FACULDADE DE ENGENHARIA
UNIVERSIDADE DO PORTO

DEPARTAMENTO DE ENGENHARIA INFORMÁTICA

Programação em Lógica com Restrições no SICStus Prolog

Novembro de 2019

SICStus Prolog User's Manual (Release 4.5.1)

Secção 10.10: Constraint Logic Programming over Finite Domains—library(clpfd)

Parcialmente baseado em slides anteriores de
Henrique Lopes Cardoso (hlc@fe.up.pt),
Luís Paulo Reis (lpreis@fe.up.pt),
outros autores e no manual
do SCIStus Prolog
(v4.5.1)

Daniel Castro Silva
dcs@fe.up.pt

25/11/2019

Conteúdo

1. Domínios de Restrições Disponíveis
2. Interface do solver CLP(FD)
   - Estrutura de um Programa em PLR
   - Declaração de Domínios
   - Colocação de Restrições
   - Restrições Materializadas
3. Restrições Disponíveis
4. Predicados de Enumeração
   - Pesquisa e otimização
5. Predicados de Estatísticas

PLR no SICStus Prolog

1. DOMÍNIOS DE RESTRIÇÕES DISPONÍVEIS

Domínios Booleanos e Reais

- Booleanos:
  - Esquema clp(B)
  - *use_module(library(clpb)).*
    - Secção 10.9 do manual do SICStus

- Reais e Racionais
  - Esquema clp(Q,R)
  - *use_module(library(clpq)).* *use_module(library(clpr)).*
    - Secção 10.11 do manual do SICStus

- Não são abordados na unidade curricular de PLOG!

Domínios Finitos

- *Solver* ***clp(FD)*** é um instância do esquema geral de PLR (CLP) introduzido em [Jafar & Michaylov 87].
- Útil para modelizar problemas de otimização discreta
  - Escalonamento, planeamento, alocação de recursos, empacotamento, geração de horários, ...
- Características do *solver* ***clp(FD)***:
  - Duas classes de restrições: primitivas e globais
  - Propagadores para restrições globais muito eficientes
  - O valor lógico de uma restrição primitiva pode ser refletido numa variável binária (0/1) – materialização (ou reificação)
  - Podem-se adicionar novas restrições primitivas escrevendo indexicais
  - Podem ser escritas novas restrições globais em Prolog

PLR no SICStus Prolog

2. INTERFACE DO SOLVER CLP(FD)

Interface do Solver CLP(FD)

- O *solver* ***clp(FD)*** está disponível como uma biblioteca

  :- use_module(library(clpfd)).

- Contém predicados para testar a consistência e o vínculo (*entailment*) de restrições sobre domínios finitos, bem como para obter soluções atribuindo valores às variáveis do problema

- Um **domínio finito** é um subconjunto de inteiros pequenos e uma **restrição sobre domínios finitos** é uma relação entre um tuplo de inteiros pequenos
- Só inteiros pequenos e variáveis não instanciadas são permitidos em restrições sobre domínios finitos
  - Inteiro pequeno: $[-2^{28}, 2^{28}-1]$ em plataformas de 32-bits, ou $[-2^{60}, 2^{60}-1]$ em plataformas de 64-bits
    - Possível usar o predicado *prolog_flag/2* para obter estes valores

Interface do Solver CLP(FD)

- Todas as **variáveis de domínio** têm um domínio finito associado, declarado explicitamente no programa ou imposto implicitamente pelo *solver*
  - Temporariamente, o domínio de uma variável pode ser infinito, se não tiver um limite mínimo (*lower bound*) ou máximo (*upper bound*) finito
  - O domínio das variáveis vai-se reduzindo à medida que são adicionadas restrições
- Se um domínio ficar vazio, então as restrições não são, em conjunto, "satisfazíveis", e o ramo atual de computação falha
- No final da computação é usual que cada variável tenha o seu domínio restringido a um único valor (*singleton*)
  - Para tal é necessária, tipicamente, alguma pesquisa
- Cada restrição é implementada por um (conjunto de) propagador(es)
  - Indexicais
  - Propagadores globais

Estrutura de um Programa em PLR

- Um programa em PLR estrutura-se nas três etapas seguintes:
  - Declaração de variáveis e seus domínios
  - Declaração de restrições sobre as variáveis
  - Pesquisa de uma solução

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

example:-
        A in 1..7,                           } variáveis e
        domain( [B, C], 1, 10),              } domínios
        A + B + C #= A * B * C,              } restrições
        A #> B,                              }
        labeling( [], [A, B, C] ).           } pesquisa
                                               de solução
```

```
| ?- example.
A = 2,
B = 1,
C = 3 ?
```

Estrutura de um Programa em PLR

- Ordem destas etapas é importante
  - Se invertermos a ordem, colocando primeiro a pesquisa de solução e depois as restrições, resulta no mecanismo *Generate&Test* tradicional, muito menos eficiente

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

badExample:-
        A in 1..7,
        domain( [B, C], 1, 10),
        labeling( [], [A, B, C] ),
        write('.'),
        A + B + C #= A * B * C,
        A #> B.
```

```
| ?- badExample.
.......................................
.......................................
.......................
A = 2,
B = 1,
C = 3 ?
```

Domínios das Variáveis

- Uma variável pode ter o seu domínio declarado usando ***in/2*** e um intervalo (*ConstantRange*):
  - *NotaPLOG in 16..20*

- A definição de *ConstantRange* permite a declaração de domínios mais complexos
  - *VarA in (2..8) \/ (15..20)*
  - *VarB in {4, 8, 15, 16, 23, 42}*

```
ConstantSet      ::= {integer,…,integer}
ConstantRange    ::= ConstantSet
                   | Constant .. Constant
                   | ConstantRange /\ ConstantRange
                   | ConstantRange \/ ConstantRange
                   | \ ConstantRange
```

Domínios das Variáveis

- Pode ainda ser usado ***in_set/2*** para declaração de domínio de uma variável
  - O segundo argumento de ***in_set/2*** é um *Finite Domain Set*, que pode ser obtido a partir de uma lista usando o predicado ***list_to_fdset(+List, -FD_Set)***.
    - Ver secção 10.10.9.3 para operações sobre *FD Sets*

Numbers = [4, 8, 15, 16, 23, 42],
list_to_fdset(Numbers, FDS_Numbers),
Var in_set FDS_Numbers.

Domínios das Variáveis

- Para declarar um mesmo domínio para uma lista de variáveis pode ser usado o predicado ***domain(+List_of_Variables, +Min, +Max)***:
  - *domain([A, B, C], 5, 12)*

- Outras restrições limitam domínios das variáveis envolvidas
  - *A #> 8*
  - *B + C #< 12*
  - *A + B + C #= 20*

Colocação de Restrições

- Uma restrição é chamada como qualquer outro predicado *Prolog*

```
| ?-  X in 1..5, Y in 2..8,
      X+Y #= T.
X in 1..5,
Y in 2..8,
T in 3..13
```

```
| ?-  X in 1..5, T in 3..13,
      X+Y #= T.
X in 1..5,
T in 3..13,
Y in -2..12
```

- A existência de uma resposta mostra a existência de domínios válidos para as variáveis
  - Não são visualizadas as restrições associadas a cada variável

Colocação de Restrições

- Ao colocar uma restrição, é chamado o mecanismo de propagação, que limita os domínios das variáveis
  - Este mecanismo pode ser computacionalmente pesado em alguns casos
- É possível colocar um conjunto de restrições de uma vez (em lote), suspendendo o mecanismo de propagação até que todas estas restrições tenham sido todas colocadas
  - ***fd_batch(+Constraints)***
    - Onde *Contraints* é uma lista de restrições a colocar
  - *domain([A,B,C], 5, 12),*
    fd_batch([A #> 8, B + C #< 12, A + B + C #= 20])

Restrições Materializadas (*Reified*)

- Por vezes é útil fazer refletir o valor de verdade de uma restrição numa variável booleana B (0/1) tal que:
  - A restrição é colocada se B for colocado a 1
  - A negação da restrição é colocada se B for colocado a 0
  - B é colocado a 1 se a restrição for vinculada (*entailed*)
  - B é colocado a 0 se a restrição não for vinculada (*disentailed*)

- Este mecanismo é conhecido como materialização (*reification*)

- Uma restrição materializada é escrita da forma:

  Constraint #<=> B.

  onde *Constraint* é uma restrição materializável

Restrições Materializadas (*Reified*)

- Exemplo: ***exactly(X,L,N)***
  - Verdadeira se ***X*** ocorre exatamente ***N*** vezes na lista ***L***
  - Pode ser definida como:

```
exactly(_, [], 0).
exactly(X, [Y|L], N) :-
        X #= Y #<=> B,
        N #= M + B,
        exactly(X, L, M).
```

- Restrições materializáveis podem ser usadas como termos em expressões aritméticas:

```
| ?-  X #= 10,
      B #= (X#>=2) + (X#>=4) + (X#>=8).
B = 3,
X = 10
```

```
| ?-  X in 1..3,
      B #= (X#>=1) + (X#>=2) + (X#>=3),
      labeling([],[X]).
X = 1, B = 1 ? ;
X = 2, B = 2 ? ;
X = 3, B = 3 ? ;
no
```

PLR no SICStus Prolog

3. RESTRIÇÕES DISPONÍVEIS

Restrições Disponíveis

- Restrições Aritméticas
- Restrições de Pertença
- Restrições Proposicionais
- Restrições Combinatórias
  - Aritmético-lógicas
  - Extensão
  - Grafo
  - Escalonamento
  - Posicionamento
  - Sequenciamento

Restrições Aritméticas

- ***?Expr RelOp ?Expr***
  - ***RelOp***: *#= | #\= | #< | #=< | #> | #>=*
  - Expressões podem ser lineares ou não lineares.
  - Expressões lineares conduzem a maior propagação
    - Por exemplo, *X/Y* e *X mod Y* bloqueiam até *Y* estar "*ground*" (definido)
  - Restrições aritméticas lineares mantêm consistência de intervalos
  - Restrições Aritméticas podem ser materializadas
  - Exemplo:

```
| ?-   X in 1..2, Y in 3..5,
       X#=<Y #<=> B.
B = 1,
X in 1..2,
Y in 3..5
```

Soma

- ***sum(+Xs, +RelOp, ?Value)***
  - ***Xs*** é uma lista de inteiros ou variáveis de domínio, ***RelOp*** é um operador relacional e ***Value*** é um inteiro ou variável de domínio
  - Verdadeira se *sum(Xs) RelOp Value* (a soma dos elementos de ***Xs*** tem a relação ***RelOp*** com ***Value***)
  - Corresponde aproximadamente a *sumlist/2* da *library(lists)*
  - Utiliza um algoritmo dedicado e é muito mais eficiente do que a colocação de uma série de restrições simples
  - Não pode ser materializada
  - Exemplos:

```
| ?- domain([X,Y], 1, 10),
     sum([X,Y], #<, 10).
X in 1..8,
Y in 1..8
```

```
| ?- domain([X,Y], 1, 10),
     sum([X,Y], #=, Z).
X in 1..10,
Y in 1..10,
Z in 2..20
```

Produto Escalar

- ***scalar_product(+Coeffs, +Xs, +RelOp, ?Value)***
- ***scalar_product(+Coeffs, +Xs, +RelOp, ?Value, +Options)***
  - ***Coeffs*** é uma lista de comprimento *n* de inteiros, ***Xs*** é uma lista de comprimento *n* de inteiros ou variáveis de domínio, ***RelOp*** é um operador relacional e ***Value*** é um inteiro ou variável de domínio
  - Verdadeira se *sum(Coeffs*Xs) RelOp Value*
  - Utiliza um algoritmo dedicado e é muito mais eficiente do que a colocação de uma série de restrições simples
  - ***Options*** é uma lista de opções
    - ***among(Least, Most, Range)*** indica que no mínimo ***Least*** e no máximo ***Most*** elementos de ***Xs*** têm valores no intervalo (*ConstantRange*) indicado em ***Range***
    - ***consistency(Cons)*** indica que nível de consistência deve ser usado pela restrição. ***Cons*** pode tomar os valores ***domain*** (deve manter consistência de domínios; útil apenas se ***RelOp*** for *#=* e todas as variáveis de domínio devem ter domínios finitos); ***bounds*** ou ***value*** (opção por omissão), indica consistência de limites.

Produto Escalar

- ***scalar_product_reif(+Coeffs, +Xs, +RelOp, ?Value, ?Reif)***
- ***scalar_product_reif(+Coeffs, +Xs, +RelOp, ?Value, ?Reif, +Options)***
  - Versão com reificação de scalar_product/4 e /5.
  - Equivalente a materializar a restrição anterior
  - Exemplo:

```
| ?-  domain([A,B,C], 1, 5),
      scalar_product([1,2,3], [A,B,C], #=, 10).
A in 1..5,
B in 1..3,
C in 1..2
```

Mínimo/Máximo

- ***minimum(?Value, +Xs)***
- ***maximum(?Value, +Xs)***
  - ***Xs*** é uma lista de inteiros ou variáveis de domínio e ***Value*** é um inteiro ou variável de domínio
  - Verdadeira se ***Value*** é o mínimo (máximo) de ***Xs***
  - Corresponde a *min_member/2* (*max_member/2*) da *library(lists)*
  - Não podem ser materializadas
  - Exemplos:

```
| ?- domain([A,B], 1, 10), C in 5..15,
     minimum(C, [A,B]).
A in 5..10,
B in 5..10,
C in 5..10
```

```
| ?- domain([A,B,C], 1, 5),
     sum([A,B,C], #=, 10),
     maximum(3, [A,B]).
A in 2..3,
B in 2..3,
C in 4..5
```

Restrições de Pertença (*Membership*)

- Predicados de definição de domínios das variáveis
  - ***domain(+Vars, +Min, +Max)***
    - Verdadeira se todos os elementos de ***Vars*** estão no intervalo ***Min..Max***
  - ***?X in +Range***
    - Verdadeira se ***X*** é um elemento do intervalo ***Range***
  - ***?X in_set +FDSet***
    - Verdadeira se ***X*** é um elemento do conjunto ***FDSet***
  - *in/2* e *in_set/2* mantêm consistência do domínio e são materializáveis
  - Exemplos:

```
| ?- domain([X],1,3),
     X in 3..5 #<=> B, labeling([],[X]).
X = 1, B = 0 ? ;
X = 2, B = 0 ? ;
X = 3, B = 1 ? ;
no
```

```
| ?- X in {1,2,3,5}.
X in(1..3)\/{5}

| ?- list_to_fdset([1,2,3,5],FD), X in_set FD.
FD = [[1|3],[5|5]],
X in(1..3)\/{5}
```

Restrições Proposicionais

- Podem definir fórmulas proposicionais sobre restrições materializáveis
- Exemplo: X #= 4 #\/ Y #= 6
  - Expressa a disjunção de duas restrições de igualdade
- As folhas das fórmulas proposicionais podem ser restrições materializáveis, as constantes 0 e 1, ou variáveis binárias (0/1)
- Podem ser definidas novas restrições materializáveis primitivas com “indexicais”
- Mantêm consistência do domínio
- Exemplo:

```
| ?- X in 1..2, Y in 1..10, X #= Y #\/ Y #< X, labeling([], [X,Y]).
X = 1, Y = 1 ? ;
X = 2, Y = 1 ? ;
X = 2, Y = 2 ? ;
no
```

Restrições Proposicionais

#\ :Q	verdadeira se a restrição Q for falsa (NOT)
:P #/\ :Q	verdadeira se as restrições P e Q são ambas verdadeiras (AND)
:P #\ :Q	verdadeira se exatamente uma das restrições P e Q é verdadeira (XOR)
:P #\/ :Q	verdadeira se pelo menos uma das restrições P e Q é verdadeira (OR)
:P #=> :Q :Q #<= :P	verdadeira se a restrição Q é verdadeira ou se a restrição P é falsa (implicação)
:P #<=> :Q	verdadeira se P e Q são ambas verdadeiras ou ambas falsas (equivalência)

- Note-se que o esquema de materialização é um caso particular da restrição proposicional de equivalência

Restrições Combinatórias

- Restrições Combinatórias são também designadas restrições simbólicas
- Não são materializáveis
- Normalmente mantêm consistência de intervalos nos seus argumentos

Arithmetic-Logical

- *smt/1 (deprecated)*
- *count/4 (deprecated)*
- *global_cardinality/[2,3]*
- *all_different/[1,2]*
- *all_distinct/[1,2]*
- *nvalue/2*
- *assignment/[2,3]*
- *sorting/3*
- *keysorting/[2,3]*
- *lex_chain/[1,2]*
- *bool_[and,or,xor]/2*
- *bool_channel/4*

Extensional

- *element/3*
- *relation/3*
- *table/[2,3]*
- *case/[3,4]*

Graph

- *circuit/[1,2]*

Scheduling

- *cumulative/[1,2]*
- *cumulatives/[2,3]*
- *multi_cumulative/[2,3]*

Placement

- *bin_packing/2*
- *disjoint1/[1,2]*
- *disjoint2/[1,2]*
- *geost/[2,3,4]*

Automata

- *automaton/[3,8,9]*

Count

(*deprecated*, ver *global_cardinality*)

- ***count(+Val, +List, +RelOp, ?Count)***
  - Restringe o número de ocorrências do valor ***Val*** na lista ***List*** a ter a relação ***RelOp*** com o valor ***Count***
  - ***Val*** é um inteiro, ***List*** uma lista de inteiros ou variáveis de domínio, ***Count*** um inteiro ou variável de domínio, e ***RelOp*** um operador relacional
  - Mantém consistência de domínio, mas na prática ***global_cardinality/2*** é uma alternativa melhor
  - Exemplos:

```
| ?- domain([X,Y,Z], 1, 3),
     count(1,[X,Y,Z], #>, Z).
X in 1..3,
Y in 1..3,
Z in 1..2
```

```
| ?- domain([A,B,C], 1, 3), X in 2..5,
     count(1, [A,B,C], #=, X),
     labeling([], [X]).
X = 2, A in 1..3, B in 1..3, C in 1..3 ? ;
A = 1, B = 1, C = 1, X = 3 ? ;
no
```

Global Cardinality

- ***global_cardinality(+Xs, +Vals)***
- ***global_cardinality(+Xs, +Vals, +Options)***
  - Restringe o número de ocorrências de cada valor numa lista de variáveis
  - ***Xs*** é uma lista de inteiros ou variáveis de domínio; ***Vals*** é uma lista de termos ***K-V***, onde ***K*** é um inteiro único e ***V*** é uma variável de domínio ou um inteiro
  - Verdadeira se cada elemento de ***Xs*** é igual a um ***K*** e para cada par ***K-V*** exatamente ***V*** elementos de ***Xs*** são iguais a ***K***
  - Se ou ***Xs*** ou ***Vals*** estão "ground", e noutros casos especiais, mantém a consistência de domínio; a consistência de intervalos não pode ser garantida
  - ***Options*** é lista de opções (para controlar funcionamento) (ver documentação)
  - Exemplos:

```
| ?- global_cardinality([A,B,C], [1-2, 3-1]).
A in {1}\/{3},
B in {1}\/{3},
C in {1}\/{3}
```

```
| ?- A in 3..10,
     global_cardinality([A,B,C], [1-2, 3-1]).
A = 3, B = 1, C = 1
```

All Different / All Distinct

- ***all_different(+Variables)*** / ***all_different(+Variables, +Options)***
- ***all_distinct(+Variables)*** / ***all_distinct(+Variables, +Options)***
  - Restringe a que todos os valores da lista ***Variables*** sejam distintos
    - Equivalente a uma restrição ***#\=*** para cada par de variáveis
  - ***Variables*** é uma lista de inteiros ou variáveis de domínio
  - ***Options*** é uma lista de zero ou mais opções (ver documentação):
    - ***L #= R*** – restrição adicional com uma expressão
    - ***on(On)*** - quando acordar a restrição
    - ***consistency(Cons)*** - que algoritmo utilizar
  - Exemplos:

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

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

```
| ?- domain([X,Y,Z], 1, 3),
       all_different([X, Y, Z]), X #< Y,
       labeling([], [X]).
X = 1, Y in 2..3, Z in 2..3 ? ;
X = 2, Y = 3, Z = 1 ? ;
no
```

Nvalue

- ***nvalue(?N, +Variables)***
  - Restringe a lista de variáveis ***Variables*** de forma a que existam exatamente ***N*** valores distintos
  - ***Variables*** é uma lista de inteiros ou variáveis de domínio com limites finitos e ***N*** é um inteiro ou variável de domínio
  - Pode ser visto como uma versão relaxada de ***all_distinct/2***
  - Exemplos:

```
| ?- domain([X,Y], 1, 3),
     domain([Z], 3, 5),
     nvalue(2, [X,Y,Z]),
     X#\=Y, X#=1.
X = 1, Y = 3, Z = 3
```

```
| ?- domain([X,Y], 1, 3),
     domain([Z], 1, 5),
     nvalue(2, [X,Y,Z]),
     X#\=Y, X#=1.
X = 1, Y in 2..3, Z in 1..3
```

```
| ?- domain([X,Y], 1, 3),
     domain([Z], 1, 5),
     nvalue(2, [X,Y,Z]),
     X#\=Y.
X in 1..3, Y in 1..3, Z in 1..5
```

Assignment

- ***assignment(+Xs, +Ys)***
- ***assignment(+Xs, +Ys, +Options)***
  - ***Xs*** = [X1,...,Xn] e ***Ys*** = [Y1,...,Yn] são listas de comprimento n de variáveis de domínio ou inteiros
  - Verdadeiro se todos os ***Xi***, ***Yi*** estão em [1,n], são únicos para a sua lista e ***Xi=j*** sse ***Yj=i*** (as listas são duais)
  - ***Options*** é uma lista que pode conter as opções:
    - *on(On)*, *consistency(Cons)*: idênticas a **all_distinct/2**
    - *circuit(Boolean)*: se *true*, *circuit(Xs,Ys)* tem que se verificar
    - *cost(Cost,Matrix)*: permite associar um custo à restrição
  - Exemplos:

```
| ?- assignment([4,1,5,2,3], Ys).
Ys = [2,4,5,1,3]
```

```
| ?-  length(Xs, 3), domain(Xs, 1, 3),
      assignment(Xs, Ys), labeling([], Xs).
Xs = [1,2,3], Ys = [1,2,3] ? ;          Xs = [1,3,2], Ys = [1,3,2] ? ;
Xs = [2,1,3], Ys = [2,1,3] ? ;          Xs = [2,3,1], Ys = [3,1,2] ? ;
Xs = [3,1,2], Ys = [2,3,1] ? ;          Xs = [3,2,1], Ys = [3,2,1] ? ;
no
```

Sorting

- ***sorting(+Xs, +Ps, +Ys)***
  - Captura a relação entre uma lista de valores, uma lista de valores ordenada de forma ascendente e as suas posições na lista original
  - ***Xs***, ***Ps*** e ***Ys*** são listas de igual comprimento n de variáveis de domínio ou inteiros
  - A restrição verifica-se se:
    - ***Ys*** está em ordenação ascendente
    - ***Ps*** é uma permutação de [1,n]
    - Para cada i em [1,n], Xs[i] = Ys[Ps[i]]
  - Exemplos:

```
| ?- length(Ys, 5), length(Ps, 5),
     sorting([2,7,9,1,3], Ps, Ys).
Ps = [2,4,5,1,3], Ys = [1,2,3,7,9]
```

```
| ?- length(Ys, 5), length(Ps, 5),
     sorting([2,7,3,1,3], Ps, Ys).
Ps = [2,5,_A,1,_B], Ys = [1,2,3,3,7],
_A in 3..4, _B in 3..4
```

Keysorting

- ***keysorting(+Xs, +Ys)***
- ***keysorting(+Xs, +Ys, +Options)***
  - Generalização de ***sorting/3*** mas ordenando tuplos de variáveis
  - Os tuplos são separados em chave e valor, sendo ordenados apenas pela chave (mantém ordem de tuplos com a mesma chave)
  - ***Xs*** e ***Ys*** são listas, com o mesmo tamanho n, de tuplos de variáveis; todos os tuplos (listas de variáveis) têm o mesmo tamanho m
  - ***Options*** é uma lista de opções:
    - ***keys(Keys)*** - ***Keys*** é o tamanho da chave (inteiro positivo; valor por omissão é 1)
    - ***permutation(Ps)*** - Ps é lista de variáveis (permutação de [1,n], tal que para cada i em [1,n], j em [1,m] : Ys[i,j] = Xs[Ps[i],j].)
  - Exemplo:

```
ln2(X):-
   length(X,2).
```

```
| ?-  _List = [[1,5], [6,5], [4,3], [7,9], [4,5], [7,8], [3,3]],
      length(_List, _Len), length(Sorted, _Len), maplist(ln2, Sorted),
      length(P, _Len), keysorting(_List, Sorted, [permutation(P)] ).
Sorted = [[1,5],[3,3],[4,3],[4,5],[6,5],[7,9],[7,8]]
P = [1,7,3,5,2,4,6] ? ;
no
```

Lex Chain

- ***lex_chain(+Vectors)***
- ***lex_chain(+Vectors, +Options)***
  - ***Vectors*** é uma lista de vetores (listas) de inteiros ou variáveis de domínio
  - A restrição verifica-se se ***Vectors*** está por ordem lexicográfica ascendente (na realidade, não descendente por omissão)
  - ***Options*** é uma lista de opções:
    - ***op(Op)*** - ***Op*** é ***#<=*** (omissão) ou ***#<*** (estritamente ascendente)
    - ***Increasing*** - listas internas ordenadas de forma estritamente ascendente
    - ***among(Least, Most, Values)*** – entre ***Least*** e ***Most*** valores de cada ***Vector*** pertencem à lista ***Values***
  - Exemplo:

```
| ?- domain([A,B,C], 1, 2),
     lex_chain([ [A,B,C], [B,C,A], [C,B,A] ]),
     labeling([], [A,B,C]).
A = 1,B = 1,C = 1 ? ;     A = 1,B = 1,C = 2 ? ;
A = 1,B = 2,C = 2 ? ;     A = 2,B = 2,C = 2 ? ;
no
```

Element

- ***element(?X, +List, ?Y)***
  - ***X*** e ***Y*** são inteiros ou variáveis de domínio; ***List*** é uma lista de inteiros ou variáveis de domínio
  - Verdadeira se o ***X***-ésimo elemento de ***List*** é ***Y***
  - Operacionalmente, os domínios de ***X*** e ***Y*** são restringidos de forma a que, para cada elemento no domínio de ***X***, existe um elemento compatível no domínio de ***Y***, e vice-versa
  - mantém consistência de domínio em ***X*** e consistência de intervalos em ***List*** e ***Y***
  - Corresponde a ***nth1/3*** da *library(lists)*.
  - Exemplos:

```
| ?- element(X,[10,20,30],Y), labeling([],[Y]).
X = 1, Y = 10 ? ;
X = 2, Y = 20 ? ;
X = 3, Y = 30 ? ;
no
```

```
| ?- L=[A,B,C], domain(L,1,5), element(2,L,4).
B = 4,
L = [A,4,C],
A in 1..5, C in 1..5
```

Relation

(*deprecated*, ver *table*)

- ***relation(?X, +MapList, ?Y)***
  - ***X*** e ***Y*** são inteiros ou variáveis de domínio e ***MapList*** é uma lista de pares *Inteiro-ConstantRange*, onde cada chave *Inteiro* ocorre uma só vez
  - Verdadeira se ***MapList*** contém um par ***X-R*** e ***Y*** está no intervalo indicado em ***R***
  - Exemplos:

```
| ?- domain([Y], 1, 3),
     relation(X, [1-{3,4,5}, 2-{1,2}], Y),
     labeling([], [X]).
X = 1, Y = 3 ? ;
X = 2, Y in 1..2 ? ;
no
```

```
| ?- domain([Y], 1, 3),
     relation(X, [1-{3,4,5}, 2-{1,2,3}], Y),
     labeling([], [Y]).
Y = 1, X = 2 ? ;
Y = 2, X = 2 ? ;
Y = 3, X in 1..2 ? ;
no
```

Table

- ***table(+Tuples, +Extension)***
- ***table(+Tuples, +Extension, +Options)***
  - Define uma restrição n-ária por extensão
  - ***Tuples*** é uma lista de listas de variáveis de domínio ou inteiros, cada uma de comprimento ***n***; ***Extension*** é uma lista de listas de inteiros, cada uma de comprimento ***n***; ***Options*** é lista de opções que permitem controlar ordem de variáveis usada internamente e estrutura de dados e algoritmo (ver doc.)
  - A restrição verifica-se se cada *Tuple* em ***Tuples*** ocorre em ***Extension***
  - Exemplos:

```
| ?- table([[A,B]],[[1,1],[1,2],[2,10],[2,20]]).
A in 1..2,
B in (1..2)\/{10}\/{20}

| ?- table([[A,B],[B,C]],[[1,1],[1,2],[2,10],[2,20]]).
A = 1,
B in 1..2,
C in (1..2)\/{10}\/{20}
```

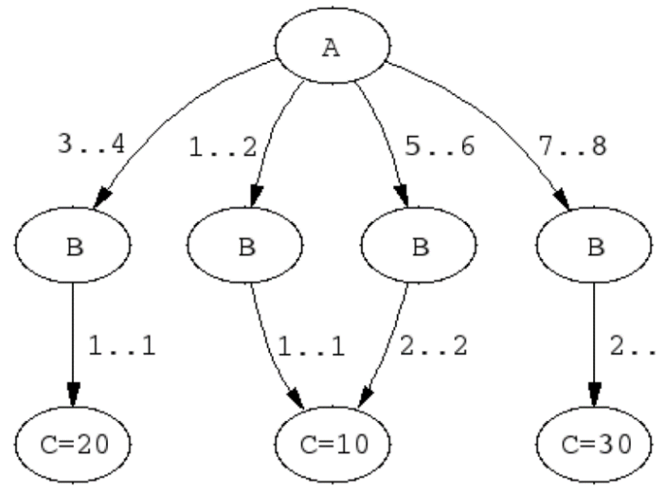
```
| ?- table([[A,B]],[[1,1],[1,2],[2,10],[2,20]]),
     labeling([],[A,B]).
A = 1, B = 1 ? ;
A = 1, B = 2 ? ;
A = 2, B = 10 ? ;
A = 2, B = 20 ? ;
no
```

Case

- ***case(+Template, +Tuples, +Dag)***
- ***case(+Template, +Tuples, +Dag, +Options)***
  - Codifica uma restrição n-ária, definida por extensão e/ou desigualdades lineares
  - Usa um DAG: nós correspondem a variáveis, cada arco é etiquetado por um intervalo admissível para a variável no nó de onde parte, ou por desigualdades lineares
  - Ordem das variáveis é fixa: cada caminho desde a raiz até a uma folha deve visitar cada variável uma vez, pela ordem em que ocorrem em ***Template***
  - ***Template*** é um termo arbitrário *non-ground*
  - ***Tuples*** é uma lista de termos da mesma forma que ***Template*** (não devem partilhar variáveis)
  - ***Dag*** é uma lista de termos na forma ***node(ID,X,Children)***, onde ***X*** é uma variável do template e ***ID*** é um inteiro identificando o nó; o primeiro nó da lista é a raiz
    - Nó interno: ***Children*** é uma lista de termos *(Min..Max)-ID2* (ou *(Min..Max)-SideConstraints-ID2*), onde *ID2* identifica um nó filho
    - Nó folha: ***Children*** é uma lista de termos *(Min..Max)* (ou *(Min..Max)-SideConstraints*)

Case

– Exemplo:




```
element(X, [1,1,1,1,2,2,2,2], Y),
element(X, [10,10,20,20,10,10,30,30], Z)
```

```
elts(X, Y, Z) :-
    case(f(A,B,C), [f(X,Y,Z)],
         [node(0, A,[(1..2)-1,(3..4)-2,(5..6)-3,(7..8)-4]),
          node(1, B,[(1..1)-5]),
          node(2, B,[(1..1)-6]),
          node(3, B,[(2..2)-5]),
          node(4, B,[(2..2)-7]),
          node(5, C,[(10..10)]),
          node(6, C,[(20..20)]),
          node(7, C,[(30..30)])]).
```

```
| ?- elts(X, Y, Z).
X in 1..8,
Y in 1..2,
Z in {10}\/{20}\/{30}

| ?- elts(X, Y, Z), Z #>= 15.
X in(3..4)\/(7..8),
Y in 1..2,
Z in {20}\/{30}

| ?- elts(X, Y, Z), Y = 1.
Y = 1,
X in 1..4,
Z in {10}\/{20}
```

Circuit

- ***circuit(+Succ)***
- ***circuit(+Succ, +Pred)***
  - ***Succ*** é uma lista de comprimento n de variáveis de domínio ou inteiros
  - O i-ésimo elemento de ***Succ*** (***Pred***) é o sucessor (predecessor) de i no grafo
  - Verdadeiro se os valores formam um circuito Hamiltoniano
    - Nós estão numerados de 1 a n, o circuito começa no nó 1, visita cada um dos nós e regressa à origem
  - Exemplos:

```
| ?- length(L,5), domain(L,1,5), circuit(L).
L = [ _A,_B,_C,_D,_E ],
_A in 2..5, _B in {1}\/(3..5), _C in
(1..2)\/(4..5), _D in (1..3)\/{5}, _E in 1..4 ?
yes
```

```
| ?- length(L,5),
domain(L,1,5), circuit(L),
labeling([],L).
L = [2,3,4,5,1] ? ;
L = [2,3,5,1,4] ? ;
…
```