Nagyhatékonyságú logikai programozás

Jegyzetek a BME informatikus hallgatói számára

- A CLP (Constraint Logic Programming) áttekintése
- CLP segédeszközök SICStusban
- A SICStus `clpfd` könyvtára
- A SICStus `clpq/r` könyvtárai
- A SICStus `clpb` könyvtára
- A Mercury programozási nyelv

Szeredi Péter

IQSOFT Rt.

szeredi@iqsoft.hu

Budapest 1997

Constraint Logic Programming (CLP)

A hagyományos logikai programozás általánosítása:

hívás (szabályban)	---> hívás vagy constraint
egyesítés	---> constraint solving
válasz-behelyettesítés	---> válasz-constraint

A CLP nyelv elemei

Egy CLP nyelvhez kell:

- egy D tartomány (domain),
  pl egészek (N), valósak (R), racionálisak(Q),
  Boole értékek (B),
  listák, füzérek (stringek),
  Prolog-fastruktúrák (Herbrand --- H)

- D-ben definiált függvényeknek egy F halmaza,
  pl +, -, *, \/, /\

- D-ben definiált relációknak (constraint-eknek)
  egy R halmaza pl =, \=, <, eleme

- egy constraint-megoldó algoritmus <D,F,R>-re, azaz a
  D tartományban az R+F -beli jelekből felépített
  constraint-ekre

Példák CLP nyelvekre

CLP(FD)

< {0, 1,..., n}, {+,-,*,/,mod}, {=,=<,eleme,...} >

Constraint satisfaction technikák:
konzisztencia-módszerek, constraint-propagálás, stb.

CLP(B)

< {0,1}, {/\, \/, not, xor}, {=}>

Ítéletkalkulusbeli tételbizonyítási módszerek

CLP(Q)

< Q (racionálisok), {+, -, *, /}, {=, =<, \=, ...} >

Lineáris programozási módszerek:
Gauss elimináció, szimplex módszer

CLP(R)

ugyanez, Q helyett R-rel

CLP szintaxis és deklaratív szemantika

Szintaxis:

program: klózok halmaza.

klóz:

P :- G1, ..., Gn

ahol mindegy Gi vagy cél, vagy constraint.

deklaratív olvasata:

P igaz, ha G1, ..., Gn mind igaz.

kérdés:

?- G1, ..., Gn

válasz egy Q kérdésre:

constraint-eknek egy olyan konjunkciója,
amelyböl a kérdés következik.

CLP procedurális szemantika

Egy 'P :- G1, ..., Gn' klóz procedurális olvasata:

P megoldásához megoldandó G1, ..., Gn.

Végrehajtási állapot: < G, s >

G --- cél/constraint sorozat
s --- constraint-tár: az eddig felhalmozott constraint-ek konjunkciója

Egy végrehajtási alaplépés:

- kiválasztandó egy részcél, P (determinisztikus)

- kiválasztandó egy klóz (nemdeterminisztikus)
  P' :- G1, ..., Gn

- ellenörzendö, hogy P = P' konzisztens a constraint-tárral

Végrehajtás vége: üres célsorozat

A végrehajtás eredménye:

- a constraint-tár, vagy annak a kérdésbeli változókra való 'vetítése'.

Általános következtetési lépések

Rezolució:

< P & G, s > ===>
< P = P' & G1 & ... & Gn & G, s >

ha a programban van egy
P' :- G1, ..., Gn klóz

constraint-megoldás:

< c & G, s > ===>
< G, s /\ c >

ha c konzisztens s-sel

constraint-erösítés:

< C & G, s > ===>
< C' & G, s /\ c >

ha c konzisztens s-sel és
ha s-böl következik, hogy
C ekvivalens (C' /\ c) -vel.

(C' = C is lehet.)

Egy CLP nyelv megvalósítása

Ehhez szükséges definiálni a következöket

- egyszerü constraintek,
  amelyekböl a constraint-tár épül fel

- constraint-termek, azok a kifejezések amelyek
  constraint-ekben elöfordulhatnak

- (általános) constraint-ek és feldolgozásuk

  - egyszerü constraint-ek --- constraint-megoldás

  - nem-egyszerü constraint-ek ---
    constraint-erösítés (korutinok)

- a szükséges algoritmusok:

  constraint-megoldás, constraint-ellenörzés,
  constraint-erösítés, constraint-tár vetítése

CLP megoldó algoritmusok

Alapmüvelet: egy constraint hozzáadása a
constraint-tárhoz, azaz

s' = s /\ c

felépítése, beleértve s/\c konzisztenciájának
ellenörzését.

Követelmények az algoritmussal szemben:

- inkrementalisság (s konzisztenciáját ne
  bizonyítsa újra)

- a visszalépés támogatása

- hatékonyság

- teljesség

CLP segédeszközök SICStusban

Korutinszervezés: Blokk-deklarációk

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

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

Ugyanarra az eljárásra több vagylagos feltétel is szerepelhet, pl.

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

Végtelen választási pontok kiküszöbölése blokk-deklarációval

```
:- block pick(-, ?, -).
pick([X|L], X, L).
pick([Y|L], X, [Y|L1]):-
      pick(L, X, L1).
```

```
perm([], []).
perm(L, [X|P]):-
      pick(L, X, L1), perm(L1, P).
```

A `perm` eljárás visszafelé is működik:

```
| ?- perm(L, [1,2]).
   1  1  Call: perm(_69,[1,2]) ?
   -  -  Block: pick(_363,1,_368)
   2  2  Call: perm(_368,[2]) ?
   -  -  Block: pick(_742,2,_747)
   3  3  Call: perm(_747,[]) ?
   -  -  Unblock: pick(_742,2,[])
   4  4  Call: pick(_742,2,[]) ?
   -  -  Unblock: pick(_363,1,[2])
   5  5  Call: pick(_363,1,[2]) ?
   5  5  Exit: pick([1,2],1,[2]) ?
   4  4  Exit: pick([2],2,[]) ?
   3  3  Exit: perm([],[]) ?
   2  2  Exit: perm([2],[2]) ?
   1  1  Exit: perm([1,2],[1,2]) ?
L = [1,2] ?
```

Korutinszervezés *(folytatás)*

Alternatív lehetőségek

`freeze(X, Hivas)`
`Hivas`t felfüggeszti mindaddig, amig `X` behelyettesítetlen változó. Definíciója lehetne:

```
:- block freeze(-, ?).
freeze(_, Goal) :- call(Goal).
```

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

További alkalmazás:
Generál-és-ellenőriz típusú programok gyorsítása

- ezek általában nem hatékonyak, mert túl sok visszalépést használnak
- korutinszervezéssel a generáló és ellenőrző rész "automatikusan" összefésülhető
- ehhez az ellenőrző részt kell előre tenni és megfelelően blokkolni

```
sort(L, S) :- sorted(L), perm(L, S).

sorted([]).
sorted([_]).
sorted([X,Y|L]):- sorted(L, X, Y).

:- block sorted(?, -, ?), sorted(?, ?, -).
sorted([], X, Y) :- X =< Y.
sorted([Z|L], X, Y) :- X =< Y, sorted(L, Y, Z).
```

Futási idők

Listahossz	7	8	9
gen-test	0.48s	3.86s	35.64s
korutinos	0.04s	0.09s	0.18s
gen-test+korutin	0.55s	4.37s	40.71

Korutinszervezés *(folytatás2)*

`when(Feltétel, Hívás)`
`Hívás`-t mindaddig blokkolja, amig `Feltétel` nem teljesül. `Feltétel` megengedett szintaxisa:

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

`dif(X, Y)`
`X` és `Y` nem egyesíthető. Mindaddig felfüggesztve marad, amig ez el nem dönthető. Definíciója lehetne:

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

`call_residue(Hivás, Maradék)`
`Hivás`-t végrehajtja, és ha a sikeres lefutás után maradnak felfüggesztett hívások, akkor azokat visszaadja `Maradékban`. Pl.

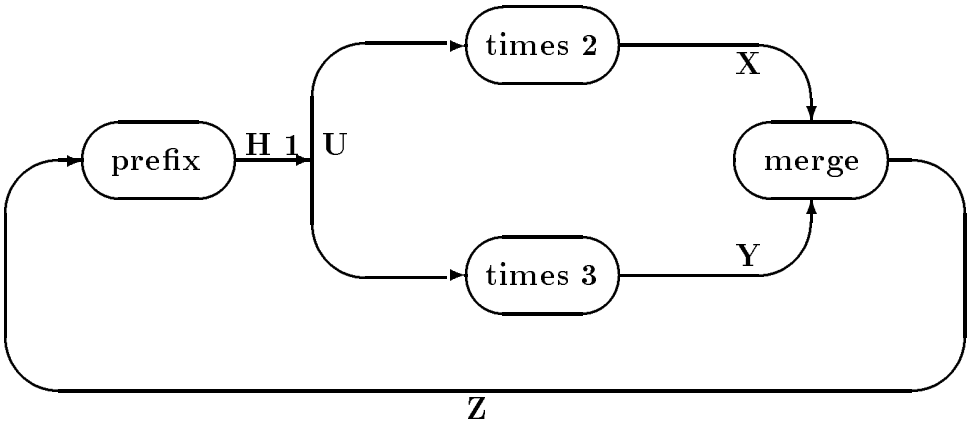
```
| ?- call_residue(dif(X, f(Y)), Maradek).
  Maradek = [[X]-(prolog:dif(X,f(Y)))] ?
  yes
| ?- call_residue((dif(X, f(Y)), X=f(Z)), Maradek).
  X = f(Z),
  Maradek = [[Y,Z]-(prolog:dif(f(Z),f(Y)))] ?
```

Dataflow programozás korutinszervezéssel

Példa: egyszerűsített Hamming feladat

- keressük a $2^i * 3^j (i \geq 1, j \geq 1)$ alakú számok közül az első `N` darabot nagyság szerint rendezve.
- "stream-and-parallelism" közelítésmódot használva korutinszervezéssel egyszerűen lehet megoldani

Hamming probléma




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

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

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

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

További CLP segédeszközök SICStusban

- Tetszőleges nagyságú egész számok

  pl:

```
| ?- fakt(100,F).

F = 933262154439441526816992388562667004907159682643816214685929638952175999932299156089414639761565182862536979208272237582511852109168640000000000000000000000000 ?
```

- Visszaléptethető módon változtatható kifejezések

  `create_mutable(Adat, ValtKif)`

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

  `get_mutable(Adat, ValtKif)`

  `Adat`-ba előveszi `ValtKif` pillanatnyi értékét.

  `update_mutable(Adat, ValtKif)`

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

- Mellékhatás visszacsinálása visszalépéskor

  `undo(Hivas)`

  Feltétel és mellékhatás nélkül sikerül. Visszalépéskor végrehajtódik a `call(Hivas)` hívás, és a visszalépés folytatódik.

  Példa:

```
assert_b(Cl) :- assert(Cl), undo(retract(Cl)).
```

  A vágók nem bántják, nem úgy mint:

```
rosszabb_assert_b(Cl) :-
      ( assert(Cl)
      ; retract(Cl), fail
      ).
```

Egy egyszerű CLP feladat: térképszinezés

Tiszta Prolog megoldás: visszalépéses keresés

```
terkep_pl(Szk, Ok) :- szinez(Szk), write(Ok), nl.

% szinez(Szk): Az Szk szomszédságok 3 színnel színezhetök
szinez([]).
szinez([P1-P2|Szk]) :- ne(P1, P2), szinez(Szk).

% ne(P1, P2): P1 és P2 különbözö színek.
ne(1,2). ne(1,3). ne(2,1). ne(2,3). ne(3,1). ne(3,2).

adat(sarkany1, [X-Y,Z-V,V-V1,V1-V2,/*...,Vn,*/X-Z,Z-Y,Y-V],
               [X,Y,Z,V,V1,V2/*,...,Vn*/]).
```

Futási idő, n=16: `sarkany1`: 10.68 sec

Korutinos megoldás: `dif/2` segítségével

```
terkep_dif(Szk, Ok) :-
  dif_feltetelek(Szk), ertekek(Ok),
  write(Ok), nl.

dif_feltetelek([]).
dif_feltetelek([P1-P2|Szk]) :- dif(P1, P2), dif_feltetelek(Szk).

ertekek([]).
ertekek([V|Vk]) :-
  szin(V), ertekek(Vk).

szin(1). szin(2). szin(3).

adat(sarkany2, [X-Y,Z-V,V-V1,V1-V2,/*...,Vn,*/X-Z,Z-Y,Y-V],
               [X,Y,V1,V2/*,...,Vn*/,Z,V]).
```

Futási idő, n=16: `sarkany1`: 0.00 sec, `sarkany2`: 35.45 sec

FD változók megvalósítása mutábilisekkel

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

% uj_valtozo(EL, CV, V): EL érték-készlettel létrehoz egy új
% CV-változót (CV), az ennek megfelelö Prolog változó a V.
uj_valtozo(E, V=ME, V) :- create_mutable(E, ME).

% erteke(CV, E): CV-nek konkrét értéke van, E.
erteke(V=_Ert, V) :- nonvar(V).

% valtozoja(CV, V): V a CV-hez tartozó változó.
valtozoja(V=_, V).

% kihagy(CV, E): CV tartományából kihagyja E-t.
kihagy(V=Ert, E) :-
  get_mutable(EL, Ert), delete(EL, E, UjEL),
  update_mutable(UjEL, Ert),
  (   UjEL = [VE] -> V = VE
  ;   UjEL \== []           ).

% valaszt(CV): CV-t a lehetséges értékeivel helyettesíti.
valaszt(V=Ert) :-
  get_mutable(EL, Ert), member(V, EL), update_mutable([V], Ert).

% valaszt(CVk0, CVk) :- értéket ad a CVk0-beli elsö értékkel
% nem bíró változónak ha van. CVk a fennmaradó változók listája.
valaszt([CV|CVk0], CVk) :-
  erteke(CV, _), !, valaszt(CVk0, CVk).
valaszt([CV|CVk0], CVk0) :-
  valaszt(CV).
valaszt([], []).

% kihagyb(CV, E): CV tartományából kihagyja E-t.
% Pontosan ugyanaz, mint kihagy, csak blokk-deklarációval.
:-block kihagyb(?, -).
kihagyb(V=Ert, E) :- ...
```

CLP(fd) megoldás: mutábilis kifejezések használata

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

terkep_fd(Szk, Ok) :-
  valtozok(Ok, Vk), megold(Szk, Ok),
  write(Vk), nl.

% valtozok(CVk, Vk):
% A CVk constraint-változók megfelelöi a Vk változók
valtozok([], []).
valtozok([CV|CVk], [V|Vk]) :-
  uj_valtozo([1,2,3], CV, V), valtozok(CVk, Vk).

% A Ck constraint-ek a CVk változókkal megoldhatók
megold([], []).
megold(Ck, CVk) :-
  valaszt(CVk, CVk1), szur(Ck, Ck1), megold(Ck1, CVk1).

% A Ck0 constraint-ek közul kihagyva a teljesülteket
% kapjuk a Ck constraint-listát, Ck tovább nem szürhetö.
szur(Ck0, Ck) :-
  szur1(Ck0, Ck1),
  (   Ck1 = Ck0 -> Ck = Ck0
  ;   szur(Ck1, Ck)
  ).

% szur1(Ck0, Ck): Ck0-ból kihagyva a teljesülteket kapjuk Ck-t.
szur1([], []).
szur1([CV1-CV2|Ck0], Ck) :-
  (   erteke(CV1, E) -> kihagy(CV2, E),  szur1(Ck0, Ck)
  ;   erteke(CV2, E) -> kihagy(CV1, E),  szur1(Ck0, Ck)
  ;   Ck = [CV1-CV2|Ck1], szur1(Ck0, Ck1)
  ).
```

Futási idő, n=16: `sarkany2`: 0.02 sec, `sarkany3`: 53.23 sec

CLP(fd) jellegű megoldás: `freeze` használata

```
terkep_fdfreeze(Szk, Ok) :-
  valtozok(Ok, Vk),
  feltetelek(Szk),
  behelyettesit(Ok),
  write(Vk), nl.

% feltetelek(Ck): A Ck constraint-eket felveszi.
feltetelek([]).
feltetelek([CV1-CV2|Ck]) :-
  feltetel(CV1, CV2), feltetel(CV2, CV1),
  feltetelek(Ck).

% feltetel(CV1, CV2): ha CV1 és CV2 értéke különböző.
feltetel(CV1, CV2) :-
  valtozoja(CV1, V1), kihagyb(CV2, V1).

% behelyettesit(CVk): CVk-et a lehetséges értékeikkel
% helyettesíti.
behelyettesit([]).
behelyettesit(CVk0) :-
  valaszt(CVk0, CVk), behelyettesit(CVk).

adat(sarkany3, [X-Y,V-V1,V1-V2,/*...,Vn,*/Z-V,X-Z,Z-Y,Y-V],
               [X,V1,V2/*,...,Vn*/,Y,Z,V]).
```

Futási idő, n=16: `sarkany3`: 49.05 sec

CLP(fd) jellegű megoldás: *first-fail* elv

```
terkep_fdfreezeff(Szk, Ok) :-
  valtozok(Ok, Vk), feltetelek(Szk),
  ff_behelyettesit(Ok),              % csak ez változott
  write(Vk), nl.

% ff_behelyettesit(CVk): az összes CVk-t lehetséges
% értékével helyettesíti.
ff_behelyettesit([]).
ff_behelyettesit(CVk0) :-
  ff_valaszt(CVk0, CVk),
  ff_behelyettesit(CVk).

% Értéket ad a CVk0-ban levö legszükebb tartományú változónak.
ff_valaszt(CVk0, CVk) :-
  hosszak(CVk0, H_CVk), H_CVk \== [], !,
  keysort(H_CVk, [_-CV|R_CVk]),
  hossztalan(R_CVk, CVk), valaszt(CV).
ff_valaszt(_, []).

hosszak([], []).
hosszak([CV|CVk], [H-CV|H_CVk]) :-
  CV=(_=Ert), get_mutable(E, Ert), length(E, H),
  hosszak(CVk, H_CVk).

hossztalan([], []).
hossztalan([_-CV|H_CVk], [CV|CVk]) :-
  hossztalan(H_CVk, CVk).

adat(sarkany4, [X-Y,V-V1,V1-V2,/*...,Vn,*/Z-V,X-Z,Z-Y,Y-V,X-V],
               [/*Vn,...*/V2,V1,X,Y,Z,V]).
```

Futási idő, n=16: `sarkany3`: 0.04 sec, `sarkany4`: 460.93 sec

CLP(fd) jellegű megoldás: *first-fail*+*most-constrained* elv (ffc)

```
terkep_fdfreezeffc(Szk, Ok) :-
  valtozok(Ok, Vk),
  feltetelek(Szk),
  c_rendez(Ok, Szk, Ok1),        % ez új
  ff_behelyettesit(Ok1),
  write(Vk), nl.

% CVk1 a CVk változók átrendezése, úgy hogy a Ck
% constraint-ekben többször elöfordulók legyenek elöbb.
c_rendez(CVk, Ck, CVk1) :-
  elofordulasok_m(CVk, Ck, E_CVk),
  keysort(E_CVk, R_CVk), hossztalan(R_CVk, CVk1).

% elofordulasok_m(CVk, Ck, E_CVk): E_CVk a CVk-nek a Ck-ben
% való elöfordulásszámai negáltjával párosított listája.
elofordulasok_m([], _, []).
elofordulasok_m([CV|CVk], Ck, [E-CV|E_CVk]) :-
  elofordulasai(Ck, CV, E0), E is -E0,
  elofordulasok_m(CVk, Ck, E_CVk).

% elofordulasai(Ck, CV, E) : CV Ck-ben E-szer fordul elö.
elofordulasai([], _, 0).
elofordulasai([V1-V2|Ck], CV, E) :-
  elofordulasai(Ck, CV, E0),
  (   azonos(V1, CV) -> E is E0+1
  ;   azonos(V2, CV) -> E is E0+1
  ;   E = E0
  ).

azonos(V1=_, V2=_) :- V1==V2.
```

Megjegyzés: a constraint-ek szerinti rendezés itt statikus, de általában dinamikus kell legyen (behelyettesítéskor változik).

Futási idő, n=16: `sarkany3`: 0.04 sec, `sarkany4`: 460.93 sec

`library(clpfd)`-re épülő megoldások

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

terkep_clpfd(Lab, Szk, Ok) :-
  domain(Ok, 1, 3), clpfd_feltetelek(Szk),
  labeling(Lab, Ok), write(Ok), nl.

clpfd_feltetelek([]).
clpfd_feltetelek([P1-P2|Szk]) :-
  P1 #\= P2, clpfd_feltetelek(Szk).
```

`labeling` lehetőségek

- `Lab = []` $\longrightarrow$ `terkep_fdfreeze` (fd mut+korutin)
- `Lab = [ff]` $\longrightarrow$ `terkep_fdfreezeff` (fd mut+korutin+ff)
- `Lab = [ffc]` $\longrightarrow$ `terkep_fdfreezeffc` (fd mut+korutin+ffc)

Futási idők összefoglalása (Sequent Symmetry 486/50 MHz)

variáns/adat	sarkany1	sarkany2	sarkany3	sarkany4	lanc
visszalépés	10.68 sec	10.68 sec	8.58 sec	170.11 sec	0.70 sec
dif	0.00 sec	35.45 sec	80.35 sec	813.25 sec	7.63 sec
fd mutábilisekkel	0.03 sec	0.02 sec	53.23 sec	498.64 sec	5.12 sec
fd mut+kortn	0.02 sec	0.01 sec	49.05 sec	370.75 sec	4.68 sec
fd mut+kortn+ff	0.02 sec	0.03 sec	0.04 sec	460.93 sec	12.10 sec
fd mut+kortn+ffc	0.05 sec	0.04 sec	0.05 sec	0.04 sec	13.20 sec
clp(fd), lab=[]	0.03 sec	0.03 sec	140.24 sec	941.07 sec	9.40 sec
clp(fd), lab=[ff]	0.04 sec	0.03 sec	0.03 sec	997.07 sec	9.59 sec
clp(fd), lab=[ffc]	0.06 sec	0.05 sec	0.06 sec	0.03 sec	11.73 sec

```
adat(lanc, [X0-Z0,X0-Y0,Y0-Z0,Z0-X1,Y0-X1, ...., X10-X0],
           [X0,Y0,Z0, ...,X9,Y9,Z9,X10]).
```

SICStus `clpfd` könyvtár

A `clpfd` kiterjesztés elemei (első közelítés)

- Tartomány: egészek (negatívak is!)
- Függvények (aritmetika):

  `+ - * / mod min max` (kétargumentumúak),
  `abs` (egyargumentumú).
- Constraint-relációk

  aritmetikaiak: `#<, #>, #=<, #>=, #= #\=`

  halmazműveletek: *X* `in` *Halmaz*, `domain([`*X*`,...], `*Min*`, `*Max*)
  Halmaz lehet: halmaz: {*Szám* ...}, intervallum: *Min* `..` *Max*,
  metszet: *Halmaz* `/\` *Halmaz*, únió: *Halmaz* `\/` *Halmaz*,
  komplemens: `\` *Halmaz*
  Min: *Szám* vagy `inf`
  Max: *Szám* vagy `sup`
- Primitív constraint-ek (constraint tár elemei): *X* `in` *Halmaz*
- Constraint-megoldó algoritmus:

  aritmetikaiak: constraint-szűkítés: ún. intervallum-konzisztencia
  halmazműveletek: teljes konzisztencia (ún. tartomány-konzisztencia)

Példa

```
| ?- 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 ?
yes
| ?- X in 10..20, X #\= 15, Y in {2}, Z #= X*Y.
Y = 2, X in(10..14)\/(16..20),
Z in 20..40 ?
yes
| ?-
```

Constraint feladatok megoldása

A clp(FD) programok tipikus felépítése

- deklaráljuk a változók tartományait
- állítsuk fel a constraint-eket
- helyettesítsük be a constraint-változókat (címkézés)

Címkézés

- `indomain(`*X*`)`: X-et a tartománya által megengedett értékkel helyettesíti (visszalépéssel felsorolja as összes értéket)
- `labeling(`*Opciók*, *Változók*`)`: A *Változók* lista minden elemét behelyettesíti, az *Opciók* lista által előírt módon, pl, `[]`: balról jobbra.

Példa: Kódaritmetika: SEND+MORE=MONEY

```
send(SEND,MORE,MONEY):-
  List= [S,E,N,D,M,O,R,Y], domain(List, 0, 9),
  alldiff(List), S #\= 0, M#\= 0,
  SEND #=  1000*S+100*E+10*N+D,
  MORE #= 1000*M+100*O+10*R+E,
  MONEY #= 10000*M+1000*O+100*N+10*E+Y,
  SEND+MORE #= MONEY, labeling(List).

% labeling(Xs) ekvivalens a beépített labeling([],Xs) eljárással
labeling([]).
labeling([X|Xs]) :- indomain(X), labeling(Xs).

% Azonos a beépített all_different eljárással
alldiff([]).
alldiff([X|Xs]) :- outof(X, Xs), alldiff(Xs).

outof(_, []).
outof(X, [Y|Ys]) :- X #\= Y, outof(X, Ys).
```

A zebra feladat

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

zebra(ZOwner, All):-
  All = [England,Spain,Japan,Norway,Italy,
         Green,Red,Yellow,Blue,White,
         Painter,Diplomat,Violinist,Doctor,Sculptor,
         Dog, Zebra, Fox, Snail, Horse,
         Juice,Water,Tea,Coffee,Milk],
  domain(All, 1, 5),
  alldiff([England,Spain,Japan,Norway,Italy]),
  alldiff([Green,Red,Yellow,Blue,White]),
  alldiff([Painter,Diplomat,Violinist,Doctor,Sculptor]),
  alldiff([Dog,Zebra,Fox,Snail,Horse]),
  alldiff([Juice,Water,Tea,Coffee,Milk]),
  England = Red,          Spain = Dog,
  Japan = Painter,        Italy = Tea,
  Norway = 1,             Green = Coffee,
  Green #= White+1,       Sculptor = Snail,
  Diplomat = Yellow,      Milk = 3,
  Violinist = Juice,      nextto(Norway, Blue),
  nextto(Fox, Doctor),    nextto(Horse, Diplomat),
  labeling([ff], All),
  nth(N, [England,Spain,Japan,Norway,Italy], Zebra),
  nth(N, [england,spain,japan,norway,italy], ZOwner).

nextto(A, B) :-  A #= B+1.
nextto(A, B) :-  A #= B-1.
```

N királynő a sakktáblán

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

% A Qs lista N királynő biztonságos elhelyezését
% mutatja egy N*N-es sakktáblán.  Ha a lista
% i. eleme j, akkor az i. királynőt az i. sor j.
% oszlopába kell helyezni.
queens(N, Qs):-
      length(Qs, N), domain(Qs, 1, N),
      safe(Qs),
      labeling([ff],Qs).                  % first-fail elv

% safe(Qs):  A Qs lista a királynők biztonságos
% elhelyezését írja le.
safe([]).
safe([Q|Qs]):-
      no_attack(Qs, Q, 1),
      safe(Qs).

% no_attack(Qs, Q, I): A Qs lista által leírt
% királynők egyike sem támadja a Q oszlopban levő
% kiralynőt, feltéve hogy Q és Qs távolsága I.
no_attack([],_,_).
no_attack([X|Xs], Y, I):-
      no_threat(X, Y, I),
      J is I+1, no_attack(Xs, Y, J).

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

A constraint-ek működése

A constraint erősítés fajtái

Legyen C egy n-változós constraint; jelöljük *D(X,s)*-sel az X változó tartományát az s constraint-tárban.

- Tartomány-konzisztencia (domain consistency).

  C tartomány-konzisztens s-re nézve, ha bármelyik Xi változójához és annak tetszőleges $Vi \in D(Xi,s)$ megengedett értékéhez található a többi változónak olyan $Vj \in D(Xj,s)$ értéke ($j = 1, \ldots, i\text{-}1, i\text{+}1, \ldots, n$), hogy *C(X1, ... Xn)* fennálljon.

  Jelöljük *D'(X,s)*-sel a *min(D(X,s)) .. max(D(X,s))* intervallumot.

- Intervallum-konzisztencia (interval consistency).

  C intervallum-konzisztens s-re nézve, ha bármelyik Xi változójához és tartománya mindkét végpontjához, azaz a $Vi = min(D(X,s))$ és $Vi = max(D(X,s))$ értékekhez található a többi változónak olyan $Vj \in D'(Xj,s)$ értéke ($j = 1, \ldots, i\text{-}1, i\text{+}1, \ldots, n$), hogy *C(X1, ... Xn)* fennálljon.

- Az aritmetikai constraint-ekre a SICStus clpfd könyvtár csak intervallum konzisztenciát garantál, de bizonyos esetekben tartomány-konzisztenciát is biztosít.

Példák

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

| ?- X in 1..10, U=3, X #\= Z-U, Z=8.     % tartomány-konz.:
                U = 3, Z = 8, X in(1..4)\/(6..10) ?
| ?- X in 1..10, X #\= Z-U, Z=8, U=3.     % intervallum-konz.:
                U = 3, Z = 8, X in 1..10 ?
```

Példa: mágikus sorozatok

- Egy $L = (x_0, \ldots, x_{n-1})$ sorozat mágikus ($x_i \in [0..n-1]$), ha L-ben az i szám pontosan x_i-szer fordul elő (minden $i \in [0..n-1]$-re).
- Példa: n=4 esetén (1,2,1,0) és (2,0,2,0) mágikus sorozatok.

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

% elofordulasok([Ei, ...], i, Sor): Sor-ban i, i+1, ...
% elöfordulásainak száma Ei, Ei+1, ...
elofordulasok([], _, _).
elofordulasok([E|Ek], I, Sor) :-
        pontosan(I, Sor, E), J is I+1, elofordulasok(Ek, J, Sor).

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

% Hatékonyság-fokozás redundáns (surrogate) constraint-ekkel.
magikus1(N, L) :-
        length(L, N), N1 is N-1, domain(L, 0, N1),
        osszege(L, S), call(S #= N),
        szorzatosszege(L, 0, SS), call(SS #= N),
        elofordulasok(L, 0, L).

osszege([], 0).
osszege([X|L], X+S) :- osszege(L, S).

szorzatosszege([], _, 0).
szorzatosszege([X|L], I, I*X+S) :-
        J is I+1, szorzatosszege(L, J, S).
```

Reifikáció

Gondok a pontosan/3 eljárás előbbi megvalósításával

- harmadik argumentumának értéket ad, nem kezeli constraint-változóként,
- választási pontokat hoz létre

Egy constraint reifikációja

- A constraint igazságértékének "tükrözése" egy 0-1 értékű constraint-változóban.
- Példa: `(X #= 3) #<=> B`: `B` az `X=3` egyenlőség igazságértéke.
- Az aritmetikai és halmaz-constraintek mind reifikálhatóak

Reifikált constraint-ek végrehajtása

- egy démon figyeli, hogy a constraint-tárból következik-e, ha igen, `B=1`;
- egy démon figyeli, hogy következik-e a negáltja, ha igen, `B=0`;
- egy harmadik démon figyeli, hogy `B` értéket kap-e, ha `B=1`, felveszi a constraint-t, ha `B=0`, felveszi a negáltját a constraint-tárba.

Constraint-levezethetőség (entailment)

- **Tartomány-levezethetőség**: *C tartomány-levezethető s*-ből, ha Xi változói minden megengedett $Vj \in D(Xj,s)$ értékére ($j = 1, \ldots, n$), *C(X1, ... Xn)* fennáll.
- **Intervallum-levezethetőség**: *C intervallum-levezethető s*-ből, ha minden $Vj \in D'(Xj,s)$ értékre ($j = 1, \ldots, n$), *C(X1, ... Xn)* fennáll.

A reifikált aritmetikai constraint-ek intervallum-levezethetőséget, a reifikált halmaz-constraintek tartomány-levezethetőséget garantálnak.

Példák tüközött constraint-ekre

```
| ?- X in 1..4, X #< Y #<=> B, X+Y #=9.
        B = 1, X in 1..4, Y in 5..8 ?
| ?- X+Y #= Z #<=> B, X=1, Z=6, Y in 1..10, Y#\=5.
        X = 1, Z = 6, Y in(1..4)\/(6..10), B in 0..1 ?
```

pontosan/3 tükrözött constraint-ekkel

```
pontosan(_, [], 0).
pontosan(I, [X|L], N) :-
        X #= I #<=> B, N #= N1+B, pontosan(I, L, N1).
```

A tükrözés általánosítása: logikai müveletek constraint-ekkel

- `#\ Q`: `Q` negáltja; `P #/\ Q`: konjunkció; `P #\/ Q`: diszjunkció; `P #\ Q`: kizáró vagy; `P #=> Q` vagy `Q #<= P`: implikáció; `P #<=> Q`: ekvivalencia.
- `P` és `Q` tetszőleges reifikálható constraint (pl aritmetikai), vagy 0-1 értékű constraint-változó
- a logikai müveletek reifikált és aritmetikai constraint-ekre vezethetők vissza, pl: `X#=4 #\/ Y#>6` $\longrightarrow$ `X#=4#<=>B1, Y#>6#<=>B2, B1+B2 #>0`

A reifikáció hatékonysága: mágikus sorozatok (586/133Mhz)

variáns/adat	n=8	n=10	n=12	n=14	n=20	n=30
választós	10.17	227.96				
választós+osszege	1.35	6.26	28.69	127.29		
vál.+szorzatosszege	0.27	0.66	1.32	2.54		
vál.+ossz.+szorzossz.	0.20	0.48	1.02	1.90	10.59	108.94
reifikálós	1.20	2.76				
reifikálós+osszege	0.47	0.95	1.73	2.54		
reif.+szorzatosszege	0.20	0.40	0.54	0.72		
reif.+ossz.+szorzossz.	0.25	0.43	0.62	0.82	1.89	4.83

Constraintek feldolgozása

A feldolgozás fázisai

- A constraint kifejtése elemi constraintekre
  - alapértelmezés: fordítási időben;
  - `call/1`-be ágyazással elhalasztható a constraint végrehajtásáig.
  - kisérletezés: `goal_expansion(`*Constraint*`, user, Kifejtés)`
- A constraint végrehajtása (posting)
  - megállapítjuk mikor kell felébreszteni; (a változói alsó, felső határának változásakor, bármilyen változáskor vagy értékadáskor);
  - felébresztjük a constraintet.
- A constraint felébresztése = constraint-erősítés

Elemi constraintek

- aritm.:`'x+y=t'/3 'x*y=z'/3 'x/y=z'/3 'x mod y=z'/3 '|x|=y'/2`
- összehasonlítás: `'x=y'/2, 'x=<y'/2, 'x\=y'/2` és reifikált változataik: `iff_aux('x` *Rel* `y'(X,Y), B)`, ahol *Rel* a `= =< \=` jelek valamelyike.
- Optimalizálások: `x*x=y'/2 'ax=t'/3 'ax+y=t'/4 'ax+by=t'/5 't+u=<c'/3 't=u+c'/3 't=<u+c'/3 't\=u+c'/3 't>=c'/2` stb.
- Példák:
  `abs(X-Y)#>1` $\longrightarrow$ `'x+y=t'(Y,A,X),'|x|=y'(A,B),'t>=c'(B,2)`
  `(X#<3)#<=>B` $\longrightarrow$ `iff_aux('x=<y'(X,2),B)`
  `X*X+Y*Y#=<16` $\longrightarrow$ `'x*x=y'(X,A),'x*x=y'(Y,B),'t+u=<c'(A,B,16)`

Constraintek kifejtése

- lineáris kifejezés esetén megmarad az intervallum-konzisztencia;
- általános esetben nem, pl `X in 0..10, X*X*X*X#=16.` $\longrightarrow$ `X in 1..4`
- de sokszor a nem-lineárisra is jó, pl. prímszámok ellenőrzésére:
  `domain([X,Y], 0, 5000), X-Y#>1, X*X-Y*Y#=6211.` $\longrightarrow$ `no`

Constraint erősítés ($< C, s > \longrightarrow < C', s \wedge c >$ **ahol** $s \models C \equiv C' \wedge c$)**; használt speciális esetei:**

- ha $s \wedge c$ inkonzisztens, akkor visszalépünk;
- ha $C' =$ `true`, a constraint levezethetővé vált, azaz sikeresen lefutott;
- ha $C' \neq C$, akkor C lefutott, helyette végrehajtjuk C'-t, pl. reifikációnál, vagy `max(X,Y) #= Z` $\longrightarrow$ `X #= Z`, ha `Y #=< X`;
- ha $C' = C$, akkor (újra) elaltatjuk a constraintet.

Constraint erősítés megvalósítása

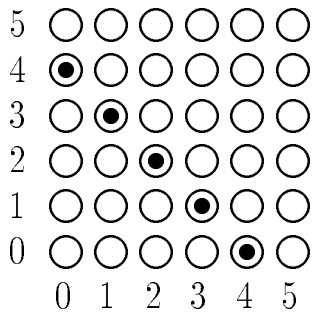
- cél: az adott constraint és egy adott tár esetén kiszűrni azokat az értékeket, amelyekre a constraint nem teljesülhet.
- közvetlen (nem reifikált) constraintek esetén képezzük a constraint és a tár metszetét, majd:
  - tartomány-konzisztencia esetén: a változó-tartományok minden pontját próbáljuk szűrni, pl. `abs(X-Y)#>1, Y in 2..4` $\rightarrow$ `X#\=3`;
  - intervallum-konzisztencia biztosításakor: a tartomány végpontjait próbáljuk kiszűrni, a többi változó tartományát intervallumnak tekintve (ha sikerül, ismételjük);
- reifikált constraintek esetén:
  - ha a logikai változó értéket kap, a kérdezett constraint vagy negáltja lép a reifikált constraint helyébe;
  - vizsgáljuk, hogy a kérdezett constraint vagy negáltja a tárból levezethetővé válik-e:
    - tartomány-levezethetőség vizsgálata: ha minden a tár által megengedett behelyettesítésre igaz a constraint, pl. `Y in {0,5}, X in 2..3` tárból `abs(X-Y)#>1` tartomány-levezethető;
    - intervallum-levezethetőség vizsgálata: ha a tárban minden tartományt intervallummá általánosítunk, akkor is mindig igaz, pl `abs(X-Y)#>1` a fenti tárból nem intervallum-levezethető.

    Ha a kérdezett constraint vagy a negáltja levezethetővé válik, a logikai változó szűkül: a 0 vagy 1 értéket kapja.

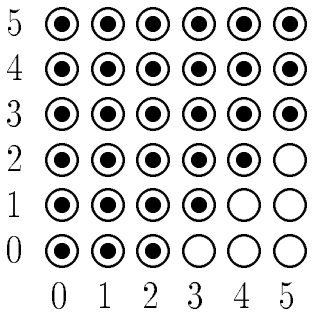
`domain([X,Y], 0, 5), C(X,Y), X#=<Y+2, X#>2, Y in {0,4,5}`

`C(X,Y)`:

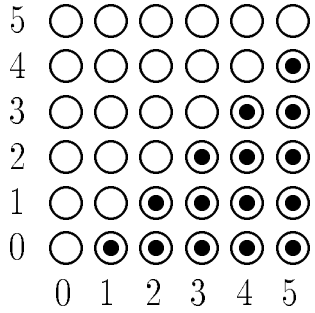
`X + Y #= 4`



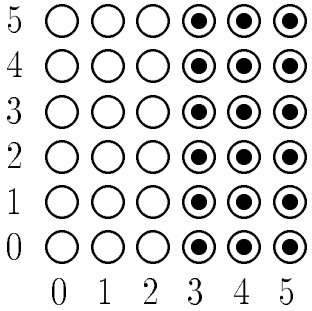
`X #=< Y+2`



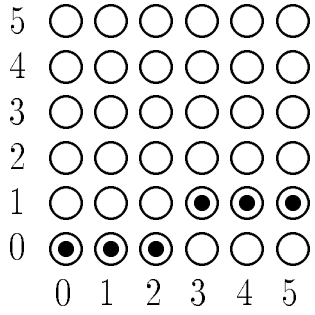
`X #> Y`



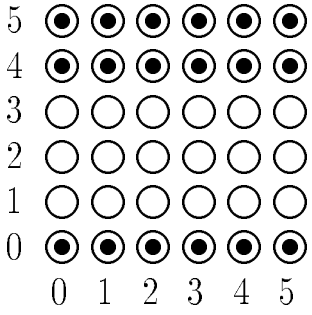
`X #> 2`



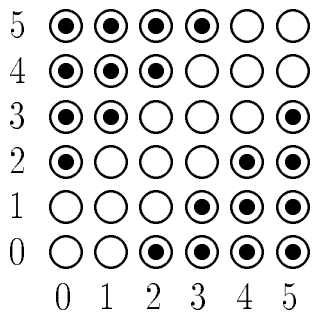
`Y #= X/3`
vagy
`X#>=3 #<=> Y`



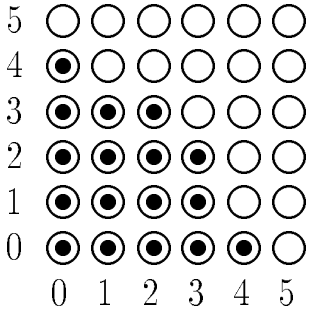
`Y in {0,4,5}`



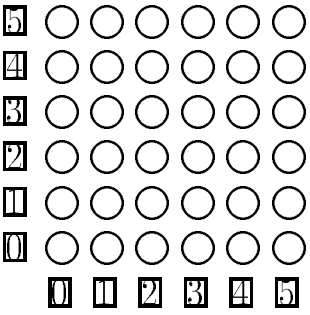
`abs(X-Y)#>1`
vagy
`X-Y#=Z #∧`
`Z #>1 #∨`
`Z #< -1`



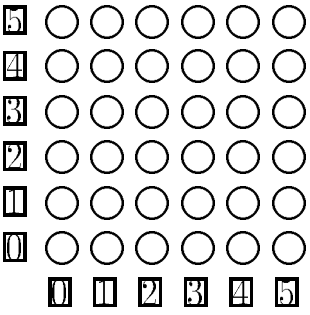
`X*X+Y*Y#=<16`



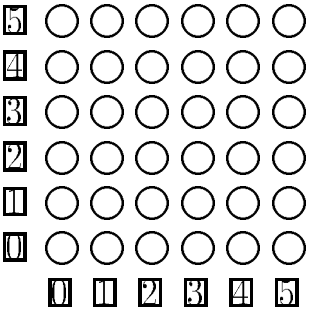
Constraint tárak



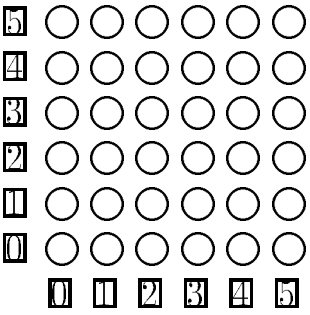
```
X+Y #= 4, X#=<Y+2, X#>2
```



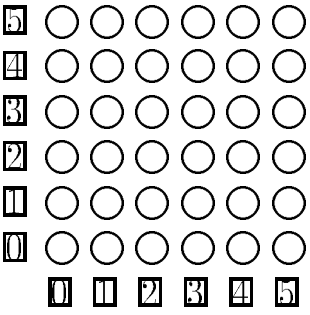
```
X #> Y, X#=<Y+2, X#>2, Y in {0,4,5}
```



```
X#>=3 #<=> Y, X#=<Y+2, X#>2
```



```
abs(X-Y)#>1, X#=<Y+2, X#>2, Y in {0,4,5}
```



```
X*X+Y*Y#=<16, X#=<Y+2, X#>2
```

Kombinatorikus (szimbolikus) constraintek

Ezek a constraintek nem reifikálhatóak.

`element(X, CLista, Y)`
Jelentése: `CLista` `X`-edik eleme `Y`, ahol `X` és `Y` constraint-változók (vagy egészek), `CLista` egészekből álló lista (#6-ban lehetnek benne változók is). Tartomány-konzisztenciát biztosít. Példák:

```
| ?- element(X, [0,1,2,3,4], Y), X in {2,5}.  % Y #= X-1
                         X in{2}\/{5}, Y in{1}\/{4} ?

% X #= c #<=> B megvalósítása, 1 =< X,c =< 6 esetére.
beq(X, C, B) :-
  X in 1..6, call(I #= X+6-C),
  element(I, [0,0,0,0,0,1,0,0,0,0,0], B).
```

`relation(X, Hozzárendelés, Y)`
Itt `X` és `Y` constraint-változók (vagy egészek), `Hozzárendelés` *egész* - *Halmaz* alakú párok listája (ahol mindegyik *egész* csak egyszer fordulhat elő). Jelentése: `Hozzárendelés` tartalmaz egy `X-Halmaz` párt, ahol `Y` eleme a `Halmaz`-nak. Tetszőleges bináris reláció definiálására használható. Tartomány-konzisztenciát biztosít. Példa:

```
'abs(x-y)>1'(X,Y) :-
  relation(X, [0-(2..5), 1-(3..5), 2-{0,4,5},
               3-{0,1,5}, 4-(0..2), 5-(0..3)], Y).

'abs(x-y)>1'(X,Y), X in 2..3. → Y in(0..1)∨(4..5)
```

`all_different(Változók)`
A Változók páronként különbözőek. Megvalósítása nem teljes, a korábbi `alldiff` eljárással azonos.

`all_distinct(Változók)`
Jelentése ugyanaz, mint `all_different`, de teljes (tartomány-konzisztenciát biztosít). A `library('clpfd/all_dist')` könyvtárban van. Például:

```
| ?- domain([X,Y,Z], 1, 2), all_distinct([X,Y,Z]).
no
```

Címkéző eljárások

`indomain(X)` — korábban ismertettük.

`minimize(Cél, X)` ill. `maximize(Cél, X)`
A `Cél` ismételt hívásával megkeresi az `X` változó minimális ill. maximális értékét. Például:

```
| ?- domain([X,P], 2, 4000), P*X#=7373,
     minimize(labeling([], [P,X]), P).
```

`labeling(Opciók, Változók)`:
A `Változók` lista minden elemét behelyettesíti, az `Opciók` lista által előírt módon. Az alábbi négy csoport mindegyikéből legfeljebb egy opció szerepelhet.

- Változó kiválasztása:
  - `leftmost` — a legelső változó (alapértelmezés);
  - `min` ill `max` — a legkisebb alsó határú ill. a legnagyobb felső határú;
  - `ff` — first-fail elv: a legkisebb tartományú változók közül a legbaloldalibb;
  - `ffc` — a legkisebb tartományúak közül a legtöbb constraintben előforduló (ha több van, ezek közül a legbaloldalibb).
- A tartomány felbontása a kiválasztott `X` változóra:
  - `step` — `X #= B` és `X #\= B` közötti választás, ahol `B` az `X` tartományának alsó vagy felső határa (alapértelmezés);
  - `bisect` — `X #< M` és `X #>= M` közötti választás, ahol `M` az `X` tartományának középső eleme.
- A tartomány bejárási iránya:
  - `up` — alulról felfelé (alapértelmezés);
  - `down` — felülről lefelé.
- A keresett megoldások :
  - `all` — visszalépéssel az összes megoldást felsorolja; (alapértelmezés)
  - `minimize(X)` ill. `maximize(X)` — csak egy megoldást keres, `X`-re vonatkozóan a minimálisat ill. a maximálisat.

Egyéb eljárások

Tartományok lekérdezése, statisztika, válaszok

- `fd_min(X, Min)`; `fd_max(X, Max)`; `fd_size(X, Size)`: Az `X` változó tartományának pillanatnyi alsó határa `Min`, felső határa `Max`, mérete `Size`. Ha valamelyik nem korlátos, az eljárás meghiúsul.
- `fd_dom(X, Dom)`; `fd_set(X, Set)`: `Dom` az `X` tartományát leiró halmazkifejezés (#6-ban megszűnik). `Set` az `X` tartományának belső ábrázolása, `[[`L_1`|`U_1`], [`L_2`|`U_2`], ...]` alakban (#6-ban ADT-vé válik).
- `fd_statistics(Kulcs, Érték)`: A `Kulcs`-hoz tartozó számláló `Érték`-ét kiadja és lenullázza. Lehetséges kulcsok és számlált események: `constraints` — constraint létrehozása; `tells` — közvetlen constraint felébresztése; `asks` — reifikált constraint felébresztése; `tell_entailments` — közvetlen constraint levezethetővé válása; `ask_entailments` — reifikált constraint levezethetővé válása; `prunings` — tartomány szűkítés; `backtracks` — a tár ellentmondásossá válása.
- `fd_statistics`: az összes számláló állását kiírja és lenullázza őket.
- `clpfd:full_answer`: ez egy dinamikus kampó eljárás. Ha nem sikerül, akkor a rendszer egy kérdésre való válaszoláskor csak a kérdésben előforduló változókat írja ki, és a még alvó constrainteket nem írja ki. Ha sikerül, kiírja az összes változót és a le nem vezetett constrainteket.

Ütemezési (szimbolikus) constraint

- `cumulative(Starts, Durations, Resources, Limit)`: az első három argumentum változókból/egészekből álló egyforma hossszú lista, az utolso egy változó/egész. Jelentése: a `Starts` kezdőidőpontokban elkezdett, `Durations` ideig tartó és `Resources` erőforrásigényű feladatok bármely időpontban összesített erőforrásigénye nem haladja meg a `Limit` határt. Jelöljük: a `=` `min(`$S1,\ldots,Sn$`)`, b `=` `max(`$S1$`+`$D1,\ldots,Sn$`+`Dn`)`, Rij `=` Rj, ha Sj `=<` i `<` Sj`+`Dj, egyébként Rij `=` `0`; ekkor a constraint jelentése: $Ri1$`+...+`Rin `=<` $Limit$, minden a `=<` i `<` b esetén.

Példa: egy egyszerű ütemezési feladat

A feladat meghatározása:

- a rendelkezésre állo erőforrások száma: 13
- az egyes tevékenységek időtartama és erőforrásigénye:

Tevékenység	t1	t2	t3	t4	t5	t6	t7
Időtartam	16	6	13	7	5	18	4
Erőforrásigény	2	9	3	7	10	1	11

A program szövege:

```
:- use_module(library(clpfd)).
:- use_module(library('clpfd/cumul')).

schedule(Ss, End) :-
        length(Ss, 7),
        Ds = [16, 6,13, 7, 5,18, 4],
        Rs = [ 2, 9, 3, 7,10, 1,11],
        domain(Ss, 1, 30),
        domain([End], 1, 50),
        after(Ss, Ds, End),
        cumulative(Ss, Ds, Rs, 13),
        labeling([ff,minimize(End)], [End|Ss]).

after([], [], _).
after([S|Ss], [D|Ds], E) :- E #>= S+D, after(Ss, Ds, E).

%% End of file

| ?- schedule(Ss, End).

Ss = [1,17,10,10,5,5,1],
End = 23 ?
```

Felhasználói constraintek

Mit kell meghatározni egy új constraint definiálásakor?

- milyen tár-szűkítést végezzen el?
- mikor ébredjen fel újabb szűkítés megkisérlése érdekében?
- mikor fejezheti be a működését (mikor válik levezethetővé)?
- ha reifikálni is akarjuk:
  - hogyan kell végrehajtani a negáltját?
  - hogyan döntsük el a tárból való levezethetőségét?
  - hogyan döntsük el a negáltjának a levezethetőségét?

Példa: `'x+y=t'(X, Y, T)`, intervallum konzisztenciával

- `T` tartományából hagyja ki a `min(X)+min(Y)`-nél kisebb és a `max(X)+max(Y)`-nál nagyobb elemeket; ezt ismételje, valahányszor `X` vagy `Y` alsó vagy felső határa változik, mindaddig, míg `X` és `Y` értéket nem kap.
- `X`-ből hagyja ki a `min(T)-max(Y)`-nél kisebb és a `max(T)-min(Y)`-nál nagyobb elemeket; ezt ismételje ha `T` vagy `Y` határa változik.
- hasonlóan szűkítse `Y`-t is.

Constraint definiálási lehetőségek SICStusban

- globális constraintek: Prologban kell megadni a fentieket (reifikálás nincs) (A SICStus 3#5 változatban nem publikus).
- FD predikátumok: ún indexikálisok segítségével deklaratív módon lehet constrainteket definiálni, reifikáltakat is.

`'x+y=t'/3` definiálása FD predikátummal

```
'x+y=t'(X,Y,T) +:
        X in min(T) - max(Y)..max(T) - min(Y),
        Y in min(T) - max(X)..max(T) - min(X),
        T in min(X) + min(Y)..max(X) + max(Y) .
```

FD predikátumok

Egy FD predikátum 1 vagy 4 klózból áll

- A `+:` nyakjelű kötelező, a `-:`, `+?`, `-?` nyakjelűek csak reifikálandó constraintek esetén szükségesek.
- A `+:` `-:` ún szűkítő (mondó, *tell*) indexikálisok az adott constraint és negáltja tárra gyakorolt hatását írják le.
- A `+?` `-?` ún kérdező (*ask*) indexikálisok az adott constraint és negáltja levezethetőségi feltételeit definiálják.
- Az FD predikátum fejében az argumentumok különböző változók; törzsében csak ezeket a változókat tartalmazó indexikálisok lehetnek.

Indexikálisok

- Alakjuk: *Változó* `in` *Tartománykifejezés*
- *Tartománykifejezés* (eredetileg *Range*) egy halmazértékű függvény, amely a benne szereplő változóktól függoen egy tartományt generál.
- Baloldalán egy a fejben felsorolt változó áll, jobboldalán az **összes** többi változót tartalmazó tartománykifejezés.
- **Alapszabály**: Ha `X in R` szerepel egy `C` constraint definiciójában, akkor ha az `R`-beli változóknak konkrét értéket adunk, akkor az `R` tartománykifejezés értéke **pontosan** a `C` constraintet kielégítő `X` értékek halmaza kell legyen.

Példa: `'x+y=t'/3` tartomány-konzisztenciával

```
'x+y=t'(X,Y,T) +:
      X in dom(T) - dom(Y),        % {t − y|t ∈ dom(T), y ∈ dom(Y)}
      Y in dom(T) - dom(X),
      T in dom(X) + dom(Y).
```

Indexikálisok szintaxisa

```
Indexical --> X in Range

X --> variable                          { constraint-valtozó }

Constant --> integer
  |   inf   |   sup                     { minusz/plusz végtelen }
  |   X                                 { csak behelyettesítve
                                          értékelhetö ki }
Term --> Constant
  |   card(X)                           { X tartományának mérete }
  |   min(X)                            { X alsó határa }
  |   max(X)                            { X felsö határa }
  |   Term + Term | Term - Term
  |   Term * Term | Term mod Term
  |   - Term
  |   Term /> Term                      { felfelé kerekített osztás }
  |   Term /< Term                      { lefelé kerekített osztás }

ConstantSet -->  {Term,...,Term}

Range --> ConstantSet
  |   dom(X)                                 { X tartománya }
  |   Term..Term                            { intervallum }
  |   Range/\Range                          { metszet }
  |   Range\/Range                          { únió }
  |   \Range                                { komplementer halmaz }
  |   - Range                               { pontonkénti negáció }
  |   Range + Range | Range + Term          { pontonkénti össszeadás}
  |   Range - Range | Range - Term
  |   Term - Range                          { pontonkénti kivonás }
  |   Range mod Range | Range mod Term      { pontonkénti modulo }
  |   Range ? Range                         { feltételes kifejezés }
  |   unionof(X,Range,Range)                { únió-kifejezés }
  |   switch(Term,MapList)                  { kapcsoló-kifejezés }
```

Szűkítő indexikálisok kezelése

Tartománykifejezések monotonitása

- Jelöljük $D(X, S)$-sel az X változó tartományát az S tárban, $S(R)$-rel egy R tartománykifejezés értékét az S tárban.
- Egy S tár kiterjesztése S', ha minden X változóra $D(X, S') \subseteq D(X, S)$.
- Egy R tartománykifejezés egy S tárra nézve monoton, ha S minden S' kiterjesztésére $S'(R) \subseteq S(R)$.
- R S-re nézve antimonoton, ha S minden S' kiterjesztésére $S'(R) \supseteq S(R)$.
- R S-ben konstans, ha monoton és antimonoton (nem változik).
- egy indexikális (anti)monoton, ha a benne levő tartománykifejezés (anti)monoton.
- **Tétel**: ha egy `X in R` indexikális monoton egy S tárban, akkor az `X` értéktartománya S(`R`)-rel szűkíthető (vö. alapszabály).

Szűkítő indexikálisok feldolgozása

- Az `X in R` indexikálist felfüggesztjük amig kiértékelhető és monoton nem lesz (a megfelelő változók be nem helyettesítődnek).
- Az `R`-beli többi változóra megállapítjuk az ébresztési feltételeket
  - `dom(Y), card(Y)` környezetben — bármilyen változáskor
  - `min(Y)` környezetben – alsó határ változásakor
  - `max(Y)` környezetben– felső határ változásakor
- Ha az indexikális felébred:
  - Ha $D(X, S)$ és $S(R)$ diszjunktak, akkor visszalépünk.
  - a tárat az X `in` $S(R)$ constrainttel **szűkitjük** (erősítjük), azaz $D(X, S) := D(X, S) \wedge S(R)$
  - Ha $S(R)$ konstans, az **indexikálist tartalmazó constraint** levezethetővé vált.
  - Egyébként újra elaltatjuk az indexikálist.

Példák szűkítő indexikálisokra

```
't+u=c'(T,U,C) +:                    % Tartomány-konzisztens!
        T in C - dom(U),        U in C - dom(T).

't+u>=c'(T,U,C) +:
        T in C - max(U)..sup,  U in C - max(T)..sup.

'ax+c=t'(A,X,C,Y) +:
        X in (min(Y) - C) /> A .. (max(Y) - C) /< A,
        Y in  min(X)*A + C     .. max(X)*A + C.

'p\\/q'(P, Q, B) +:
        B in (min(P)+min(Q))/>2 .. (max(P)+max(Q))/>2,
        P in min(B)*2 - max(Q) - 1  .. max(B)*2 - min(Q),
        Q in min(B)*2 - max(P) - 1  .. max(B)*2 - min(P).
% | ?- domain([P,Q,B], 0, 1), 'p\\/q'(P, Q, B), P=1.
%                            B = 1, P = 1, Q in 0..1 ?

'abs(x-y)>c'(X, Y, C) +:
        X in (inf .. max(Y)-C-1) \/ (min(Y)+C+1 .. sup),
        Y in (inf .. max(X)-C-1) \/ (min(X)+C+1 .. sup).
% | ?- 'abs(x-y)>c'(X,Y,3), X in 3..5.
%                            X in 3..5, Y in(inf..1)\/(7..sup) ?

no_threat_3(X, Y, I) +:
        X in \({Y} \/ {Y+I} \/ {Y-I}),
        Y in \({X} \/ {X+I} \/ {X-I}).

% Hibás constraint, sérti az alapszabályt!
'x=<y=<z'(X, Y, Z) +:
        Y in min(X)..max(Z),
        Z in min(Y).. sup,
        X in inf..max(Y).
% | ?- 'x=<y=<z'(15, 5, Z).
%                            Z in 5..sup ?
```

Bonyolultabb tartománykifejezések

Únió-kifejezés

`unionof(X, H, T)` — ahol `X` változó, `H` és `T` tartománykifejezések. Kiértékelése egy S tárban: kiértékeljük `H`-t, legyen ennek értéke $\{x_1, \ldots, x_n\}$. Képezzük a T_i kifejezéseket úgy, hogy `T`-ben `X` helyébe x_i-t írjuk. Ekkor az únió-kifejezés értéke a $S(T_1), \ldots, S(T_n)$ halmazok uniója. Ha $S(H)$ végtelen, felfüggesztjük. Példa:

```
% Maximálisan szükítö, de hatékonytalan!
no_threat_4(X, Y, I) +:
        X in unionof(B,dom(Y),\({B} \/ {B+I} \/ {B-I})),
        Y in unionof(B,dom(X),\({B} \/ {B+I} \/ {B-I})).
```

Feltételes kifejezés

`Felt ? Tart` – ahol `Felt` és `Tart` tartománykifejezések. Ha S(`Felt`) üres halmaz, akkor a feltételes kifejezés értéke is üres halmaz, egyébként pedig azonos S(`Tart`) értékével. Példa:

```
'x=<y=<z'(X, Y, Z) +:       % Ez már helyes!
        Y in min(X)..max(Z),
        Z in ((inf..max(Y)) /\ dom(X)) ? (min(Y).. sup),
        X in ((min(Y)..sup) /\ dom(Z)) ? (inf..max(Y)).
```

Használható a kiértékelés késleltetésére: `( Felt?(inf..sup) \/ Tart )` eredménye S(`Tart`), ha S(`Felt`) üres, egyébként `inf..sup`. Az ilyen szerkezetekben `Tart` értékét ki sem számoljuk, amíg `Felt` nem üres. Példa:

```
% Maximálisan szükít, kicsit kevésbé hatékonytalan
no_threat_5(X, Y, I) +:
    X in (4..card(Y)) ? (inf..sup) \/
          unionof(B,dom(Y),\({B} \/ {B+I} \/ {B-I})),
    Y in (4..card(X)) ? (inf..sup) \/
          unionof(B,dom(X),\({B} \/ {B+I} \/ {B-I})).
```

Kapcsoló-kifejezés

`switch(T, MapList)` — ahol `T` a `term` szintaxisának megfelelő kifejezés, `MapList` pedig `integer-Range` alakú párokból álló lista. Ha $S(\texttt{T})$ értéke Key és `MapList` tartalmaz egy $Key - Expr$ párt, akkor a kapcsoló értéke $S(Expr)$, egyébként üres. Például az alábbi két definició hatása azonos:

```
p(X, Y) :- relation(X, [1-{1},2-{1,2},3-{1,2,3}], Y).

q(X, Y) +:
  X in unionof(B,dom(Y),switch(B,[1-{1,2,3},2-{2,3},3-{3}])),
  Y in unionof(B,dom(X),switch(B,[1-{1},2-{1,2},3-{1,2,3}])).
```

Reifikálható constraintek definiálása

Négy klózból állo FD-predikátum szükséges, példa:

```
'x\\=y'(X,Y) +:        % 1. a constraintet szükitö indexikálisok
        X in \{Y},
        Y in \{X}.
'x\\=y'(X,Y) -:        % 2. a negáltját szükitö indexikálisok
        X in dom(Y),
        Y in dom(X).
'x\\=y'(X,Y) +?        % 3. a levezethetöséget kérdezö
        X in \dom(Y).  %                              indexikális
'x\\=y'(X,Y) -?        % 4. a negált levezethetöségét kérdezö
        X in {Y}.      %                              indexikális
```

A kérdező klózok csak egyetlen indexikálist tartalmazhatnak.

`'x\\=y'(X,Y) #<=> B` végrehajtásakor

- A 3. klóz figyeli, hogy `X` és `Y` tartománya diszjunktttá vált-e, ha igen, `B=1`;
- A 4. klóz figyeli, hogy `X=Y` igaz-e, ha igen `B=0`;
- egy külön démon figyeli, hogy `B` behelyettesítődött-e, ha igen, és `B=1`, meghivja az 1. klózt, ha `B=0`, akkor a 2. klózt.

Kérdező indexikálisok feldolgozása

- Az `X in R` indexikálist felfüggesztjük amig kiértékelhető és antimonoton nem lesz (a megfelelő változók be nem helyettesítődnek).
- Az ébresztési feltételek (`Y` az `R`-ben előforduló változó):
  - `X` tartományának bármilyen változáskor
  - `dom(Y), card(Y)` környezetben — bármilyen változáskor
  - `min(Y)` környezetben – alsó határ változásakor
  - `max(Y)` környezetben – felső határ változásakor
- Ha az indexikális felébred:
  - Ha $D(X,S) \subseteq S(R)$ akkor a constraint levezethetővé vált.
  - Egyébként, ha $D(X,S)$ és $S(R)$ diszjunktak, valamint $S(R)$ konstans, akkor a constraint negáltja levezethetővé vált.
  - Egyébként újra elaltatjuk az indexikálist.

Példa

```
'x=<y'(X,Y) +:
        X in inf..max(Y),
        Y in min(X)..sup.
'x=<y'(X,Y) -:
        X in (min(Y)+1)..sup,
        Y in inf..(max(X)-1).
'x=<y'(X,Y) +?
        X in inf..min(Y).
'x=<y'(X,Y) -?
        X in (max(Y)+1)..sup.
```

FD-predikátumok, indexikálisok összefoglalása

- Legyen `C(Y1, ...)` egy FD-predikátum, amelyben szerepel egy

  `Yi in R(Y1, ...)`

  indexikális. A baloldali `Yi` változót néha `X`-nek nevezzük. Az `R` tartománykifejezéshez tartozó halmazfüggvényt nevezzük R-nek.

- (definició:) Az indexikális az alábbi relációt definiálja:

  C `=` $\{ < y_1, \ldots > \mid y_k \in$ `inf..sup` $\wedge y_i \in R(\{y_1\}, \ldots)\}$

- (alapszabály:) Egy FD-predikátum csak akkor értelmes, ha a pozitív klózaiban levő összes indexikális ugyanazt a relációt definiálja; továbbá a negatív klózaiban levő összes indexikális ennek a relációnak a negáltját (komplemensét) definiálja.

- Ha R monoton egy S tárra nézve, akkor $S(R)$-ről belátható, hogy minden olyan x értéket tartalmaz, amelyek (az S által megengedett y_j értékekkel együtt) a C relációt kielégítik. Ezért szűkítő indexikálisok esetén jogos az X tartományát $S(R)$-rel szűkíteni.

- Ha R antimonoton egy S tárra nézve, akkor $S(R)$-ről belátható, hogy minden olyan x értéket kizár, amelyekre (az S által megengedett y_j értékekkel együtt) a C reláció nem áll fenn. Ezért kérdező indexikálisok esetén, ha $D(X,S) \subseteq S(R)$, jogos a constraintet az S tárbol1 levezethetőnek tekinteni.

- A fentiek miatt természetesen adódik az indexikálisok felfüggesztési szabálya: a szűkítő indexikálisok végrehajtását mindaddig felfüggesztjük, amig monotonná nem válnak; a kérdező indexikálisok végrehajtását mindaddig felfüggesztjük, amig antimonotonná nem válnak.

- **Az indexikálisok deklaratív volta:** Ha a fenti alapszabályt betartjuk, akkor a `clpfd` megvalósítás az FD-predikátumot helyesen valósítja meg, azaz mire a változók teljesen behelyettesítetté válnak az FD-predikátum akkor és csak akkor for sikeresen lefutni, vagy 1-gyé tükroződni (reifikálódni), ha a változók értékei a predikátum által definiált relációhoz tartoznak. Az indexikális megfogalmazásán csak az múlik, hogy a nem-konstans tárak esetén milyen jó lesz a szűkítő ill. kérdező viselkedése.

Constraintek automatikus fordítása indexikálisokká

Indexikálissá fordítandó constraint

- Formája: *Head* `+:` *Constraint*.
- Ebben *Constraint* lehet
  - aritmetikai (csak lineáris kifejezésekre);
  - `relation/3` és `element/3` szimbólikus constraint.
- Csak a `+:` nyakjel használható, ezek a constraintek nem reifikálhatóak.

Mi a különbség?

- `p(X,Y,U,V) :- X+Y #< U+V)`. törzse `clpfd` könyvtári hívásokra fordul: `clpfd:('x+y=t'(X,Y,_A),'x+y=t'(U,V,_B),'t>=u+c'(_B,_A,1))`
- `p(X,Y,U,V) +: X+Y #< U+V)`. egyetlen FD predikátummá válik: `p(X,Y,U,V) +: X in` *R*`(Y,U,V), Y in ..., U in ..., V in ....`
- Az első változat kevesebb helyet foglal el, a második gyorsabb.
- Vigyázat: az aritmetikai kifejezésekből fordított indexikális mérete négyzetesen nő a forráskód méretével.
- A `relation/3` és `element/3` szimbólikus constraintek unió- és kapcsoló-kifejezésekké fordulnak (lineáris méret), vigyázat a nagyméretű kezdő-tartományokkal.
- Példa: `sum2(S,E,N,D,M,O,R,Y) +: 1000*S+100*E+10*N+D + 1000*M+100*O+10*R+E #= 10000*M+1000*O+100*N+10*E+Y`. kb. 7-szer nagyobb helyigényű, 1.5-szer gyorsabb, mint a `:-`-os változat.
- A #6-os SICStus változatban egy gyors `scalar_product` eljárás van, ezzel viszont a fenti példa 2-szer lassabb `:-(`!

Újdonságok a következő SICStus-ban (#6)

Új és bővült eljárások

- `scalar_product(Coeffs, Xs, Relop, Value)`: Igaz, ha `Coeffs * Xs Relop Value`, ahol `Relop` aritmetikai összehasonlító operátor, `*` skalárszorzást jelöl, `Coeffs` egészekből álló lista.
- `sum(Xs, Relop, Value)`: Σ `Xs Relop Value`.
- `count(Val, List, Relop, Count)`: Igaz, ha a `Val` egész szám `Lista`-ban való előfordulásainak száma n, és n `Relop Count`.
- `element(N, List, X)`: `List`-ben lehetnek constraint-változók.
- `permutation(Xs, Ys, Ctrl)`: `Xs` és `Ys` n hosszúságú listák. Igaz ha `X[i]` és `Y[i]` mind az `1`..n tartományba esnek, és `X[i]=j` $\Leftrightarrow$ `Y[j]=i`.
- `circuit(Xs)`: `Xs` n hosszúságú lista. Igaz ha minden `X[i]` az `1`..n tartományba esik és `X[1], X[X[1]], ...` az `1`..n egy permutációja.
- `cumulative(Starts, Durations, Resources, Limit)`: jobb a szűkítés
- `serialized(Starts, Durations)`: a `cumulative` speciális esete, ahol az összes erőforrás-igény és a korlát is 1.

Globális constraintek

- `fd_global(Constraint, State, Susp)`: `Constraint` végrehajtásának elindítása, `State` kezdőállapottal, `Susp` ébresztési listával (elemei: `dom(X)`, `min(X)`, `max(X)`, `minmax(X)`, `val(X)`).
- `clpfd:dispatch_global(Constraint, State0, State, Actions)`: A kampó-eljárás törzse definiálja a `Constraint` felébredésekor elvégzendőket: `State0` a régi, `State` az új állapot. Az `Actions` lista elemei lehetnek: `exit`, `fail`, `X=V`, `X in R`, `X in_set S`, `call(Module:Goal)`.

FD-halmazok

- `fd_set(X, Set)`: `X` tartománya a `Set` FD-halmaz (néhai `fd_dom`).
- `X in_set Set`: `X`-et `Set`-re szűkíti.
- Az FD-halmaz: diszjunkt intervallumok rendezett listája. Mint absztrakt adattípus, alapműveletei:
  - `empty_fdset(S)`: `S` az üres FD-halmaz.
  - `fdset_parts(S, Min, Max, Rest)`: `S` első intervalluma `Min..Max`, a maradék FD-halmaz: `Rest`. FD-halmaz építésére is jó.
  - `is_fdset(S)`: `S` egy korrekt FD-halmaz.
- További, kényelmi szolgáltatások, pl.
  - `fdset_singleton(Set, Elt)`: `Set` az egyetlen `Elt`-ből áll.
  - `fdset_union/3`, `fdset_intersection/3`, `fdset_complement/2`.
  - `fdset_member(Elt, Set)`: `Elt` eleme a `Set` FD-halmaznak.

Példa: `exactly/3` (korábbi `pontosan/3`)

```
% An implementation of exactly(I, [X1,...Xm], N). Rewrite rules:

% [1] |= Xi=I  ==> exactly(I, [X1,...Xi-1,Xi+1,...Xm], N-1)
% [2] |= Xi\=I ==> exactly(I, [X1,...Xi-1,Xi+1,...Xm], N)
% [3] |= N=0   ==> X1\=I, ... Xm\=I
% [4] |= N=m   ==> X1=I,  ... Xm=I
%                          Necessary condition: 0 =< N =< m.

% the entrypoint
exactly(I, Xs, N) :-
    dom_susps(Xs, Susp),
    fd_global(exactly(I,Xs,N), state(Xs,N), Susp).

dom_susps([], []).
dom_susps([X|Xs], [dom(X)|Susp]) :- dom_susps(Xs, Susp).
```

```
% the solver method
:- multifile clpfd:dispatch_global/4.
clpfd:dispatch_global(exactly(I,_,_),
                state(Xs0,N0), state(Xs,N), Actions) :-
    ex_filter(Xs0, Xs, N0, N, I), length(Xs, M),
    (   N=:=0 -> Actions = [exit|Ps], ex_neq(Xs, I, Ps)
    ;   N=:=M -> Actions = [exit|Ps], ex_eq(Xs, I, Ps)
    ;   N>0, N<M -> Actions = []
    ;   Actions = [fail]
    ).

% rules [1,2]: filter the X's, decrementing N
ex_filter([], [], N, N, _).
ex_filter([X|Xs], Ys, L, N, I) :-
    X==I, !, M is L-1, ex_filter(Xs, Ys, M, N, I).
ex_filter([X|Xs], Ys0, L, N, I) :-
    fd_set(X, Set), fdset_member(I, Set), !,
    Ys0 = [X|Ys], ex_filter(Xs, Ys, L, N, I).
ex_filter([_|Xs], Ys, L, N, I) :- ex_filter(Xs, Ys, L, N, I).

% rule [3]: all must be neq I
ex_neq(Xs, I, Ps) :- fdset_singleton(Set0, I),
    fdset_complement(Set0, Set), eq_all(Xs, Set, Ps).

% rule [4]: all must be eq I
ex_eq(Xs, I, Ps) :-
    fdset_singleton(Set, I), eq_all(Xs, Set, Ps).

eq_all([], _, []).
eq_all([X|Xs], Set, [X in_set Set|Ps]) :- eq_all(Xs, Set, Ps).

% sample queries:
| ?- exactly(5,[A,B,C],1), A=5.
      A = 5, B in(inf..4)\/(6..sup), C in(inf..4)\/(6..sup)
| ?- exactly(5,[A,B,C],1), A in 1..2, B in 3..4.
      C = 5, A in 1..2, B in 3..4
```

Házifeladat

Egy constraint feladat megoldása

- A megoldás lehet `clpfd`-ben, `clpq`-ban, Prolog-ban, Mercury-ban, stb.
- Előny, ha többféle megoldás is van, és ezeket össze is hasonlítja a dolgozat, lásd pl. a térképszinezés, N-királynő megoldásvariánsait korábban.
- `clpfd` megoldás esetén érdekes lehet a különféle constraint-megfogalmazások, `labeling` technikák összehasonlítása.

Javasolt feladat: japán feladvány (starpic)

- Adott egy NxM-es négyzethálós üres (fehér) téglalap.
- A téglalap bizonyos mezőit be kell feketíteni, hogy egy rejtett ábra tünjék elő.
- A téglalap minden sorához és oszlopához adott egy számsor: az adott sorban/oszlopban levő összefüggő fekete szakaszok hosszai.
- Egy egyszerű példa és megoldása:

```
      1 1 2 1 1                  1 1 2 1 1
      1   1 1                    1   1 1

2     . . . . .          2       . . x x .
1 1   . . . . .          1 1     x . x . .
1 1   . . . . .          1 1     . x . . x
1 2   . . . . .          1 2     x . x x .
```

- A példa egy lehetséges meghívási módja:

```
| ?- starpic([[2],[1,1],[1,1],[1,2]],
             [[1,1],[1],[2,1],[1,1],[1]]).
[0,0,1,1,0]
[1,0,1,0,0]
[0,1,0,0,1]
[1,0,1,1,0]
```

Példa: Négyzetdarabolás

A feladat

- Adott egy nagy négyzet oldalhosszúsága, pl: `10`.
- Adottak kis négyzetek oldalhosszúságai, pl `[6,4,4,4,2,2,2,2]` (területösszegük megegyezik a nagy négyzet területével).
- A kis négyzetekkel pontosan le kell fedni a nagyot (meghatározandók a kis négyzetek koordinátái), pl:
  ```
  Xs = [1,7,7,1,5,5,7,9]
  Ys = [1,1,5,7,7,9,9,9]
  Ss = [6,4,4,4,2,2,2,2]
  ```

Próba-adatok

- `10, [6,4,4,4,2,2,2,2],`
- `20, [9,8,8,7,5,4,4,4,4,4,3,3,3,2,2,1,1],`
- `112,`
  `[50,42,37,35,33,29,27,25,24,19,18,17,16,15,11,9,8,7,6,4,2],`
- `175,`
  `[81,64,56,55,51,43,39,38,35,33,31,30,29,20,18,16,14,9,8,5,4,3,2,1]`.

Négyzetdarabolás: Prolog megoldás

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

squares_1(Ss, Size, SXYs) :-
        MSize is -Size,
        squares(Ss, [MSize/Size,Size/0], Size, SXYs).

squares([], _, _, [])  :- !.
squares(Ss0, L0, Size, SXYs) :-
        L0=[V0/_|_], Y0 is Size+1+V0,
        hole(L0, 1-Y0, XY, L1, L1T, L2),
        fill_hole(L2, L1T, XY, Size, Ss0, Ss, SXYs1, SXYs),
        squares(Ss, L1, Size, SXYs1).

fill_hole([V0/0,V/H|L], [V1/H|L], _XY, _Size,
                        Ss0, Ss0, SXYs0, SXYs0) :-
        !, V1 is  V0+V.
fill_hole([V/H|L0], [V1/S|L1], X-Y, Size,  Ss0, Ss,
                        SXYs0, [s(S,X,Y)|SXYs]) :-
        MaxS is min(H, Size+1-Y),
        select(S, Ss0, Ss1), S =< MaxS,
        V1 is V+S, MS is -S, H1 is H-S, X1 is X+S,
        fill_hole([MS/H1|L0], L1, X1-Y, Size, Ss1, Ss, SXYs0, SXYs).

hole([V1/0,V2/H|L], XY0, XY, L1, L1T, L2) :-
        !, V is V1+V2,
        hole([V/H|L], XY0, XY, L1, L1T, L2).
hole([V/H1,0/H2|L], XY0, XY, L1, L1T, L2) :-
        !, H is H1+H2,
        hole([V/H|L], XY0, XY, L1, L1T, L2).
hole([V1/H1|L], XY, XY, L1, L1, [V1/H1|L]) :-
        H1 > 0, V1 < 0, L = [V2/_|_], V2>0, !.
hole([VH1,VH2|L], X0-Y0, XY, [VH1|L1], L1T, L2) :-
        VH1=_/H1, VH2=V2/_, X1 is X0+H1, Y1 is Y0+V2,
        hole([VH2|L], X1-Y1, XY, L1, L1T, L2).
```

Négyzetdarabolás: naív `clpfd` megoldás

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

squares_2(Sizes, Size, Xs, Ys) :-
        generate_coordinates(Xs, Ys, Sizes, Size),
        state_no_overlap(Xs, Ys, Sizes),
        labeling([], Xs), labeling([], Ys).

generate_coordinates([], [], [], _).
generate_coordinates([X|Xs], [Y|Ys], [S|Ss], Size) :-
        Sd is Size-S+1,
        domain([X,Y], 1, Sd),
        generate_coordinates(Xs, Ys, Ss, Size).

state_no_overlap([], [], []).
state_no_overlap([X|Xs], [Y|Ys], [S|Ss]) :-
        state_no_overlap(X, Y, S, Xs, Ys, Ss),
        state_no_overlap(Xs, Ys, Ss).

state_no_overlap(_, _, _, [], [], []).
state_no_overlap(X, Y, S, [X1|Xs], [Y1|Ys], [S1|Ss]) :-
        no_overlap(X, Y, S, X1, Y1, S1),
        state_no_overlap(X, Y, S, Xs, Ys, Ss).

no_overlap(X1, _Y1, S1, X2, _Y2, _S2) :-
        leqc(X1, S1, X2).
no_overlap(X1, _Y1, _S1, X2, _Y2, S2) :-
        leqc(X2, S2, X1).
no_overlap(_X1, Y1, S1, _X2, Y2, _S2) :-
        leqc(Y1, S1, Y2).
no_overlap(_X1, Y1, _S1, _X2, Y2, S2) :-
        leqc(Y2, S2, Y1).

leqc(X1, S1, X2) +: X1+S1 #=< X2.
```

Négyzetdarabolás: diszjunktív constraintek

A diszjunkció lehetséges megvalósításai

- spekulatív: Prolog választási pontokkal (lásd `squares_2`).
- reifikáció(számosság)-alapú: a `#\/` müvelettel, pl:
  `X1+5 #=< X2 #\/ X2+5 #=< X1.`
- (gyenge) konstruktív diszjunkció:
  `domain([X1,X2], 0, 6), X1+5 #=< X2 \/ X2+5 #=< X1`
  `X1 in (0..1)\/(5..6), X2 in (0..1)\/(5..6)`
- teljes konstruktív diszjunkció:
  `domain([X,Y], 0, 20), X+Y #= 20, X #=<5 \/ Y #=< 5.`
  `X in (0..5)\/(15..20), Y in (0..5)\/(15..20)`

Számosság-alapú `no_overlap`

```
no_overlap_3(X1, Y1, S1, X2, Y2, S2) :-
        X1+S1 #=< X2 #<=> B1,
        X2+S2 #=< X1 #<=> B2,
        Y1+S1 #=< Y2 #<=> B3,
        Y2+S2 #=< Y1 #<=> B4,
        B1+B2+B3+B4 #>= 1.
```

Gyenge konstruktív diszjunkción alapuló `no_overlap`

```
no_overlap_4(wcd1, X1, Y1, S1, X2, Y2, S2) :-
  call( abs(2*(X1-X2)+(S1-S2)) #>= S1+S2 #\/
        abs(2*(Y1-Y2)+(S1-S2)) #>= S1+S2 ).
```

```
no_overlap_5(X1, Y1, S1, X2, Y2, S2) +:
  X1 in ((((min(Y1)+S1)..max(Y2))\/((min(Y2)+S2)..max(Y1))) ? (inf..sup))
        \/ ((inf..max(X2)-S1) \/ (min(X2)+S2..sup)),
  X2 in ((((min(Y1)+S1)..max(Y2))\/((min(Y2)+S2)..max(Y1))) ? (inf..sup))
        \/ ((inf..max(X1)-S2) \/ (min(X1)+S1..sup)),
  Y1 in ((((min(X1)+S1)..max(X2))\/((min(X2)+S2)..max(X1))) ? (inf..sup))
        \/ ((inf..max(Y2)-S1) \/ (min(Y2)+S2..sup)),
  Y2 in ((((min(X1)+S1)..max(X2))\/((min(X2)+S2)..max(X1))) ? (inf..sup))
        \/ ((inf..max(Y1)-S2) \/ (min(Y1)+S1..sup)).
```

Négyzetdarabolás: redundáns constraintek

Kapacitás-constraintek

- minden sorban és minden oszlopban vizsgáljuk meg a metszett négyzeteket,
- kössük ki, hogy ezek oldalhossz-összege a nagy négyzet oldalával megegyezik.

A program

```
squares_6(Sizes, Size, Xs, Ys) :-
        generate_coordinates(Xs, Ys, Sizes, Size),
        state_no_overlap(Xs, Ys, Sizes),
        state_capacity(Xs, Sizes, Size),
        state_capacity(Ys, Sizes, Size),
        labeling([], Xs), labeling([], Ys).

state_capacity(Cs, Sizes, Size) :-
        state_capacity(1, Size, Cs, Sizes).

state_capacity(Pos, Size, _, _) :-
        Pos>Size, !, true.
state_capacity(Pos, Size, Cs, Sizes) :-
        accumulate(Cs, Sizes, Pos, Size),
        Pos1 is Pos+1,
        state_capacity(Pos1, Size, Cs, Sizes).

accumulate([], [], _, 0).
accumulate([C|Cs], [S|Ss], Pos, Sum) :-
        Crutch is Pos-S+1,
        C in Crutch .. Pos #<=> B,
        B*S+Sum1 #= Sum,
        accumulate(Cs, Ss, Pos, Sum1).
```

Négyzetdarabolás: speciális cimkézés

A címkézés lényege:

- keressük meg a legkisebb alsó határt,
- sorra próbáljuk a változókat a legkisebb határ-értékkel behelyettesíteni,
- ha egy változóra ez nem sikerül, zárjuk ki az alsó határt a tartományából.

A program

```
squares_7(Sizes, Size, Xs, Ys) :-
        generate_coordinates(Xs, Ys, Sizes, Size),
        state_no_overlap(Xs, Ys, Sizes),
        state_capacity(Xs, Sizes, Size),
        state_capacity(Ys, Sizes, Size),
        dual_labeling(Xs, 1, Size),
        dual_labeling(Ys, 1, Size).

dual_labeling([], _, _) :- !.
dual_labeling(L, I, Limit) :-
        dual_labeling(L, L1, I, Limit, J),
        dual_labeling(L1, J, Limit).

dual_labeling([], [], _, J, J).
dual_labeling([X|L1], L2, I, J0, J) :-
        (   integer(X) -> dual_labeling(L1, L2, I, J0, J)
        ;   X = I, dual_labeling(L1, L2, I, J0, J)
        ;   X #> I,
            fd_min(X, J1), J2 is min(J0,J1), L2 = [X|L3],
            dual_labeling(L1, L3, I, J2, J)
        ).
```

A négyzetdarabolás megoldásainak összehasonlítása

Futási idő (másodperc)

variáns/méret	10	20	112	175
1	0.00	6.29	152.63	
2	5.68			
6+2	5.68			
6+3	0.44	2.97		
6+4	0.56	3.74		
6+5	0.30	1.93		
7+2	5.59			
7+6+2	5.83			
7+6+3	0.44	2.22	69.95	125.79
7+6+4	0.55	2.75	71.08	123.75
7+6+5	0.32	1.50	63.29	115.83

Visszalépések száma

variáns/méret	10	20	112	175
1	0	1678	3398379	
2	46954			
6+2	46954			
6+3	0	18		
6+4	0	18		
6+5	0	18		
7+2	46954			
7+6+2	46954			
7+6+3	0	0	150	105
7+6+4	0	0	150	105
7+6+5	0	0	150	105

A SICStus clp(Q,R) kiterjesztései

A `clpq/clpr` könyvtárak

- Tartomány:
  - `clpr`: lebegőpontos számok
  - `clpq`: racionális számok
- Függvények:

  `+ - * / min max pow exp` (kétargumentumúak),

  `+ - abs sin cos tan` (egyargumentumúak).
- Constraint-relációk:

  `= =:= <, >, =<, >=, \=`
- Primitív constraint-ek (constraint tár elemei):

  lineáris kifejezéseket tartalmazó relációk
- Constraint-megoldó algoritmus:

  lineáris programozási módszerek: Gauss elimináció, szimplex módszer

A könyvtár betöltése:

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

A fő beépített eljárás

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

Egyszerű példák

```
SICStus 3  #3:  Fri Oct 18 19:14:50 MET DST 1996
| ?- use_module(library(clpq)).
{loading .../library/clpq.ql...}
...
{loaded .../library/clpq.ql in module linear, 750 msec 670816 bytes}
| ?- {X=Y+4, Y=Z-1, Z=2*X-9}.          % lineáris egyenlet
X = 6, Y = 2, Z = 3 ?

| ?- {X+Y+9<4*Z, 2*X=Y+2, 2*X+4*Z=36}.% egyenlőtlenség is lehet
{X=1+1/2*Y},  {Z=17/2-1/4*Y},  {Y<48/5} ?

| ?- {(Y+X)*(X+Y)/X = Y*Y/X+100}.      % lineárissá egyszerüsíthető
{X=100-2*Y} ?

| ?- {(Y+X)*(X+Y) = Y*Y+100*X}.        % így már nem...
nonlin:{2*(X*Y)-100*X+X^2=0} ?

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

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

| ?- {A-B+C=10, C=5+5}.                % egyenlőség is levezethető
B = A, C = 10 ?

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

Egy összetettebb példa: hiteltörlesztés

```
| ?- [user].
% 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).
| {user consulted, 20 msec 160 bytes}

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

| ?- 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(Constraint)` — `Constraint` levezethető a jelenlegi tárból.

`inf(Kif, Inf)` ill. `sup(Kif, Sup)` — kiszámolja `Kif` infimumát ill. szuprémumát, és egyesíti `Inf`-fel ill. `Sup`-pal. Példa:

```
| ?- { 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}
```

`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
```

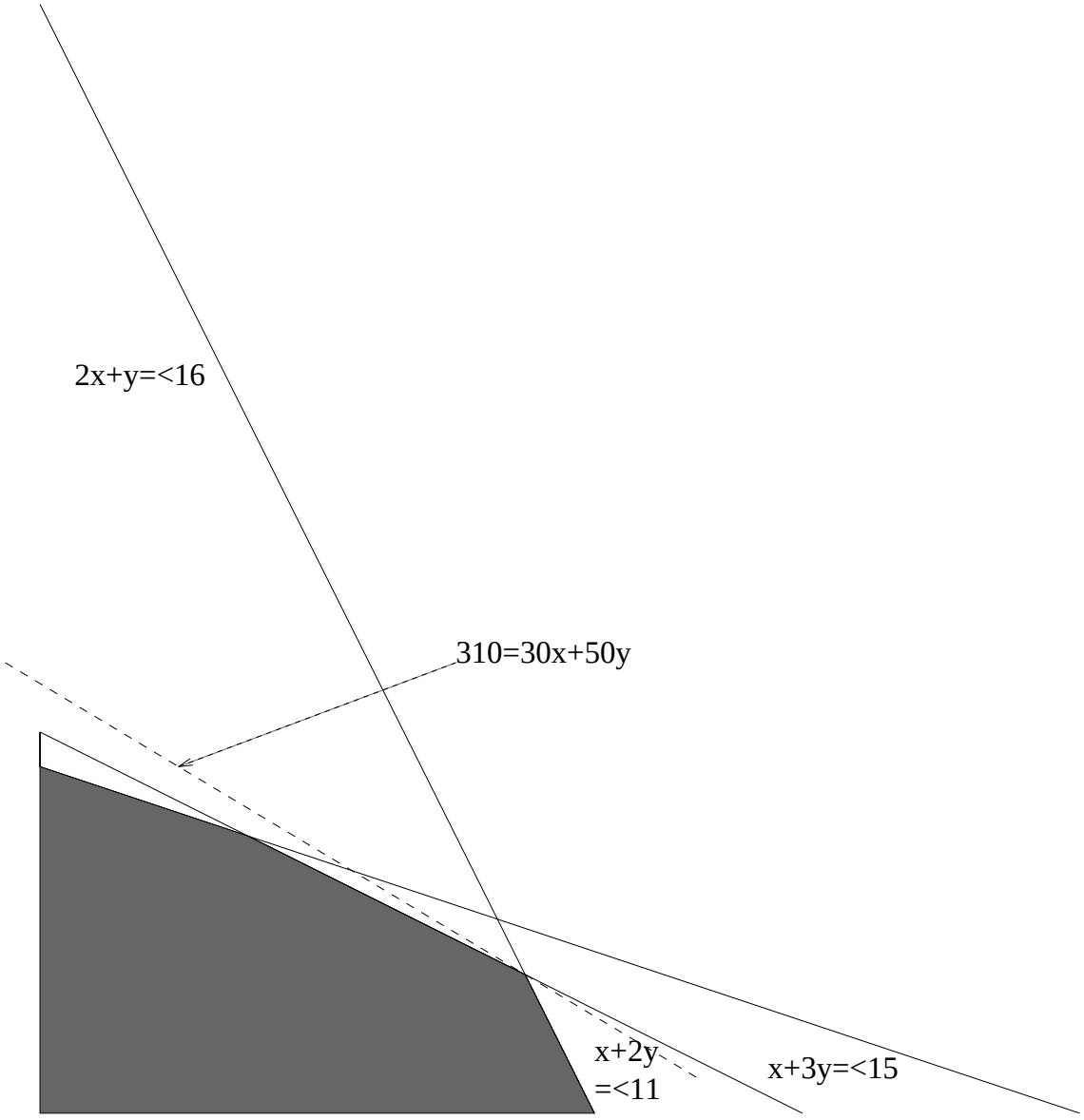
`bb_inf(Egészek, Kif, Inf)` — kiszámolja `Kif` infimumát, azzal a további feltétellel, hogy az Egészek listában levő minden változó egész (ún. "Mixed Integer Optimisation Problem").

```
| ?- {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-constraintben mint a `V2` változó.

`ordering([V1,V2,...])` — `V1`, ... ebben a sorrendben szerepeljen az eredmény-constraintben.

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
% {\'a}ltal kifesz{\'\i}tett h{\'a}romsz{\"o}gben 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} ?
```

Belső ábrázolás

`clpr` — lebegőpontos szám; `clpq` —- `rat(`*Számláló*`,` *Nevezö*`)`, ahol *Számláló* és *Nevezö* 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 ?
| ?- {X=5}, X=5.
no
| ?- {X=5}, X=rat(5,1).
X = 5 ?
```

Példa: tökéletes téglalapok

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

distinct_squares([]).
distinct_squares([S|Ss]) :-
  { S > 0 }, outof(Ss, S), distinct_squares(Ss).

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

% filled_hole(L0, L, Ss0, Ss): Hole in line L0
% filled with squares Ss0-Ss (diff list) gives line 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 on
% HL horizontal line gives (vertical) line L.
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 }.

?- length(Ss, 9), filled_rectangle(Width, Ss).
    Ss = [15/32,9/16,1/4,7/32,1/8,7/16,1/32,5/16,9/32],
    Width = 33/32 ?
```

Nem teljesen tökéletes téglalapok

Az `outof` hívás kihagyásával kapott eredmények

```
W =   1, Ss = [  1]
W =   2, Ss = [  1,  1]
W = 3/2, Ss = [1/2,  1,1/2]
W = 3/2, Ss = [  1,1/2,1/2]
W =   3, Ss = [  1,  1,  1]
W = 4/3, Ss = [1/3,  1,1/3,1/3]
W =   1, Ss = [1/2,1/2,1/2,1/2]
W = 5/3, Ss = [2/3,  1,1/3,1/3]
W = 4/3, Ss = [  1,1/3,1/3,1/3]
W = 5/3, Ss = [  1,2/3,1/3,1/3]
W = 5/3, Ss = [1/3,1/3,  1,2/3]
W = 5/2, Ss = [1/2,  1,  1,1/2]
W = 5/3, Ss = [  1,1/3,1/3,2/3]
W = 5/2, Ss = [  1,1/2,  1,1/2]
W = 5/2, Ss = [  1,  1,1/2,1/2]
W =   4, Ss = [  1,  1,  1,  1]
W = 5/4, Ss = [1/4,  1,1/4,1/4,1/4]
W = 7/5, Ss = [2/5,  1,2/5,1/5,1/5]
W = 7/6, Ss = [1/2,2/3,1/2,1/3,1/3]
W = 6/5, Ss = [3/5,3/5,2/5,2/5,2/5]
W = 7/5, Ss = [2/5,  1,1/5,1/5,2/5]
W = 8/5, Ss = [3/5,  1,1/5,2/5,1/5]
W = 8/5, Ss = [3/5,  1,2/5,1/5,1/5]
W = 7/4, Ss = [3/4,  1,1/4,1/4,1/4]
W = 7/6, Ss = [2/3,1/2,1/2,1/3,1/3]
W = 5/4, Ss = [  1,1/4,1/4,1/4,1/4]
W = 7/5, Ss = [  1,2/5,2/5,1/5,1/5]
W = 7/5, Ss = [  1,2/5,1/5,1/5,2/5]
W = 8/5, Ss = [  1,3/5,1/5,2/5,1/5]
W = 8/5, Ss = [  1,3/5,2/5,1/5,1/5]
...
```

A `clpb` könyvtár

- **Tartomány:** logikai értékek (`1` és `0`, igaz és hamis)
- **Függvények** (egyben constraint-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*).
`X ^ P`	Létezik olyan `X`, hogy `P` igaz (azaz `P[X/0]+P[X/1]` igaz).
`P =\= Q`	Ugyanaz mint `P # Q`.
`P =:= Q`	Ugyanaz mint `~(P # Q)`.
`P =< Q`	Ugyanaz mint `~P + Q`.
`P >= Q`	Ugyanaz mint `P + ~Q`.
`P < Q`	Ugyanaz mint `~P * Q`.
`P > Q`	Ugyanaz mint `P * ~Q`.
`card(Is, Es)`	Az `Es` listában szereplő igaz értékű kifejezések száma eleme az `Is` által jelölt halmaznak (`Is` egészek és `Tol-Ig` szakaszok listája).

- **Primitív constraintek** (constraint tár elemei): tetszőleges constraint (Boole-egyesítők formájában).
- **Constraint-megoldó algoritmus:** Boole-egyesítés.

A `library(clpb)` könyvtár eljárásai

- `sat(`*Kifejezés*`)`, ahol *Kifejezés* változókból, a `0` `1` konstansokból és atomokból (ún. szimbolikus konstansok) a fenti müveletekkel felépített logikai kifejezés. Hozzáveszi *Kifejezés*t a constraint-tárhoz.
- `taut(`*Kif*`,` *Ért*`)`. Megvizsgálja, hogy *Kif* **levezethető-e** a tárból, ekkor *Ért*`=1`; vagy negáltja levezethető-e, ekkor *Ért*`=0`. Egyébként meghiúsul.
- `labeling(`*Változók*`)`. Behelyettesíti a *Változók*at `0` `1` értekekre, úgy, hogy a tár teljesüljön. Visszalépéskor felsorolja az összes lehetséges értéket.

Boole-egyesítés

A feladat:

- Adott `g` és `h` logikai kifejezések.
- Keressük a `g = h` egyenletet megoldó legáltalánosabb egyesítőt (mgu).
- Példa: `mgu(A+B, 1)` lehet `A = B*W + ~B`.
- Egyszerűsítés: A `g = h` egyenlet helyettesíthető az `f = 0` egyenlettel, ahol `f = g # h`.

Az egyesítési algoritmus az f = 0 egyenlőségre

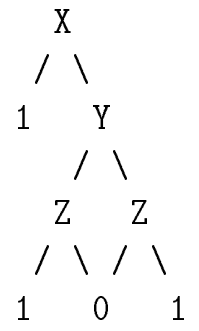
- Ha f-ben nincs változó, akkor azonosnak kell lennie `0`-val (különben nem egyesíthető).
- Egyébként válasszunk ki egy `X` változót. Legyen $f_{\mathtt{X}}$ az f-ből az `X=1`, $f_{-\mathtt{X}}$ az `X=0` behelyettesítéssel kapott kifejezés.
- Helyettesítsünk: `X = W*~`$f_{\mathtt{X}}$ `+ ~W*`$f_{-\mathtt{X}}$ (Boole-egyesítő)
- Folytassuk az egyesítést az $f_{\mathtt{X}}$ `*` $f_{-\mathtt{X}}$ `= 0` egyenlőségre.

Példák

- `mgu(X+Y, 0)` $\longrightarrow$ `X = 0, Y = 0`;
- `mgu(X+Y, 1) = mgu(~(X+Y), 0)` $\longrightarrow$ `X = W + ~W*~Y`;
- `mgu(X*Y, ~(X*Z)) = mgu((X*Y)#(X*Z)#1, 0)` $\longrightarrow$ `X = 1, Y = ~Z`.

Belső ábrázolás: BDD (Boolean/Binary Decision Diagrams)

`X*Y # X*Z # 1` ábrázolása:



Egyszerű példák

```
| ?- use_module(library(clpb)).
{ ... loading ...}

| ?- sat(X + Y).                                  sat(X=\=_A*Y#Y) ?

| ?- sat(x + Y).                                  sat(Y=\=_A*x#x) ?

| ?- taut(_A ^ (X=\=_A*Y#Y) =:= X+Y, T).          T = 1 ?

| ?- sat(A # B =:= 0).                            B = A ?

| ?- sat(A # B =:= C), A = B.                     B = A, C = 0 ?

| ?- taut(A =< C, T).                             no

| ?- sat(A =< B), sat(B =< C), taut(A =< C, T).
                                                  T = 1,
                                                  sat(A=:=_A*_B*C),
                                                  sat(B=:=_B*C) ?

| ?- taut(X, T).                                  no

| ?- taut(x, T).                                  T = 0 ?

| ?- taut(~x, T).                                 T = 0 ?
```

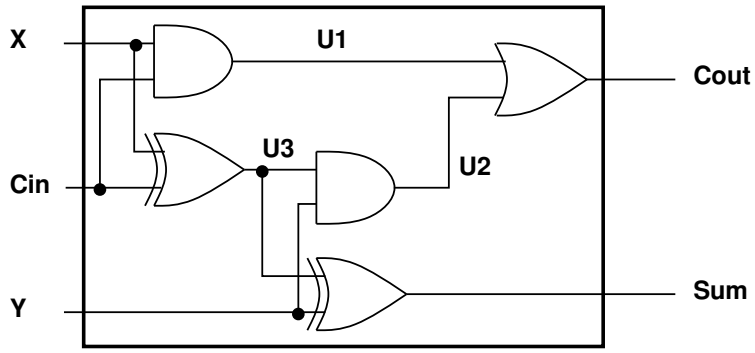
Megjegyzések

- A tár megjelenítése: `sat(V =:= Kif)` ill. `sat(V =\ Kif)` ahol `Kif` egy "polinom", azaz konjunkciókból kizáró vagy (`#`) művelettel képzett kifejezés.
- Az atommal jelölt szimbolikus konstansok nem behelyettesíthetőek, (legkívül) univerzálisan kvantifikált változóknak tekinthetők.

Példa: 1-bites összeadó

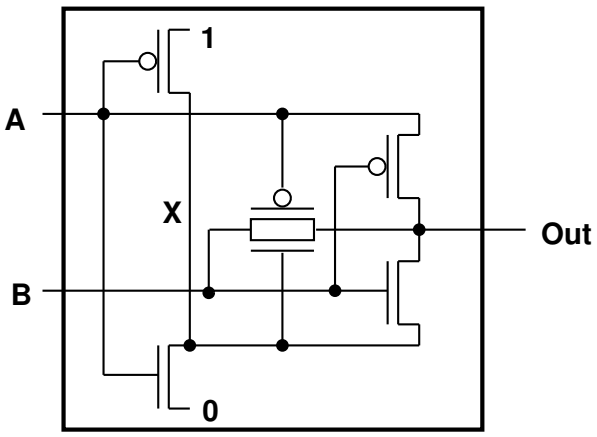
```
| ?- [user].
| adder(X, Y, Sum, Cin, Cout) :-
     sat(Sum =:= card([1,3],[X,Y,Cin])),
     sat(Cout =:= card([2-3],[X,Y,Cin])).
| {user consulted, 40 msec 576 bytes}

yes
| ?- adder(x, y, Sum, cin, Cout).

sat(Sum=:=cin#x#y),
sat(Cout=:=x*cin#x*y#y*cin) ?

yes
| ?- adder(x, y, Sum, 0, Cout).

sat(Sum=:=x#y),
sat(Cout=:=x*y) ?

yes
| ?- adder(X, Y, 0, Cin, 1), labeling([X,Y,Cin]).

Cin = 0, X = 1, Y = 1 ? ;

Cin = 1, X = 0, Y = 1 ? ;

Cin = 1, X = 1, Y = 0 ? ;

no
```

Példa: Hibakeresés áramkörben




```
fault([F1,F2,F3,F4,F5], [X,Y,Cin], [Sum,Cout]) :-
        sat(
                    card([0-1],[F1,F2,F3,F4,F5]) *
                    (F1 + (U1 =:= X * Cin)) *
                    (F2 + (U2 =:= Y * U3)) *
                    (F3 + (Cout =:= U1 + U2)) *
                    (F4 + (U3 =:= X # Cin)) *
                    (F5 + (Sum =:= Y # U3))
                ).

fault_ex(1, Faults) :- fault(Faults, [1,1,0], [1,0]).
fault_ex(2, Faults) :- fault(Faults, [1,0,1], [0,0]).

| ?- fault_ex(I,L), labeling(L).

I = 1, L = [0,0,0,1,0] ? ;
I = 2, L = [1,0,0,0,0] ? ;
I = 2, L = [0,0,1,0,0] ? ;
no
| ?- fault([0,0,0,0,0], [x,y,cin], [Sum,Cout]).

sat(Cout=:=x*cin#x*y#y*cin),
sat(Sum=:=cin#x#y)
```

Példa: Tranzisztoros áramkör verifikálása




```
n(D, G, S) :-    % Gate => Drain = Source
        sat( G*D =:= G*S).

p(D, G, S) :-    % ~ Gate => Drain = Source
        sat( ~G*D =:= ~G*S).

xor(A, B, Out) :-
        p(1, A, X),
        n(0, A, X),
        p(B, A, Out),
        n(B, X, Out),
        p(A, B, Out),
        n(X, B, Out).
```

```
| ?- n(D, 1, S).                S = D ?

| ?- n(D, 0, S).                true ?

| ?- p(D, 0, S).                S = D ?

| ?- p(D, 1, S).                true ?

| ?- xor(a, b, X).              sat(X=:=a#b) ?
```

Minesweeper clpb-ben

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

mine(Rows, Cols, Mines, Bd) :-
  length(Bd, Rows), all_length(Bd, Cols), append_lists(Bd, All),
  sat(card([Mines], All)), play_mine(Bd, []).

all_length([], _).
all_length([L|Ls], Len) :- length(L, Len), all_length(Ls, Len).

append_lists([], []).
append_lists([L|Ls], Es) :-
  append_lists(Ls, Es0), append(L, Es0, Es).

play_mine(Bd, Asked) :- select_field(Bd, Asked, R, C, E), !,
  format('Row ~w, col ~w (m for mine)? ', [R,C]), read(Ans),
  process_ans(Ans, E, R, C, Bd), play_mine(Bd, [R-C|Asked]).
play_mine(_Bd, _Asked).

select_field(Bd, Asked, R, C, E) :-
  nth(R, Bd, L), nth(C, L, E), non_member(R-C, Asked), taut(E, 0), !.
select_field(Bd, _Asked, R, C, E) :-
  nth(R, Bd, L), nth(C, L, E), \+ taut(E,1), !.

process_ans(m, 1, _, _, _) :- format('Mine!~n', []), !, fail.
process_ans(Ans, 0, R, C, Bd) :-
  integer(Ans), neighbs(n(R, C, Bd), Ns), sat(card([Ans], Ns)).

neighbs(RCB, N7) :-
  neighbour(-1,-1, RCB, [], N0), neighbour(-1, 0, RCB, N0, N1),
  neighbour(-1, 1, RCB, N1, N2), neighbour( 0,-1, RCB, N2, N3),
  neighbour( 0, 1, RCB, N3, N4), neighbour( 1,-1, RCB, N4, N5),
  neighbour( 1, 0, RCB, N5, N6), neighbour( 1, 1, RCB, N6, N7).

neighbour(ROf, COf, n(R0, C0, Bd), Nbs, [E|Nbs]) :-
  R is R0+ROf, C is C0+COf, nth(R, Bd, Row), nth(C, Row, E), !.
neighbour(_, _, _, Nbs, Nbs).
```

A Mercury nagyhatékonyságú LP megvalósítás – áttekintés

Célok, alapelvek

- Tiszta deklaratív logikai programozás
- Nagy, megbízható, hatékony valódi alkalmazások létrehozásának támogatása
- A programozási munka hatékonyságának növelése

Fő tulajdonságok

- erős polimorfikus típus-rendszer

```
:- type tree(K,V) ---> empty ; node(K,V,tree(K,V),tree(K,V)).
:- pred lookup(tree(K, V), K, V).
```

- erős mód-rendszer (argumentumok be- és kimenő voltának jelzése)

```
:- mode lookup(in, in, out).
```

- erős determinizmus-rendszer

```
:- mode lookup(in, in, out) is semidet.
```

det	pontosan egy megoldás
semidet	legfeljebb egy megoldás
multi	legalabb egy megoldás
nondet	akárhány megoldás

- modul-rendszer
- magasabbrendű szerkezetek (`solutions` hasonló `bagof`-hoz)
- teljesen deklaratív (még a be- és kivitel is)

Mercury példaprogram

File-név illesztés

A feladat: operációs rendszerek file-név-illesztéséhez hasonló funkció megvalósítása.

Adott minta és karaktersorozat illesztésekor

- A `?` egy tetszőleges karakterrel illeszthető.
- A `*` egy tetszőleges (esetleg üres) karakter-sorozattal illeszthető.
- A `\`c karakter-pár a c karakterrel illeszthető (ha egy minta `\`-re végződik, az illesztés meghiúsul).
- Bármely más karakter csak önmagával illeszthető.

A Mercury program hívási formája:

```
match Pattern1 Name Pattern2
```

Itt a `Pattern1` és `Pattern2` mintákban a `*` és `?` karaktereknek azonos számban és azonos elrendezésben kell előfordulniuk.

A program funkciója

- a `Pattern1` mintára (az összes lehetséges módon) illeszti a `Name` nevet,
- a `*` és `?` karakterek helyébe kerülő szövegeket a `Pattern2` mintába rendre behelyettesíti,
- és az így kapott neveket kiírja.

A file-név-illesztő Mercury program listája

```
:- module match.
/*-----------------------------------------------------*/
:- interface.

:- import_module io.
:- pred main(io__state::di, io__state::uo) is det.

/*-----------------------------------------------------*/
:- implementation.
:- import_module list, int, std_util, string.

main -->
    io__command_line_arguments(Args),
    (
        { Args = [P1,N1,P2] } ->
        { solutions( match(P1, N1, P2), Sols) },
        io__write_string("Pattern '"), io__write_string(P1),
        io__write_string("' matches '"), io__write_string(N1),
        io__write_string("' as\n"),
        io__write_string("pattern '"), io__write_string(P2),
        io__write_string("' matches the following:\n\n"),
        write_sols(Sols)
    ;
        io__write_string("Usage: match <p1> <n1> <p2>\n")
    ).

:- pred write_sols(list(string)::in, io__state::di,
          io__state::uo) is det.
write_sols([])-->
    io__write_string("\n*** No (more) solutions\n\n").
write_sols([S|Ss])-->
    io__write_string("    "), io__write_string(S),
    io__write_string("\n"), write_sols(Ss).
```

Mercury példaprogram *(folytatás)*

```
:- pred match(string::in, string::in, string::in,
      string::out) is nondet.
match(Pattern1, Namé, Pattern2, Name2):-
    string__to_char_list(Pattern1, Ps1),
    string__to_char_list(Namé, Cs1),
    string__to_char_list(Pattern2, Ps2),
    match_list(Ps1, Cs1, L),
    match_list(Ps2, Cs2, L),
    string__from_char_list(Cs2, Name2).

:- type subst ---> any(list(char))  ;  one(char).

:- pred match_list(list(char), list(char), list(subst)).
:- mode match_list(in, in, out) is nondet.
:- mode match_list(in, out, in) is nondet.
match_list([], [], []).
match_list([?|Ps], Cs, [one(X)|L]):-
    Cs = [X|Cs1],
    match_list(Ps, Cs1, L).
match_list([*|Ps], Cs, [any(X)|L]):-
    list__append(X, Cs1, Cs),
    match_list(Ps, Cs1, L).
match_list([\, C|Ps],  [C|Cs], L):-
    match_list(Ps, Cs, L).
match_list([C|Ps], [C|Cs], L):-
    C \= (*), C \= (?), C \= (\),
    match_list(Ps, Cs, L).
```

Négyzetdarabolás Mercury-ban

```
:- module sq.
:- interface.
:- import_module io.

:- pred main(io__state::di, io__state::uo) is cc_multi.

:- implementation.

:- import_module list, int, std_util, string.

main -->
    io__command_line_arguments(Args),
    (
        { Args = [Size0|Ss0] }, { string__to_int(Size0, Size) },
        { list__map(string__to_int, Ss0, Ss) } ->
        (
            { squares(Ss, Size, SXYs) } -> write_sols(SXYs)
        ;
            io__write_string("No solution\n")
        )
    ;
        io__write_string("Usage: sq <size> <sq1> <sq2> ... \n")
    ).

:- pred write_sols(list(placement)::in, io__state::di,
           io__state::uo) is det.
write_sols([]) -->
    io__write_string("\n").
write_sols([s(S,X,Y)|Ss]) -->
    io__write_string("square size "), io__write_int(S),
    io__write_string(" placed at ("), io__write_int(X),
    io__write_string(","), io__write_int(Y),
    io__write_string(")\n"), write_sols(Ss).
```

```
:- type placement ---> s(size, xcoord, ycoord).
:- type size == int.
:- type xcoord == int.
:- type ycoord == int.
:- type segment ---> xcoord/ycoord.
:- type line == list(segment).
:- type point == pair(xcoord, ycoord).

:- pred squares(list(size), size, list(placement)).
:- mode squares(in, in, out) is nondet.
squares(Ss, Size, SXYs) :-
    MSize is -Size,
    squares(Ss, [MSize/Size,Size/0], Size, [], SXYs).

:- pred squares(list(size)::in, line::in, size::in,
        list(placement)::in, list(placement)::out) is nondet.
squares(Ss0, L0, Size, SXYs0, SXYs) :-
    (
        Ss0=[] -> SXYs=SXYs0
    ;
        L0=[V0/_|_], Y0 is Size+1+V0,
        hole(L0, 1-Y0, XY, L1, L2),
        fill_hole(L2, L3, XY, Size, Ss0, Ss, SXYs0, SXYs1),
        list__append(L1, L3, L),
        squares(Ss, L, Size, SXYs1, SXYs)
    ).
```

```
:- pred hole(line::in, point::in, point::out,
         line::out, line::out) is semidet.
hole(LL0, XY0, XY, L1, L2) :-
    LL0=[V1/H1|LL1], LL1=[V2/H2|L],
    (   H1=0 -> V is V1+V2,
        hole([V/H2|L], XY0, XY, L1, L2)
    ;   V2=0 -> H is H1+H2,
        hole([V1/H|L], XY0, XY, L1, L2)
    ;   H1>0, V1<0, V2>0 -> L1=[], L2=LL0, XY=XY0
    ;   XY0=X0-Y0, X1 is X0+H1, Y1 is Y0+V2,
        L1=[V1/H1|L11],
        hole(LL1, X1-Y1, XY, L11, L2)
    ).

:- pred fill_hole(line::in, line::out, point::in, size::in,
        list(size)::in, list(size)::out,
        list(placement)::in, list(placement)::out) is nondet.
fill_hole([V0/0,V/H|L], [V1/H|L], _XY, _Size,
                        Ss0, Ss0, SXYs0, SXYs0) :-
    !, V1 is  V0+V.
fill_hole([V/H|L0], [V1/S|L1], X-Y, Size,  Ss0, Ss,
                        SXYs0, [s(S,X,Y)|SXYs]) :-
    H > 0, Siz1 is Size+1-Y, int__min(H, Siz1, MaxS),
    select_lseq(S, MaxS, Ss0, Ss1),
    V1 is V+S, MS is -S, H1 is H-S, X1 is X+S,
    fill_hole([MS/H1|L0], L1, X1-Y, Size, Ss1, Ss, SXYs0, SXYs).

:- pred select_lseq(int::out, int::in,
                    list(int)::in, list(int)::out) is nondet.
select_lseq(S, Mx, [S0|Ss0], Ss) :-
    (   S0 > Mx -> Ss = [S0|Ss1], select_lseq(S, Mx, Ss0, Ss1)
    ;   S0 = S, Ss = Ss0
    ;   Ss = [S0|Ss1], Mx1 is S0-1,
        select_lseq(S, Mx1, Ss0, Ss1)
    ).
```

A program fordítása és futása

```
19:03 ~/oktatas/sq> mmake sq.depend
mc --generate-dependencies  sq.m
rm -f sq_init.c
19:03 ~/oktatas/sq> mmake sq
make -f /tmp/mmake.6313 -r  sq.c
rm -f sq.c
mc --compile-to-c --grade asm_fast.gc  sq.m > sq.err2 2>&1
mgnuc --grade asm_fast.gc  -c sq.c -o sq.o
rm sq.c
c2init  sq.m > sq_init.c
mgnuc --grade asm_fast.gc  -c sq_init.c -o sq_init.o
ml -s asm_fast.gc  -o sq sq_init.o \
sq.o
19:09 ~/oktatas/sq> mmake sq.sicstus
msc  -o sq.ql sq.m
msc: compiling `sq.m'
{fcompiling /tmp/msc6400.pl...}
{/tmp/msc6400.pl fcompiled, 530 msec 4304 bytes}
msl  -o sq.sicstus sq.ql
SICStus 3  #5+FD: Fri Apr 25 13:15:20 MET DST 1997
...
{SICStus state saved in .../oktatas/sq/sq.sicstus}
19:11 ~/oktatas/sq> time sq 112 50 42 37 35 33 29 27 ...
square size 33 placed at (1,80)
...
square size 50 placed at (1,1)
square size 35 placed at (51,1)
square size 27 placed at (86,1)

88.460u 1.670s 1:32.54 97.3% 0+0k 0+0io 93pf+0w
19:12 ~/oktatas/sq> time sq.sicstus 112 50 42 37 35 33  ...
...
375.800u 10.370s 6:36.42 97.4% 0+0k 0+0io 258pf+0w
```

A Mercury nyelv szintaxisa

Áttekintés

- hasonló a Prologhoz, de
- új deklarációk: típus, mód, determinizmus, stb.
- új jelölés: függvények

Egy Mercury program felépítése

- program: modulok halmaza
- modul (egy file): programelemek (deklarációk vagy klózok) sorozata
- programelem: egy term, ponttal lezárva
- term: lexikai elemek sorozata

Lexikai elemek

- azonosak az ISO Prologgal
- SICStus-hoz képest:
  - karakterek jelölése: 1-betűs nevek
  - string: külön adattípus (nem kódok listája)

Termek

- szintaktikailag azonosak az ISO Prologgal, de új operátorfajták: `fxx`, `fxy`, `fyx`.
- jelentésben különböznek:
  - függvénynek deklarált nevű termek
  - lambda kifejezések
  - magasabbrendű függvény-alkalmazások

Deklarációk

- típusokkal kapcsolatosak: `type`, `pred`, `func`
- módokkal kapcsolatosak: `inst`, `mode`
- modulokkal kapcsolatosak: `module`, `interface`, `implementation`, `import_module`, `external`, `end_module`
- a fordítóprogram vezérlése: `pragma`

Klózok

- függvény-tényállítás alakja: *Head* `=` *Result*
- tényállítás: term, amelynek funktora nem `:-/1`, `:-/2`, `-->/2`, `=/2`
- függvény-szabály alakja: *Head* `=` *Result* `:-` *Body*
- szabály alakja: *Head* `:-` *Body* vagy DCG: *Head* `-->` *Body*
- korábbi `func` vagy `pred` deklaráció kell vonatkozzék ra.

Célok

- `some` *Vars Goal* — egzisztenciális kvantor;
- `any` *Vars Goal* — univerzális kvantor ≡ `not (some` *Vars*`, not` *Goal*`)`;
- `call(`*Closure*`)`, `call(`*Closure, Arg1*`)`, `call(`*Closure, Arg1, Arg2*`)`, ... — magasabbrendű hívások;
- *CondGoal* `->` *ThenGoal* `;` *ElseGoal* — if-then-else (kötelező else!);
- diszjunkció, konjunkció, `true`, `fail`, negáció (`not` vagy `\+`) — a szokásos;
- *Goal1* `=>` *Goal2* — implikáció, azonos a `not (`*Goal1*`, not` *Goal2*`)` céllal;
- `<=`, `<=>` — hasonló értelemben;
- *Term1* `=` *Term2*, *Term1* `\=` *Term2* — egyenlő, ill. nem-egyenlő;
- minden más cél egy predikátum meghívása.

Típusok

Beépített típusok

- primitív típusok: `char, int, float, string`
- magasabbrendű típusok: `pred, pred(T), pred(T1, T2), ...`
  `func(T1) = T, func(T1, T2) = T, ...`
- univerzális típus: `univ`
- további könyvtári típusok, pl
  `list(T), pair(T1,T2), ...` , `io__state` (világ-állapot)

Definiált típusok

- Megkülonböztetett únió:
  <úniótípus> ::= `:- type` <típusazonosító> `--->`
  <típustörzs>
  <típustörzs> ::= <konstruktor>
  ; ...
  <konstruktor> ::= <név>
  | <rekordnév>(<típusazonosító>, ...)
  <típusazonosító> ::= <név>
  | <név>(<változó>,...)
- típusátnevezés:
  <típusátnevezés> ::= `:- type` <típusazonosító> == <típusazonosító>
- absztrakt típus:
  <absztrakt típus> ::= `:- type` <típusazonosító>

Predikátum- és függvénydeklarációk

- predikátumdeklaráció: `:- pred` <elj>(<típusazonosító>, ...).
- függvénydeklaráció:
  `:- function` <f>(<típusazonosító>, ...) `=` <típusazonosító>

Példák Mercury típusokra

Megkülonböztetett úniók

```
:- type gyumolcs ---> alma ; korte; szilva.
    % egy gyümölcs a fenti három atom valamelyike.

:- type szemely ---> szemely(string, string, int).
    % olyan három-argumentumú struktúra amelynek
    % elsö két argumentuma string, a harmadik egész.

:- type list(T) --->   []
        ;              [T|list(T)].

:- type pair(T1, T2) ---> T1 - T2.
    % egy olyan '-' nevü kétargumentumú struktúra,
    % amelynek argumentumai T1 ill. T2 típusúak
```

Típusátnevezések

```
:- type assoc_list(KeyType, ValueType)
          == list(pair(KeyType, ValueType)).

:- type szotar == assoc_list(magyar, angol).
:- type magyar == string.
:- type angol == string.
```

Absztrakt típusok

```
:- type titkos.
:- type titkos1(T1, T2)
```

Predikátum-deklarációk

```
:- pred member(T, list(T)).
:- func length(list(T)) = int.
```

Módok

Behelyettesítettségi állapot (intantiatedness)

- egy argumentumról elmondja, hogy mely részei üresek és melyek behelyettesítettek
- építőkövek: `free`, `bound( ... ; ... )`, `ground`
- példa: `:- inst listskel = bound( [] ; [free | listskel] ).`

Módok deklarálása

- új mód: `:- mode m :: inst1 -> inst2`
- átnevezés: `:- mode m1 :: m2`
- például: `:- mode in :: ground -> ground.`
  `:- mode out :: free -> ground.`
  `:- mode (+) :: in.  :- mode (-) :: out.`

Paraméteres mód- és inst-deklarációk

```
:- inst listskel(Inst) = bound( [] ; [Inst|listskel(Inst)] ).
:- mode m :: listskel(listskel(free)) -> ground.
:- mode in(Inst) :: Inst -> Inst.
:- mode out(Inst) :: free -> Inst.
```

Predikátumok és függvények módjának deklarálása

```
:- func length(list(T)) = int.
:- mode length(in(listskel)) = out.
:- mode length(out(listskel)) = in.

:- pred append(list(T), list(T), list(T)).
:- mode append(in, in, out).
:- mode append(out, out, in).
```

Korlátozások

- Változók összekapcsolása nincs megengedve, pl. ez hibás:
  `:- mode append(in(listskel), in(listskel), out(listskel)).`
- A jelenlegi val1tozat csak `ground` és `free` inst-eket kezel.

Determinizmus

Minden egyes predikátum/függvény-módhoz egy determinizmust rendelünk

	Maximum number of solutions		
Can fail?	0	1	> 1
no	erroneous	det	multi
yes	failure	semidet	nondet

Példák

```
:- pred append(list(T), list(T), list(T)).
:- mode append(in, in, out) is det.
:- mode append(out, out, in) is multi.
:- mode append(in, in, in) is semidet.

:- func length(list(T)) = int.
:- mode length(in) = out is det.
:- mode length(in(list_skel)) = out is det.
:- mode length(in) = in is semidet.
```

Összevont predikátumdeklarációk

- ha csak egy mód van
- a mód és determinizmus a `pred` deklarációban is megadható
- `:- pred append(list(T)::in, list(T)::in, list(T)::out) is det.`

CC (committed choice) nemdeterminizmus

- olyan választási pont, amelyre sosem lépünk vissza
- például: `main/2`, vagy egy kimenő változók nélküli cél
- két új determinizmus: `cc_multi, cc_nondet`

További lehetőségek

Egyedi módok

- egyedi inst-ek: `unique`, `unique(...)`, olyan mint `ground` és `bound(...)`, de csak egyszer hivatkozunk rá
- `dead`: olyan érték, amire nincs több hivatkozás
- Ezekre épülö szabványos módok:

```
:- mode uo :: free -> unique.     % unique output
:- mode ui :: unique -> unique.   % unique input
:- mode di :: unique -> dead.     % destructive input
```

- ha visszalépéskor kell az érték, akkor `mostly_unique` ill. `mostly_dead` (nincs megvalósítva).

Magasabbrendű programozás

- Legyen a `sum` deklarációja:

```
:- pred sum(list(int), int).
:- mode sum(in, out) is det.
```

  Ekkor a következők ekvivalensek:

```
X = lambda([List::in, Length::out] is det, sum(List, Length))
Y = (pred(List::in, Length::out) is det :- sum(List, Length))
Z = sum
```

- Az `X`, `Y`, `Z` változók típusa: `pred(list(int), int)`
- Currying: `Sum123 = sum([1,2,3])` típusa `pred(int)`.
- Magasabbrendű predikátumok meghívása: `call/1,2,3...`
- Fontos könyvtári eljárás (vö: `bagof/3`):

```
:- pred solutions(pred(T), list(T)).
:- mode solutions(pred(out) is nondet, out) is det.
```

- `pred(out) is nondet` — ez egy magasabbrendű mód.

Modulok

A modulok szerkezete

:- module <n>.
:- interface.
<látható rész>
:- implementation.
<lokális definíciók>
:- end_module <n>.

Az indítási pont

- :- pred main(io__state::di, io__state::uo) is <det>
- ahol <det> det vagy cc_multi

Példa absztrakt adattípusra

```
:- module stack.
:- interface.
:- type stack(T).     % absztrakt adattípus

:- func empty_stack = stack(T).
:- func push(stack(T), T) = stack(T).
:- pred pop(stack(T)::in, T::out, stack(T)::out) is semidet.

:- implementation.
:- import_module list.
:- type stack(T) == list(T).

empty_stack = [].
push(Xs, X) = [X|Xs].
pop([X|Xs], X, Xs).

:- end_module stack.
```