+10000*1+1000*0+Y
MORE#=E+1000*1+100*0+10*R
SEND#=1000*9+100*E+10*N+D
SEND+MORE#=MONEY
MONEY#=10*E+100*N+10000*1+1000*0+Y
all_different([9,E,N,D,1,0,R,Y])

MONEY#=10*E+100*N+10000*1+1000*0+Y
SEND#=1000*9+100*E+10*N+D
SEND+MORE#=MONEY


MONEY#=10*E+100*N+10000*1+1000*0+Y
MORE#=E+1000*1+100*0+10*R
all_different([9,E,N,D,1,0,R,Y])
SEND#=1000*9+100*E+10*N+D

SEND+MORE#=MONEY

MONEY#=10*E+100*N+10000*1+1000*0+Y
```

```
E = 2..8 -> {5}                    Constraint exited.
N = 2..8 -> (2..4)\/(6..8)
D = 2..8 -> (2..4)\/(6..8), etc.
MONEY = 10244..10888 -> 10252..10858
MORE = 1022..1088 -> 1025..1085
SEND = 9222..9866 -> 9522..9588
MONEY = 10252..10858 -> 10547..10673
N = (2..4)\/(6..8) -> {6}
D = (2..4)\/(6..8) -> (2..4)\/(7..8)
R = (2..4)\/(6..8) -> (2..4)\/(7..8), etc.
MONEY = 10547..10673 -> 10652..10658
SEND = 9522..9588 -> 9562..9568
SEND = 9562..9568 -> 9567..9568
MORE = 1025..1085 -> 1084..1085
MONEY = 10652..10658 -> 10652..10653
Y = (2..4)\/(7..8) -> 2..3
R = (2..4)\/(7..8) -> {8}
D = (2..4)\/(7..8) -> (2..4)\/{7}
D = (2..4)\/{7} -> {7}
SEND = 9567..9568 -> {9567}        Constraint exited.
MORE = 1084..1085 -> {1085}
MONEY = 10652..10653 -> {10652}    Constraint exited.
Y = 2..3 -> {2}                    Constraint exited.
```

SEND = 9567, MORE = 1085, MONEY = 10652 ?

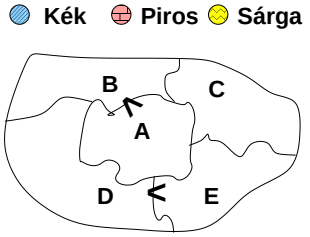
(Labeling for E = 6, 7 and 8 fails, no more answers produced)

Tartalom

Háttér: CSP (Constraint Satisfaction Problems)

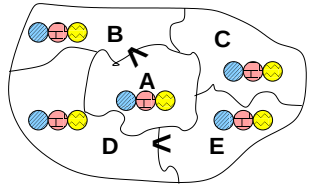
Példafeladat

Az alábbi térkép kiszínezése kék, piros és sárga színekkel úgy, hogy a szomszédos országok különböző színűek legyenek, és ha két ország határán a < jel van, akkor a két szín ábécé-rendben a megadott módon kövesse egymást.



Egy lehetséges megoldási folyamat (*zárójelben a CSP elnevezések*)

1. Minden mezőben elhelyezzük a három lehetséges színt (*változók és tartományaik felvétele*).

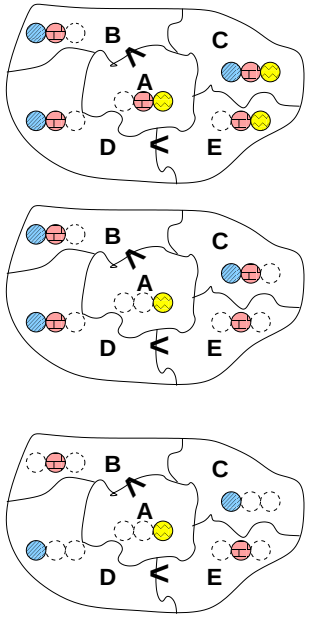


Háttér: CSP (Constraint Satisfaction Problems)

2. Az „A” mező nem lehet kék, mert annál „B” nem lehetne kisebb. A „B” nem lehet sárga, mert annál „A” nem lehetne nagyobb. Az „E” és „D” mezők hasonlóan szűkíthetők (*szűkítés, él-konzisztencia biztosítása*).

3. Ha az „A” mező piros lenne, akkor mind „B”, mind „D” kék lenne, ami ellentmondás (*globális korlát, ill. borotválási technika*). Tehát „A” = sárga. Emiatt a vele szomszédos „C” és „E” nem lehet sárga (*él-konzisztens szűkítés*), így „E” = piros.

4. „C” és „D”, mivel a piros „E”-vel határosak, nem lehetnek pirosak, tehát mindkettő = kék. így „B”-nek van biztosan sárga („A”) és biztosan kék szomszédja („D”), emiatt „B” csak piros lehet (*él-konszitens szűkítés*). Tehát az egyetlen megoldás: A = sárga, B = piros, C = kék, D = kék, E = piros.



A CSP fogalma

- CSP = (Xs, Ds, Cs)
  - $Xs = \langle x_1, \ldots, x_n \rangle$ — változók
  - $Ds = \langle D_1, \ldots, D_n \rangle$ — tartományok, azaz nem üres halmazok
  - az x_i változó a D_i véges halmazból (x_i tartománya) vehet fel értéket
  - Cs a problémában szereplő korlátok (atomi relációk) halmaza, argumentumaik Xs változói (például $Cs \ni c = r(x_1, x_3)$, $r \subseteq D_1 \times D_3$)
- A térképszínezési probléma a fenti jelölésrendszerben:
  - Változók: $Xs = \langle A, B, C, D, E \rangle$
  - Tartományok (kezdőállapota): $Ds = \langle \{k, p, s\}, \{k, p, s\}, \{k, p, s\}, \{k, p, s\}, \{k, p, s\} \rangle$
  - Korlátok (constraints): $B{<}A$, $E{<}D$, $A{\neq}C$, $A{\neq}D$, $A{\neq}E$, $B{\neq}C$, $C{\neq}E$, $D{\neq}B$, ahol a „$<$” reláció definíciója: $\{\langle k, p \rangle, \langle p, s \rangle, \langle k, s \rangle\}$, a „$\neq$” relációé pedig: $\{\langle k, p \rangle, \langle p, k \rangle, \langle p, s \rangle, \langle s, p \rangle, \langle s, k \rangle, \langle k, s \rangle\}$
  - A CSP feladat megoldása: minden x_i változóhoz egy $v_i \in D_i$ értéket kell rendelni úgy, hogy minden $c \in Cs$ korlátot egyidejűleg kielégítsünk.

A CSP fogalma, folyt.

- **Definíció:** egy c korlát egy x_i változójának d_i értéke *felesleges*, ha nincs a c többi változójának olyan értékrendszere, amely d_i-vel együtt kielégíti c-t. Pl. tekintsük a $B{<}A$ korlátot, ahol az A és B változók tartománya egyaránt $\{k, p, s\}$. Itt B-ben *felesleges* az s érték, mert A-ban nincs s-nél nagyobb érték, és hasonlóan *felesleges* az A változó k értéke.
- Felesleges érték(ek) elhagyását szűkítésnek nevezzük (a változók tartományai szűkülnek)
- (Triviális) **Állítás:** Szűkítéssel, azaz *felesleges értékek elhagyásával* ekvivalens CSP-t kapunk, másszóval a szűkített CSP megoldáshalmaza azonos az eredeti CSP megoldáshalmazával.
- **Definíció:** egy korlát *él-konzisztens* (arc consistent), ha egyik változójának tartományában sincs felesleges érték. A CSP *él-konzisztens*, ha minden korlátja él-konzisztens. Az él-konzisztencia szűkítéssel biztosítható.
- Ha minden reláció bináris, a CSP probléma gráffal ábrázolható (változó $\Rightarrow$ csomópont, reláció $\Rightarrow$ él). Az *él*-konzisztencia elnevezés ebből fakad.

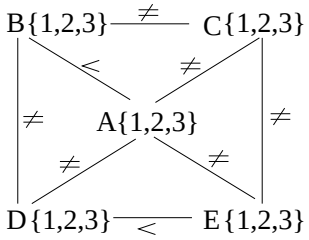
A térképszínezés mint CSP feladat

Modellezés (a feladat leképezése CSP-re)

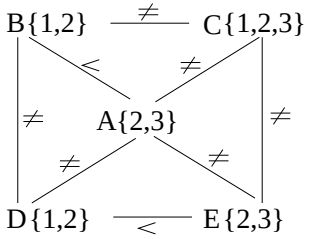
- változók meghatározása: országonként egy változó, amely az ország színét jelenti;
- változóértékek kódolása: kék $\rightarrow$ `1`, piros $\rightarrow$ `2`, sárga $\rightarrow$ `3` (sok CSP megvalósítás kiköti, hogy a tartományok elemei nem-negatív egészek legyenek – így pl. a Prolog rendszerek `clpfd` könyvtárai is);
- korlátok meghatározása:
  - az előírt < relációk teljesülnek,
  - a többi szomszédos ország-pár színkódja különböző.

A térképszínezés mint CSP feladat

A kiinduló korlát-gráf:



A korlát-gráf él-konzisztens szűkítése:



A CSP megoldás folyamata

- Felvesszük a változók tartományait;
- Felvesszük a korlátokat mint démonokat, amelyek szűkítéssel él-konzisztenciát biztosítanak;
- Többértelműség esetén (azaz, ha van olyan változó, amelynek tartománya nem egyelemű) címkézést (labeling) végzünk:
  - kiválasztunk egy változót (pl. a legkisebb tartományút),
  - a tartományt két vagy több részre osztjuk (választási pont),
  - az egyes választásokat visszalépéses kereséssel bejárjuk (egy tartomány üresre szűkülése váltja ki a visszalépést).

CLP(FD) = a CSP beágyazása a CLP($\mathcal{X}$) sémába

A CSP $\rightarrow$ CLP(FD) megfeleltetés

- CSP változó $\rightarrow$ CLP változó
- CSP: x tartománya T $\rightarrow$ CLP: „`X in T`” egyszerű korlát.
- CSP korlát $\rightarrow$ CLP korlát, *általában összetett!*

A CLP(FD) korlát-tár

- Tartalma: X `in` *Tartomány* alakú egyszerű korlátok.
- Tekinthető úgy mint egy hozzárendelés a változók és tartományaik (lehetséges értékek) között.
- Egyszerű korlát hozzávétele a tárhoz: egy már bennlévő változó tartományának szűkítése vagy egy új változó-hozzárendelés felvétele.

CLP(FD) = a CSP beágyazása a CLP($\mathcal{X}$) sémába

Összetett CLP(FD) korlátok

- A korlátok többsége démon lesz, hatását a *korlát-erősítés*en keresztül fejti ki ($\langle C, s\rangle \longrightarrow \langle C', s \wedge c\rangle$ ahol $s \models C \equiv C' \wedge c$).
- Az erősítés egy egyszerű korlát hozzávételét jelenti, azaz a CLP(FD) esetén egy változó tartományának szűkítését.
- A démonok ciklikusan működnek: szűkítenek, elalszanak, aktiválódnak, szűkítenek, ...
- A démonokat a korlátbeli változók tartományának változása aktiválja.
- Különböző korlátok különböző mértékű szűkítést alkalmazhatnak (a maximális szűkítés túl drága lehet).

Tartalom

A `clpfd` könyvtár — alapvető-korlátok

Alapvető aritmetikai korlátok (ún. formula-korlátok)

- Függvények:
  `+ - * / mod div rem` (kétargumentumú operátorok)
  `min max` (kétargumentumú függvények, zárójeles jelöléssel)
  `abs` (egyargumentumú függvény, zárójeles jelöléssel)
- Korlát-relációk:
  `#<, #>, #=<, #>=, #= #\=` (mind `xfx 700` operátorok)

Halmazkorlátok

- *X* `in` *KTartomány*, jelentése: $X \in H$, ahol H a *KTartomány* (konstans tartomány) által leírt halmaz (Az `in` atom egy `xfx 700` operátor);
- `domain(`$[X, Y, \ldots]$`,`*Min*`,`*Max*`)`: $X \in [Min, Max]$, $Y \in [Min, Max]$, ...

Itt *Min* lehet *Szám* vagy `inf` ($-\infty$), *Max* pedig *Szám* vagy `sup` ($+\infty$);
(Megjegyzés: a végtelen tartományok főleg kényelmi célokat szolgálnak: nem kell kiszámolnunk az alsó/felső korlátokat, ha azok kikövetkeztethetők.)

A `clpfd` könyvtár — alapvető-korlátok

Egy *KTartomány* a következők egyike lehet:

- felsorolás: `{`*Szám*`, ...}`,
- intervallum: *Min*`..`*Max*, (`xfx 550` operátor),
- metszet: *KTartomány* `/\` *KTartomány* (`yfx 500` operátor),
- únió: *KTartomány* `\/` *KTartomány*, (`yfx 500` operátor),
- komplemens: `\` *KTartomány*, (`fy 500` operátor).

Példák

```
| ?- X in (10..20)/\ (\{15}), Y in 6..sup, Z #= X+Y.

   X in(10..14)\/(16..20), Y in 6..sup, Z in 16..sup ?

| ?- X in 10..20, X #\= 15, Y in {2}, Z #= X*Y.

   Y = 2, X in(10..14)\/(16..20), Z in 20..40 ?
```


A térképszínezési feladat SICStus-ban

```
| ?- use_module(library(clpfd)).
...
| ?- domain([A,B,C,D,E], 1, 3),
     A #> B, A #\= C, A #\= D, A #\= E,
     B #\= C, B #\= D, C #\= E, D #< E.
            A in 2..3, B in 1..2,
            C in 1..3, D in 1..2, E in 2..3 ? ;
            no

| ?- domain([A,B,C,D,E], 1, 3),
     A #> B, A #\= C, A #\= D, A #\= E,
     B #\= C, B #\= D, C #\= E, D #< E,
     member(A, [1,2,3]). % címkézés, hivatalosan:
%    indomain(A).        % vagy:
%    labeling([], [A]).  % általánosan:
%    labeling([], [A,B,C,D,E]).
            A = 3, B = 2, C = 1, D = 1, E = 2 ? ;
            no

| ?- domain([A,B,C,D,E], 1, 3),
     A #> B, A #\= D, B #\= C, B #\= D, D #< E,
%    A #\= C, A #\= E, C #\= E helyett:
     all_distinct([A,C,E]).
     % Az "A, C, E különbözőek" korlát okos
     % megvalósítása, globális kombinatorikai korláttal
            A = 3, B = 2, C = 1, D = 1, E = 2 ? ; no
```

Címkéző könyvtári eljárások — rövid előzetes

- `indomain(`*X*`)`: X-et a tartománya által megengedett értékkel helyettesíti, visszalépéskor felsorolja az összes értéket (növekedő sorrendben)
- `labeling(`*Opciók*`, `*Változók*`)`: A *Változók* lista minden elemét behelyettesíti, az *Opciók* lista által előírt módon (az *Opciók* lista lehet üres is)

Szűkítési szintek

Informálisan, `r(X,Y)` bináris relációra

- Tartomány-szűkítés: `X` tartományából minden olyan `x` értéket elhagyunk, amelyhez nem található `Y` tartományában olyan `y` érték, amelyre `r(x,y)` fennáll. Hasonlóan szűkítjük `Y` tartományát. (Ez él-konzisztenciát eredményez.)
- Intervallum-szűkítési lépés: `X` tartományából elhagyjuk annak **alsó vagy felső** határát, ha ahhoz nem található `Y` **tartományának szélső értékei közé eső** olyan `y` érték, amelyre `r(x,y)` fennáll, és fordítva. Ezeket a lépéseket ismételjük, ameddig szűkítenek.

Szűkítési szintek – példa

- Legyen
  - `r(X,Y): X` = *abs*(`Y`).
  - `X` tartománya `0..5`
  - `Y` tartománya `{-1,1,3,4}`
- A tartomány-szűkítés elhagyja `X` tartományából a `0,2,5` értékeket, eredménye `X` ∈ `{1,3,4}`.
- Az intervallum-szűkítés `X` tartományából csak az `5` értéket hagyja el, eredménye `X` ∈ `0..4`.
- Az intervallum-szűkítés kétféle módon is gyengébb mint a tartomány-szűkítés:
  - csak a tartomány szélső értékeit hajlandó elhagyni, ezért nem hagyja el a `2` értéket;
  - a másik változó tartományában nem veszi figyelembe a „lukakat”, így a példában `Y` tartománya helyett annak *lefedő intervallumát*, azaz a `-1..4` intervallumot tekinti — ezért nem hagyja el `X`-ből a `0` értéket.
- Ugyanakkor az intervallum-szűkítés általában konstans idejű művelet, míg a tartomány-szűkítés ideje (és az eredmény mérete) függ a tartományok méretétől.

Szűkítési szintek – definíciók

Jelölések

- Legyen C egy n-változós korlát, s egy tár,
- $D(X, s)$ az X változó tartománya az s tárban,
- $D'(X, s) = min(D(X, s))..max(D(X, s))$, az X változó tartományát *lefedő* (legszűkebb) *intervallum*.

A szűkítési szintek definíciója

- Tartomány-szűkítés (domain consistency)
  C **tartomány-szűkítő**, ha minden szűkítési lépés lefutása után az adott C korlát él-konzisztens, azaz bármelyik X_i változójához és annak tetszőleges $V_i \in D(X_i, s)$ megengedett értékéhez található a többi változónak olyan $V_j \in D(X_j, s)$ értéke ($j = 1, \ldots, i-1, i+1, \ldots, n$), hogy $C(V_1, \ldots V_n)$ fennálljon.
- Intervallum-szűkítés (interval consistency)
  C **intervallum-szűkítő**, ha minden szűkítési lépés lefutása után igaz, hogy C bármelyik X_i változója esetén e változó tartományának mindkét **vég**pontjához (azaz a $V_i = min(D(X_i, s))$ illetve $V_i = max(D(X_i, s))$ értékekhez) található a többi változónak olyan $V_j \in D'(X_j, s)$ értéke ($j = 1, \ldots, i-1, i+1, \ldots, n$), hogy $C(V_1, \ldots V_n)$ fennálljon.

Szűkítési szintek – definíciók

Megjegyzések

- A tartomány-szűkítés lokálisan (egy korlátra nézve) a lehető legjobb;
- **DE** mégha minden korlát tartomány-szűkítő, a megoldás nem garantálható, pl.
  `| ?- domain([X,Y,Z], 1, 2), X#\=Y, X#\=Z, Y#\=Z.`
- Egy CLP(FD) probléma megoldásának hatékonysága fokozható:
  - több korlát összefogását jelentő ún. globális korlátokkal, pl. `all_distinct(L)`: Az `L` lista csupa különböző elemből áll;
  - redundáns korlátok felvételével.

Garantált szűkítési szintek SICStusban

A SICStus által garantált szűkítési szintek

- A halmaz-korlátok (triviálisan) tartomány-szűkítők.
- A *lineáris* aritmetikai korlátok legalább intervallum-szűkítők.
- A nem-lineáris aritmetikai korlátokra nincs garantált szűkítési szint.
- Ha egy változó valamelyik határa végtelen (`inf` vagy `sup`), akkor a változót tartalmazó korlátokra nincs szűkítési garancia (bár az aritmetikai és halmaz-korlátok ilyenkor is szűkítenek).
- A később tárgyalandó korlátokra egyenként megadjuk majd a szűkítési szintet.

Garantált szűkítési szintek SICStusban – példák

```
| ?- X in {4,9}, Y in {2,3}, Z #= X-Y.
        % intervallum-szűkítő:
        X in {4}\/{9}, Y in 2..3, Z in 1..7 ?

| ?- X in {4,9}, Y in {2}, Z #= X-Y.
        % tartomány-szűkítő (SICStus 4.9.0-ban):
        X in {4}\/{9}, Y = 2,       Z in {2}\/{7} ?
        % de SICStus 4.7.1-ben:     Z in 2..7

| ?- X in {4,9}, Y in {2,3}, plus(Y, Z, X).
        % plus(A, B, C): A+B=C tartomány-szűkítő módon
        X in {4}\/{9}, Y in 2..3, Z in(1..2)\/(6..7) ?

| ?- domain([X,Y], -10, 10), X*X+2*X+1 #= Y.
        % Ez nem interv.-szűkítő, mert Y<0 nem lehet!
        X in -4..4, Y in -7..10 ?

| ?- domain([X,Y], -10, 10), (X+1)*(X+1) #= Y.
        % bár ez nem garantált, de intervallum-szűkítő:
        X in -4..2, Y in 0..9 ?
```


Korlátok végrehajtása

A végrehajtás fázisai

- A korlát kifejtése elemi korlátokra (fordítási időben, lásd később) pl. `X*X #< 17` $\Longrightarrow$ `X*X #= Z, Z #< 17`
- A korlát felvétele (posting):
  - azonnali végrehajtás (pl. `X #< 3`), vagy
  - démon létrehozása: első szűkítés elvégzése, újra-aktiválási feltételek meghatározása, a démon elaltatása.
- A démon aktiválása
  - szűkítés elvégzése,
  - döntés a folytatásról:
    - a démon lefut, ha a korlát már biztosan fennáll (következménye a tárnak);
    - vagy a démon újra elalszik.

Korlátok végrehajtása

Elemi korlátok működése — példák

`A #\= B` (tartomány-szűkítő)

- Mikor **aktiválódik**? Ha vagy `A` vagy `B` konkrét értéket kap.
- Hogyan **szűkít**? A felvett értéket kihagyja a másik változó tartományából.
- Hogyan **folytatódik** a démon végrehajtása?
  A démon befejezi működését (lefut).

`A #< B` (tartomány-szűkítő)

- **Aktiválás**: ha `A` alsó határa (min `A`) vagy `B` felső határa (max `B`) változik
- **Szűkítés**: `A` tartományából kihagyja az $X \geq$ max `B` értékeket,
  `B` tartományából kihagyja az $Y \leq$ min `A` értékeket
- **Döntés a folytatásról**: (a legprecízebb:) ha max `A` $<$ min `B`, akkor lefut, különben újra elalszik. (A legtöbb Prolog rendszerben a korlát csak akkor fejezi be működését, ha `A` vagy `B` behelyettesítődik, mert ez gazdaságosabb.)

Korlátok végrehajtása – további példák

`all_distinct([A1,...])` (tartomány-szűkítő)

- **Aktiválás**: ha bármelyik változó tartománya változik
- **Szűkítés**: (páros gráfokban maximális párosítást kereső algoritmus segítségével) minden olyan értéket elhagy, amelyek esetén a korlát nem állhat fenn. Példa:

```
| ?- A in 2..3, B in 2..3, C in 1..3,
     all_distinct([A,B,C]).
                C = 1, A in 2..3, B in 2..3 ?
```

- **Folytatás**: ha már csak egy nem-konstans argumentuma van, akkor lefut, különben újra elalszik. (Jobb döntésnek tűnhet lefutni, ha a tartományok mind diszjunktak, de a SICStus nem így csinálja, valószínűleg nem éri meg.)

Korlátok végrehajtása – további példák

`X+Y #= T` (intervallum-szűkítő)

- **Aktiválás**: ha bármelyik változó alsó vagy felső határa változik
- **Szűkítés**:
  `T`-t szűkíti a `(min(X)+min(Y))..(max(X)+max(Y))` intervallumra,
  `X`-t szűkiti a `(min(T)-max(Y))..(max(T)-min(Y))` intervallumra,
  `Y`-t szűkiti a `(min(T)-max(X))..(max(T)-min(X))` intervallumra.
- **Folytatás**: ha (a szűkítés után) mindhárom változó behelyettesített (konstans), akkor lefut, különben újra elalszik.

Példa a szűkítések kölcsönhatására

```
| ?- domain([X,Y], 0, 100), X+Y #=10, X-Y #=4.
                X in 4..10, Y in 0..6 ?

| ?- domain([X,Y], 0, 100), X+Y #=10, X+2*Y #=14.
                X = 6, Y = 4 ?
```


Globális aritmetikai korlátok

`scalar_product(Coeffs, Xs, Relop, Value[,Options])`
Igaz, ha a `Coeffs` és `Xs` listák skalárszorzata a *`Relop`* relációban van a `Value` értékkel, ahol *`Relop`* aritmetikai összehasonlító operátor (`#=`, `#<`, stb.).
Alapértelmezésben intervallum-szűkítést biztosít, kivéve ha `Options = [consistency(domain)]`, amikor is tartomány-konzisztensen szűkít.
`Coeffs` egészekből álló lista, `Xs` elemei és `Value` egészek vagy korlát változók lehetnek.
Megjegyzés: minden lineáris aritmetikai korlát átalakítható egy `scalar_product` hívássá.

`sum(Xs, Relop, Value)`
Jelentése: Σ `Xs` *`Relop`* `Value`.
Ekvivalens a következővel: `scalar_product(Csupa1, Xs, Relop, Value)`, ahol `Csupa1` csupa `1` számból álló lista, `Xs`-sel azonos hosszú.

`minimum(Value, Xs)`, `maximum(Value, Xs)`
Jelentése: az `Xs` lista elemeinek minimuma/maximuma `Value`.

Példa globális aritmetikai korlátok használatára

```
send(SEND, MORE, MONEY) :-
    length(List, 8),
    domain(List, 0, 9),                % tartományok megadása
    send2(List, SEND, MORE, MONEY),    % korlátok felvétele
    labeling([], List).                % címkézés

send2(List, SEND, MORE, MONEY) :-
    List= [S,E,N,D,M,O,R,Y],
    Pow10 = [1000,100,10,1],
    all_different(List), S #\= 0, M#\= 0,
    scalar_product(Pow10, [S,E,N,D], #=, SEND),
    %   SEND #=  1000*S+100*E+10*N+D,
    scalar_product(Pow10, [M,O,R,E], #=, MORE),
    %   MORE #= 1000*M+100*O+10*R+E,
    scalar_product([10000|Pow10], [M,O,N,E,Y], #=, MONEY),
    %   MONEY #= 10000*M+1000*O+100*N+10*E+Y,
    SEND+MORE #= MONEY.
```


Miért más a CLP(FD), mint a többi CLP rendszer?

A CLP könyvtárak összehasonlítása

	clpq/r	(clpb)	clpfd
Korlátok:	aritmetikai	logikai	aritmetikai, logikai, kombinatorikai,...
Egyszerű korlátok:	lineárisak	összes	*X* in *Halmaz*
Összetett korlátok végrehajtása:	várakozás, míg lineáris nem lesz	nincs ilyen	erősítés (szűkítés)
A tár konzisztenciájának biztosítása:	Gauss eliminá-ció, szimplex módszer	Bináris Döntési Diagrammok	triviális: *X* in *Halmaz*→ *Halmaz* nem üres
Az összes korlát konzisztenciájának biztosítása:	lineáris esetben automatikus	automatikus	csak címkézésen keresztül
Átlátszóság:	fekete doboz	fekete doboz	üveg-doboz
Kiterjeszthetőség:	nem	nem	igen

Miért más a CLP(FD), mint a többi CLP rendszer?

A CLP(FD) fő jellemzői

- A tár konzisztenciájának biztosítása triviális.
- A lényeg a démonok erősítő (szűkítő) működésében van.
- A démonok nem látják egymást, csak a táron keresztül hatnak egymásra.
- Globális korlátok: egyszerre több (akárhány) korlátot helyettesítenek, így erősebb szűkítést adnak (pl. `all_distinct`).
- A megoldás megléte általában csak a címkézéskor derül ki.

A CLP(FD) jellemzői — példák

```
| ?- domain([X,Y,Z], 1, 2), X #\= Y, X #\= Z, Y #\= Z.
                X in 1..2, Y in 1..2, Z in 1..2 ?

| ?- X #> Y, Y #> X.
                Y in inf..sup, X in inf..sup ?

| ?- domain([X,Y], 1, 10), X #> Y, Y #> X.
                no

| ?- statistics(runtime,_),
     (   domain([X,Y], 1, 10000000), X #> Y, Y #> X
     ;   statistics(runtime,[_,T]),
         findall(K-V, fd_statistics(K,V), L)
     ).
T = 7915,
L = [resumptions-10000001,entailments-1,prunings-10000002,
     backtracks-1,constraints-2] ?
```

CLPFD statisztikák lekérdezése

- `fd_statistics(Kulcs, Érték)`: A `Kulcs`-hoz tartozó számlálót `Érték`-kel egyesíti, majd **lenullázza**.
- Lehetséges kulcsok és számlált események:
  - `constraints` — korlát létrehozása;
  - `resumptions` — korlát felébresztése;
  - `entailments` — korlát (vagy negáltja) levezethetővé válásának észlelése;
  - `prunings` — tartomány szűkítése;
  - `backtracks` — a tár ellentmondásossá válása (Prolog meghiúsulások nem számítanak).
- `fd_statistics`: az összes számláló állását kiírja és lenullázza őket.

A szűkítések nyomkövetése az FDBG könyvtár segítségével

```
| ?- use_module(library(clpfd)).
| ?- use_module(library(fdbg)).
| ?- fdbg_on, fdbg_assign_name(X, x), fdbg_assign_name(Y, y),
     domain([X,Y], 1, 10), X #> Y, Y #> X.

domain([<x>,<y>],1,10) ==> x = inf..sup -> 1..10, y = inf..sup -> 1..10
                           Constraint exited.

<x> #> <y>     ==> x = 1..10 -> 2..10,   y = 1..10 -> 1..9
<x> #< <y>     ==> x = 2..10 -> 2..8,    y = 1..9 -> 3..9
<x> #> <y>     ==> x = 2..8 -> 4..8,     y = 3..9 -> 3..7
<x> #< <y>     ==> x = 4..8 -> 4..6,     y = 3..7 -> 5..7
<x> #> <y>     ==> x = 4..6 -> {6},      y = 5..7 -> {5}
                       Constraint exited.
<x> #< <y>     ==> x = {6}, y = {5}
                       Constraint failed.
no
```


Klasszikus CSP/CLP programok: a „zebra” feladat

A feladvány

Egy utcában öt különböző színű ház van egymás mellett. A házakban különböző nemzetiségű és foglalkozású emberek laknak. Mindenki különböző háziállatot tart és más-más a kedvenc italuk is. A következőket tudjuk.

- Az angol a piros házban lakik.
- A festő japán.
- A norvég a balszélső házban lakik.
- A zöld ház a fehérnek jobboldali szomszédja.
- A diplomata a sárga házban lakik.
- A hegedűművész gyümölcslevet iszik.
- Az orvos szomszédja rókát tart.
- A spanyol kutyát tart.
- Az olasz a teát kedveli.
- A zöld házban lakó kávét iszik.
- A szobrász csigát tart.
- A tejet a középső házban kedvelik.
- A norvég a kék ház mellett lakik.
- A diplomata melletti házban lovat tartanak.

Kérdés: Kinek a háziállata a zebra, (és ki iszik vizet)?
(Lásd pl. `https://www.ps.uni-saarland.de/alice/manual/cptutorial/node30.html`)

Klasszikus CSP/CLP programok: a „zebra” feladat

Modellezés

- Változók meghatározása: egy-egy változó tartozik minden nemzetiséghez, háziállathoz, házszínhez, foglalkozáshoz és italhoz.
- Változóértékek kódolása: A változó értéke annak a háznak a száma (balról számozva), amelynek lakóját (ill. annak nemzetiségét), állatát, színét, stb. jelöli az adott változó.
- Korlátok meghatározása:
  - az egyes változó-csoportok (pl. nemzetiség) mind különböznek: `all_different/1` könyvtári korlát, pl. `all_different([Angol,Spanyol,Japán,Norvég,Olasz])`
  - két tulajdonság azonossága: egy `#=` korlát, pl. „Az angol a piros házban lakik.” $\Longrightarrow$ `Angol #= Piros` (angol ház száma = piros ház száma)
  - két tulajdonság szomszédossága: házszámok különbsége 1, ill. 1 abszolút értékű, pl. „A norvég a kék ház mellett lakik” $\Longrightarrow$ `abs(Norvég-Kék)#=1`
  - A sorban egy konkrét ház megnevezése: egy számmal való egyenlőség, pl. „A tejet a középső házban kedvelik.” $\Longrightarrow$ `Tej #= 3`.

A „zebra” feladvány CLPFD megoldása

```
:- use_module(library(lists)).   :- use_module(library(clpfd)).

% ZOwner a zebra tulajdonosának nemzetisége.
zebra(ZOwner):-

  Nations     = [England,Spain,Japan,Norway,Italy],
  Animals     = [Dog, Zebra, Fox, Snail, Horse],
  Colors      = [Green,Red,Yellow,Blue,White],
  Professions = [Painter,Diplomat,Violinist,Doctor,Sculptor],
  Drinks      = [Juice,_Water,Tea,Coffee,Milk],

  Categories  = [Nations,Animals,Colors,Professions,Drinks],
  append(Categories, AllVars), domain(AllVars, 1, 5),
  maplist(all_distinct, Categories),

  England #= Red,     Violinist #= Juice,  Sculptor #= Snail,
  Japan #= Painter,   nextto(Fox, Doctor), Milk #= 3,
  Norway #= 1,        Spain #= Dog,        nextto(Norway, Blue),
  Green #= White+1,   Italy #= Tea,        nextto(Horse, Diplomat),
  Diplomat #= Yellow, Green #= Coffee,

  labeling([], AllVars),
  Answers     = [england,spain,japan,norway,italy],
  nth1(N, Nations, Zebra), nth1(N, Answers, ZOwner).

nextto(A, B) :-  abs(A-B) #= 1.    % A és B szomszédos egész számok.
```

```
| ?- zebra(ZOwner).
ZOwner = japan ? ; no
```


Klasszikus CSP/CLP programok: N vezér a sakktáblán

A feladvány

Egy N*N-es sakktáblán N vezért kell elhelyezni úgy, hogy egyik se üsse semelyik másikat, azaz ne legyen két vezér ugyanabban a sorban, ugyanabban az oszlopban, vagy ugyanazon átlós irányú vonal mentén.

Modellezés

- Változók meghatározása: minden vezérhez egy változót rendelünk. Az X_i változó írja le az i. sorban levő vezér helyzetét.
- Változóértékek kódolása: az X_i változó azt az oszlopot jelöli, amelybe az i. sorban levő vezér kerül.

N vezér a sakktáblán – korlátok meghatározása

- Ne legyen két vezér egy sorban: nem szükséges külön korlát, mert a modellezés (változók jelentése) automatikusan biztosítja.
- Ne legyen két vezér egy oszlopban:
  X_i `#\=` X_j, minden $1 \leq i < j \leq N$ esetén.
- Minden átlós vonalban legfeljebb egy vezér legyen, azaz bármely két vezér vízszintes és függőleges távolsága különbözzék:
  $\mathrm{abs}(X_i - X_j)$ `#\=`$j - i$, minden $1 \leq i < j \leq N$ esetén.
- **Összegezve:** minden `X`, `Y` változópárra, amelyek sortávolsága `I`> 0 (azaz `X` $= X_i$, `Y` $= X_j$, `I` $= abs(i - j)$), a következő három korlát fennállását kell biztosítani:
  `Y #\= X, Y #\= X-I, Y #\= X+I`
- A fenti korlátok eljárásba foglalása:

```
% Az X és Y oszlopokban I sortávolságra levő
% vezérek nem támadják egymást.
no_threat(X, Y, I) :-
        Y #\= X, Y #\= X-I, Y #\= X+I.
```


N vezér a sakktáblán – Prolog (szervező) kód

```
% A Qs lista N vezér biztonságos elhelyezését mutatja egy N*N-es
% sakktáblán: ha a lista i. eleme j, akkor az i. vezért az i. sor
% j. oszlopába kell helyezni.  LabOpts a címkézési opciók listája.
queens(N, Qs, LabOpts) :-
        queens_nolab(N, Qs), labeling(LabOpts,Qs).

% A Qs lista egy biztonságos N vezér elhelyezés.
queens_nolab(N, Qs) :-
        length(Qs, N), domain(Qs, 1, N), safe(Qs).

% safe(Qs): A Qs vezér-lista biztonságos.
safe([]).
safe([Q|Qs]) :- no_attack(Qs, Q, 1), safe(Qs).

% no_attack(Qs, Q, I): A Qs lista által leírt vezérek egyike sem
% támadja a Q által leírt vezért, ahol Qs a (j, j+1, ...) sorbeli
% vezéreket írja le, Q a i. sorbeli vezért, és I = j-i > 0.
no_attack([],_,_).
no_attack([X|Xs], Y, I) :-
        no_threat(X, Y, I), I1 is I+1, no_attack(Xs, Y, I1).
```

N vezér a sakktáblán – Futási példák

```
| ?- queens_nolab(4, Qs).
        Qs = [_A,_B,_C,_D],
        _A in 1..4, _B in 1..4, _C in 1..4, _D in 1..4 ?
| ?- queens_nolab(4, Qs), Qs=[1|_].
        Qs = [1,_A,_B,_C],
        _A in 3..4, _B in{2}\/{4}, _C in 2..3 ?
| ?- Qs = [1|_], queens(4, Qs, []).
        no
| ?- queens_nolab(4, Qs), Qs=[2|_].
        Qs = [2,4,1,3] ?
```


2. kis házi feladat: számkeresztrejtvény (kakuro)

- Adott egy keresztrejtvény, amelynek egyes kockáiba 1..*Max* számokat kell elhelyezni (szokásosan $Max = 9$).
- A vízszintes és függőleges „szavak” meghatározásaként a benne levő számok összege van megadva.
- Egy szóban levő betűk (kockák) mind különböző értékkel kell bírjanak. (Lásd pl. `https://hu.wikipedia.org/wiki/Kakuro`)

A keresztrejtvény Prolog ábrázolása:

- listák listájaként megadott mátrix;
- a fekete kockák helyén $F\backslash V$ alakú struktúrák vannak, ahol F és V az adott kockát követő függőleges ill. vízszintes szó összege, vagy `x`, ha nincs ott szó, *vagy egy egybetűs szó van*;
- a kitöltendő fehér kockákat (különböző) változók jelzik.

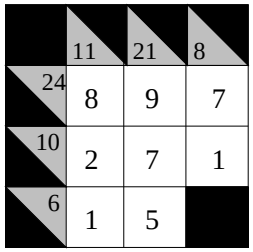
Megjegyzés:

- A címkézéshez (amiről részletesen még nem volt szó) elegendő a `labeling([], Változólista)` eljáráshívás használata.

2. kis házi feladat: számkeresztrejtvény

A megírandó Prolog eljárás és használata

```
% szamker(SzK, Max): SzK az 1..Max számokkal
% helyesen kitöltött számkeresztrejtvény.
% Megjegyzés: egyes sorban/oszlopban középen
% is lehet 'x'!

pelda(mini, [[x\ x,11\x,21\x, 8\x],
             [x\24,   _,   _,   _],
             [x\10,   _,   _,   _],
             [x\6,    _,   _, x\x]], 9).
```




```
| ?- pelda(mini, SzK, _Max), szamker(SzK, _Max).
               SzK = [[x\x, 11\x,21\x,8\x],
                      [x\24,8,   9,   7  ],
                      [x\10,2,   7,   1  ],
                      [x\6, 1,   5,   x\x]] ? ; no
```


Egy összetettebb példa: mágikus sorozatok

- Egy $L = [x_0, \ldots, x_{n-1}]$ egész-sorozat mágikus ha $\forall i$-re $0 \leq x_i \leq n-1$, és $\forall i \in [0, n-1]$-re az i szám pontosan x_i-szer fordul elő L-ben
- **Példa:** n=4 esetén (1,2,1,0) és (2,0,2,0) mágikus sorozatok.
- **Alapvető segédeljárás:** Állapítsuk meg egy `L` egészlistában egy adott `I` érték előfordulásainak számát (`N`)!

```
% pontosan(L, I, N): Az I szám L-ben N-szer fordul elő.
pontosan(L, I, 0) :- outof(I, L).                  (p1)
pontosan([I|L], I, N) :-
        N #> 0, N1 #= N-1, pontosan(L, I, N1).     (p2)
pontosan([X|L], I, N) :-
        N #> 0, X #\= I,   pontosan(L, I, N).      (p3)

% outof(X, L): az X érték nem fordul elő az L listában
outof(_, []).
outof(X, [Y|Ys]) :- X #\= Y, outof(X, Ys).
```

- A `pontosan/3` eljárás spekulatív, azaz **választási pontot** hoz létre:
  - `p1`: az `L` listában `I` egyáltalán nem fordul elő
  - `p2`: az `L` lista feje `I`
  - `p3`: az `L` lista farkában előfordul az `I` szám

Mágikus sorozatok – folytatás

```
% Az L lista egy N hosszúságú mágikus sorozat.
magikus(N, L) :-
        length(L, N), N1 is N-1, domain(L, 0, N1),
        elofordulasok(L, N1, L),
        labeling([], L).              % most felesleges

:- use_module(library(between), numlist/3).

% elofordulasok(+Sorozat, +N1, ?E): N1 = Sorozat hossza - 1.
% Jelölje E_i az E lista i-edik elemét, ekkor
%     E_i = az i előfordulásainak száma Sorozat-ban, i = 0, 1, ..., N1.
elofordulasok(Sorozat, N1, Ek) :-
        numlist(0, N1, Indexek),      % eredménye: Indexek = [0,1,...,N1]
        maplist(pontosan(Sorozat), Indexek, Ek).
```


Mágikus sorozatok: redundáns korlátok

Állítás: Ha az $L = (x_0, \ldots, x_{n-1})$ sorozat mágikus, akkor $\sum_{i<n} x_i = n$, és $\sum_{i<n} i * x_i = n$.

Hatékonyabb változat, a fenti redundáns korlátokkal

```
% N=10 esetén kb. 50-szer gyorsabb az előző programnál!
magikus2(N, L) :-
    length(L, N), N1 is N-1, domain(L, 0, N1),
    osszege(L, S),                    % ∑ L_i = S
    szorzatosszege(L, 0, SP),         % ∑ i*L_i = SP
    call(S #= N), call(SP #= N),      % lásd a megjegyzést
    elofordulasok(L, N1, L).          % lásd az előző változatnál
```

Megjegyzés

- Az aritmetikai beépített eljárások megengednek (aritmetikai) struktúrákat tartalmazó változókat, pl. `Kif = S1+S2, ..., Kif =:= 0`.
- CLPFD-ben ez nem megengedett: `Kif=S1+S2, ..., Kif #= 0` $\Longrightarrow$ `Hiba!` Ennek oka: a korlát-kifejtés csak betöltéskor történik meg.
- A megoldás a korlát-kifejtési fázis késleltetése: `Kif=S1+S2, ..., call(Kif #= 0).`

Mágikus sorozatok: redundáns korlátok (folyt.)

Segédeljárások aritmetikai kifejezések építésére

```
% osszege(L, Ossz): Ossz = ∑ L_i
osszege([], 0).
osszege([X|L], X+S) :- osszege(L, S).

% szorzatosszege(L, I, Ossz): Ossz = I*L_1 + (I+1)*L_2 + ...
szorzatosszege([], _, 0).
szorzatosszege([X|L], I, I*X+S) :-
    J is I+1, szorzatosszege(L, J, S).

| ?- magikus2(4, L).
% visszalépés nélkül adja ki az első megoldást!
 +      1      1 Call: pontosan([_A,_B,_C,_D],0,_A) ?
(...)
?+      1      1 Exit: pontosan([2,0,2,0],0,2) ? z
```


Mágikus sorozatok: redundáns korlátok (folyt. 2)

Globális korlátok (`scalar_product/4`, `sum/3`) használata

```
magikus3(N, L) :-
    length(L, N),
    N1 is N-1, domain(L, 0, N1),
    sum(L, #=, N),                    % ∑ L_i = N
    numlist(0, N1, Indexek),          % Indexek = [0,1,...,N1]
    scalar_product(Indexek,
                   L, #=, N),         % ∑ i*L_i = N
    maplist(pontosan(L), Indexek, L).
```