4.2.3. Négation

L'*hypothèse du monde clos* consiste à supposer que tout ce qui n'est pas connu comme vrai est considéré comme faux.

En programmation logique, cette hypothèse est adoptée. Ainsi, tout ce qui **n'est pas démontrable** à partir des prédicats du programme est **considéré comme faux**.

Exemple

Soit le programme

```
couleur(vert).
couleur(rouge).
```

et la requête

```
?- couleur(jaune).
false.
```

Cette requête est considérée comme **fausse** car elle est **indémontrable** par les clauses contenues dans le programme.

Attention : si une requête n'admet pas de solution, il est incorrect d'affirmer que sa négation admet une solution.

L'absence de démonstration d'une requête n'équivaut pas à l'existence d'une démonstration de sa négation.

L'**opérateur de négation par l'échec** est `\+`.

Exemple

Soient les trois prédicats

```
couleur(vert).
couleur(jaune).
couleur(rouge).
couleur(bleu).
```

```
claire(vert).
claire(jaune).
```

```
foncee(C) :-
    couleur(C),
    \+ claire(C).
```

Le prédicat `foncee/1` est tel que `foncee(C)` si `C` est une couleur telle que `claire(C)` échoue. Ainsi,

```
?- foncee(C).
C = rouge ;
C = bleu ;
false.
```

Exercice : développer l'arbre de résolution de la requête de cet exemple.

Il se construit de la même façon que les précédents, à la différence qu'un nœud de la forme `\+ q`, où `q` est un but, produit une solution si et seulement si `q` ne produit aucune solution.

L'**opérateur de différence** `\=` est la négation de l'unification. Ceci signifie que pour tous termes `t1` et `t2`, `t1 \= t2` si et seulement si `\+ (t1 = t2)`. Nous avons donc `t1 \= t2` si et seulement si `t1` et `t2` ne sont pas unifiables.

La négation par l'échec (incluant `\+` et `\=`) ne doit être employée que sur des termes dont **toutes les variables sont spécialisées**.

Exemple

Considérons le programme formé par l'unique prédicat `p(a).`.
Considérons les trois requêtes

```
?- p(X).
X = a.
```

```
?- X = b, p(X).
false.
```

```
?- X = b, \+ p(X).
X = b.
```

```
?- \+ p(X), X = b.
false.
```

- La 1re est immédiate : `X = a` est la seule solution de `p(X)`.
- La 2e est immédiate aussi : cela revient à regarder `p(b)` qui ne peut pas être démontrée.
- La 3e regarde si `p(b)` ne peut pas être démontrée, ce qui est bien le cas.
- La 4e, obtenue en transposant les deux sous-buts de la requête précédente, est surprenante. Comme la requête commence par `\+ p(X)` et que la variable `X` n'est à ce stade pas spécialisée, et comme `p(X)` admet une solution $\{X \mapsto a\}$, sa négation `\+ p(X)` échoue. Ainsi, la requête toute entière échoue.

4.3. Nombres et listes

4.3.1. Listes

PROLOG possède une **syntaxe dédiée** pour les **listes**.

Voici les principales constructions :

- la **liste vide** est noté `[]` ;
- la liste contenant les termes `t1`, ..., `tn` est notée `[t1, ..., tn]` ;
- si `t` est un terme et `lst` est une liste, alors `[t | lst]` est la liste dont la **tête** est `t` et la **queue** est `lst`.

  Cette notation s'étend de la façon suivante : si `t1`, ..., `tn` sont des termes et `lst` est une liste, alors `[t1, ..., tn | lst]` est la liste dont les éléments sont, dans l'ordre, `t1`, ..., `tn` puis ceux de `lst`.

Exemples

Voici quelques tentatives d'unification de listes :

```
?- [b, a, X] = [X, a, Y].
X = Y, Y = b.
```

```
?- [a, b, c] = [X | Y].
X = a, Y = [b, c].
```

```
?- [X, b, X] = [a, b, c].
false.
```

```
?- [a, a, X | [c, b]] = [Y | Z].
Y = a, Z = [a, X, c, b].
```

Soit `membre/2` le prédicat tel que `membre(X, LST)` si `X` est un élément de la liste `LST`.

Il se définit par

```
membre(X, [X | _]).
membre(X, [_ | LST]) :-
    membre(X, LST).
```

Exemples

```
?- membre(a, [a, b, c]).
true ;
false.
```

```
?- membre(a, [b, a, c]).
true ;
false.
```

```
?- membre(a, [b, b, c]).
false.
```

```
?- membre(X, [b, b, c]).
X = b ;
X = b ;
X = c ;
false.
```

```
?- membre(X, [b, a, Y]).
X = b ;
X = a ;
X = Y ;
false.
```

```
?- membre(X, Y).
Y = [X|_] ;
Y = [_, X|_] ;
Y = [_, _, X|_] ;
Y = [_, _, _, X|_] ;
% ...
```

Il s'agit du prédicat `member/2` prédéfini.

Soit `tete/2` le prédicat tel que `tete(X, LST)` si `X` **apparaît en tête** de la liste non vide `LST`. Il se définit par

```
tete(X, [X | _]).
```

Soit `dernier/2` le prédicat tel que `dernier(X, LST)` si `X` **apparaît comme dernier élément** de la liste non vide `LST`. Il se définit par

```
dernier(X, [X]).
dernier(X, [_ | LST]) :-
    dernier(X, LST).
```

Exemples

```
?- tete(X, Y).
Y = [X|_].
```

```
?- dernier(X, [a, b, a, b]).
X = b ;
false.
```

```
?- dernier(a, L).
L = [a] ;
L = [_, a] ;
L = [_, _, a] ;
L = [_, _, _, a] ;
% ...
```

Soit `concatenation/3` le prédicat tel que `concatenation(LST_1, LST_2, LST_3)` si la liste `LST_3` est la concaténation des deux listes `LST_1` et `LST_2`. Il se définit par

```
concatenation([], LST, LST).
concatenation([X | LST_1], LST_2, [X | LST_3]) :-
    concatenation(LST_1, LST_2, LST_3).
```

Exemples

```
?- concatenation([a, b, c], [d, c], L).
L = [a, b, c, d, c].
```

```
?- concatenation([d, c], L, [d, c, b, b]).
L = [b, b].
```

```
?- concatenation(L1, [d, c], L2).
L1 = [], L2 = [d, c] ;
L1 = [_A], L2 = [_A, d, c] ;
L1 = [_A, _B], L2 = [_A, _B, d, c] ;
L1 = [_A, _B, _C], L2 = [_A, _B, _C, d, c] ;
% ...
```

```
?- concatenation(L, [d, c], [a, d, c]).
L = [a] .
```

```
?- concatenation([c], L, [d, c]).
false.
```

```
?- concatenation(L1, L2, [b, c, a, a]).
L1 = [], L2 = [b, c, a, a] ;
L1 = [b], L2 = [c, a, a] ;
L1 = [b, c], L2 = [a, a] ;
L1 = [b, c, a], L2 = [a] ;
L1 = [b, c, a, a], L2 = [] ;
false.
```

Il s'agit du prédicat `append/3` prédéfini.

Soit `prefixe/2` le prédicat tel que `prefixe(LST_1, LST_2)` si la liste `LST_1` est un préfixe de la liste `LST_2`. Il se définit par

```
prefixe(LST_1, LST_2) :-
    concatenation(LST_1, _, LST_2).
```

Soit `suffixe/2` le prédicat tel que `suffixe(LST_1, LST_2)` si la liste `LST_1` est un suffixe de la liste `LST_2`. Il se définit par

```
suffixe(LST_1, LST_2) :-
    concatenation(_, LST_1, LST_2).
```

Ils se basent sur l'idée que `LST_1` est un préfixe (resp. suffixe) de `LST_2` s'il existe une liste `LST_3` telle que `LST_2` est la concaténation de `LST_1` et `LST_3` (resp. `LST_3` et `LST_1`).

Exemples

```
?- prefixe(L, [a, b, c]).
L = [] ;
L = [a] ;
L = [a, b] ;
L = [a, b, c] ;
false.
```

```
?- suffixe(L, [a, b, c]).
L = [a, b, c] ;
L = [b, c] ;
L = [c] ;
L = [] ;
false.
```

Soit `facteur/2` le prédicat tel que `facteur(LST_1, LST_2)` si la liste `LST_1` **est un facteur** de la liste `LST_2`. Il se définit par

```
facteur(LST_1, LST_2) :-
    suffixe(LST_3, LST_2),
    prefixe(LST_1, LST_3).
```

Il se base sur l'idée que `LST_1` est un facteur de `LST_2` si `LST_1` est un préfixe d'un suffixe de `LST_2`.

Exemples

```
?- facteur([a, b], [c, a, b, b]).
true ;
false.
```

```
?- facteur([a, b], [c, a, c, b]).
false.
```

```
?- facteur(L, [a, b, b]).
L = [] ;
L = [a] ;
L = [a, b] ;
L = [a, b, b] ;
L = [] ;
```

```
L = [b] ;
L = [b, b] ;
L = [] ;
L = [b] ;
L = [] ;
false.
```

```
?- facteur([b, a], L).
L = [b, a|_] ;
L = [_, b, a|_] ;
L = [_, _, b, a|_] ;
L = [_, _, _, b, a|_] ;
L = [_, _, _, _, b, a|_] ;
% ...
```

L'idée précédente énonçant que `LST_1` est un facteur de `LST_2` si `LST_1` est un préfixe d'un suffixe de `LST_2` admet naturellement la **variante** suivante.

La liste `LST_1` est un facteur de `LST_2` si `LST_1` est un suffixe d'un prefixe de `LST_2`.

Cela nous permet de définir le prédicat `facteur2/2` tel que `facteur2(LST_1, LST_2)` si la liste `LST_1` **est un facteur** de la liste `LST_2` par

```
facteur2(LST_1, LST_2) :-
    prefixe(LST_1, LST_3),
    suffixe(LST_3, LST_2).
```

Cependant, ce prédicat n'est pas effectivement équivalent au précédent. Ceci est illustré par la requête ci-contre.

```
?- facteur2(L, [a, b]).
L = [] ;
L = [] ;
L = [] ;
L = [a] ;
L = [b] ;
L = [a, b] ;
% Résolution infinie.
```

Sa résolution s'engouffre dans une branche qui mène à un arbre de résolution infini, sans produire de nouvelles solutions.

La partie infinie de l'arbre commence au moment où un but `concatenate(L1, L2, L3)` est considéré.

Exercice : dessiner le début de l'arbre de résolution de la requête plus simple `facteur2(L, []).` qui produit déjà un arbre de résolution infini après avoir trouvé la solution `L = []`. Comparer avec la résolution de `facteur(L, []).`.

Soit `sous_liste/2` le prédicat tel que `sous_liste(LST_1, LST_2)` si la liste `LST_1` est une sous-liste de la liste `LST_2`. Il se définit par

```
sous_liste([], _).
sous_liste([X | LST_1], [X | LST_2]) :-
    sous_liste(LST_1, LST_2).
sous_liste(LST_1, [_ | LST_2]) :-
    sous_liste(LST_1, LST_2).
```

Exemples

```
?- sous_liste([a, c], [a, b, c]).
true ;
false.
```

```
?- sous_liste([c, a], [a, b, c]).
false.
```

```
?- sous_liste(L, [a, b, c]).
L = [] ;
L = [a] ;
L = [a, b] ;
L = [a, b, c] ;
L = [a, b] ;
L = [a] ;
L = [a, c] ;
L = [a] ;
```

```
L = [] ;
L = [b] ;
L = [b, c] ;
L = [b] ;
L = [] ;
L = [c] ;
L = [] ;
false.
```

```
?- sous_liste([a, b], L).
L = [a, b|_] ;
L = [a, b, _|_] ;
L = [a, b, _, _|_] ;
L = [a, b, _, _, _|_] ;
% ...
```

Soit `miroir/2` le prédicat tel que `miroir(LST_1, LST_2)` si la liste `LST_1` **est la liste miroir** de la liste `LST_2`. Il se définit par

```
miroir([], []).
miroir([X | LST_1], LST_2) :-
    miroir(LST_1, LST_3),
    concatenation(LST_3, [X], LST_2).
```

Il se base sur l'idée que la liste miroir d'une liste non vide peut s'exprimer en considérant l'image miroir de sa queue à laquelle est ajoutée en dernière position la tête de la liste originale.

Exemples

```
?- miroir([a, b, c], L).
L = [c, b, a].
```

```
?- miroir(L, [a, b, c]).
L = [c, b, a] ;
% Résolution infinie.
```

Exercice : dessiner le début de l'arbre de résolution de la requête `miroir(L, [a, b, c]).` pour observer comment l'arbre infini se développe après que la solution a été trouvée.

Considérons une version améliorée avec un paramètre jouant le rôle d'un accumulateur.

Voici un prédicat ternaire

```
miroir2_acc([], LST, LST).
miroir2_acc([X | LST_1], LST_2, LST_3) :-
    miroir2_acc(LST_1, [X | LST_2], LST_3).
```

et le prédicat binaire enrobant

```
miroir2(LST_1, LST_2) :-
    miroir2_acc(LST_1, [], LST_2).
```

Nous pouvons comparer les performances relatives des prédicats `miroir/2` par `miroir2/2` à l'aide du prédicat `time/1` qui est tel que `time(G)` résout le but `G` et imprime des informations sur sa résolution.

Exemples

```
?- time(miroir([a, b, c, d], L)).
 % 13 inferences, 0.000 CPU in 0.000 seconds
 (70% CPU, 622039 Lips)
L = [d, c, b, a].
```

```
?- time(miroir([a, b, c, d, e], L)).
 % 19 inferences, 0.000 CPU in 0.000 seconds
 (73% CPU, 978423 Lips)
L = [e, d, c, b, a].
```

```
?- time(miroir([a, b, c, d, e, f], L)).
 % 26 inferences, 0.000 CPU in 0.000 seconds
 (71% CPU, 1091749 Lips)
L = [f, e, d, c, b, a].
```

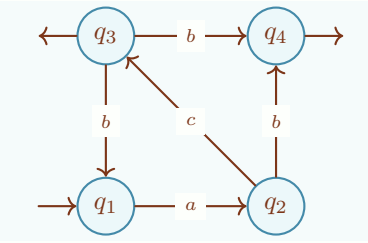
```
?- time(miroir2([a, b, c, d], L)).
 % 4 inferences, 0.000 CPU in 0.000 seconds
 (66% CPU, 211640 Lips)
L = [d, c, b, a].
```

```
?- time(miroir2([a, b, c, d, e], L)).
 % 5 inferences, 0.000 CPU in 0.000 seconds
 (74% CPU, 299724 Lips)
L = [e, d, c, b, a].
```

```
?- time(miroir2([a, b, c, d, e, f], L)).
 % 6 inferences, 0.000 CPU in 0.000 seconds
 (76% CPU, 263389 Lips)
L = [f, e, d, c, b, a].
```

Voici un autre exemple de manipulation de listes basé sur les mots reconnus par un **automate**.

Considérons l'automate



Le nœud (appelé `état`) q_1 est **initial**. Ceci est mentionné par la flèche entrante.

Les états q_3 et q_4 sont finaux. Ceci est mentionné par les flèches sortantes.

Les autres flèches (appelées `transitions`) sont décorées par des lettres.

Un mot u est *reconnu* par un automate s'il existe un chemin partant d'un état initial et arrivant dans un état final dont les décorations des transitions empruntées forment le mot u.

Par exemple, les mots ab, ac et $acbab$ sont reconnus par cet automate. À l'inverse, les mots a et abc ne sont pas reconnus par ce dernier.

Notre problématique est de représenter cet automate et de définir un prédicat permettant de vérifier (et donc, d'engendrer) les mots qu'il reconnaît.

Il faut tout d'abord représenter l'automate précédent.

Un automate étant caractérisé par l'information des états qu'il contient, de quels états sont initiaux ou terminaux et par ses transitions, nous fournissons les prédicats suivants :

```
etat(q1).
etat(q2).
etat(q3).
etat(q4).
```

```
initial(q1).

final(q3).
final(q4).
```

```
transition(q1, a, q2).
transition(q2, b, q4).
transition(q2, c, q3).
transition(q3, b, q1).
transition(q3, b, q4).
```

```
reconnu(MOT) :-
    initial(Q),
    reconnu(MOT, Q).
```

```
reconnu([], Q) :-
    final(Q).
reconnu([A | MOT], Q) :-
    etat(Q2),
    transition(Q, A, Q2),
    reconnu(MOT, Q2).
```

Notons que les prédicat `reconnu/1` et `reconnu/2`, du même nom mais ayant des arités différentes, sont différents.

Voici quelques requêtes :

```
?- reconnu([]).
false.
```

```
?- reconnu([a, b]).
true ;
false.
```

```
?- reconnu([a, c, b]).
true .
```

```
?- reconnu(MOT).
MOT = [a, c] ;
MOT = [a, c, b, a, c] ;
MOT = [a, c, b, a, c, b, a, c] ;
MOT = [a, c, b, a, c, b, a, c, b|...] ;
% ...
```

Voici quelques autres prédicats prédéfinis supplémentaires sur les listes.

- `append/2` tel que `append(LST_1, LST_2)` si l'aplatissement de la liste `LST_1` est la liste `LST_2`.

Exemple

```
?- append([[a], [b, c]], [a, b, c]).
true.
```

- `same_length/2` tel que `append(LST_1, LST_2)` si les listes `LST_1` et `LST_2` sont de la même longueur.

Exemple

```
?- same_length([a, b], [c, a]).
true.
```

- `select/3` tel que `select(X, LST_1, LST_2)` si la liste `LST_2` est obtenue en supprimant une occurrence de `X` dans la liste `LST_1`.

Exemple

```
?- select(a, [b, a, c, a], L).
L = [b, c, a] ;
L = [b, a, c].
```

Si `LST` est une liste, notons par `LST[`i`]` son i^{e} élément (commençant ici par convention à l'indice 1).

Pour tout $n \geqslant 1$, le prédicat prédéfini `maplist/`$n+1$ est tel que `maplist(P, LST_1, ..., LST_`n`)` si `P` est un prédicat d'arité n et `LST_1`, ..., `LST_`n sont des listes de longueur $k \geqslant 0$ telles que pour tout $1 \leqslant i \leqslant k$, `P(LST_1[`$i$`], ..., LST_`$n$`[`$i$`])`. Dans SWI-Prolog, ces prédicats sont définis pour tout $1 \leqslant n \leqslant 4$.

Exemples

En reprenant les entiers de Peano vu précédemment, nous pouvons écrire les requêtes suivantes.

- Pour $n = 2$:

```
?- maplist(fact, [s(z), z, s(s(s(z)))], L).
L = [s(z), s(z), s(s(s(s(s(z))))))].
```

- Pour $n = 3$:

```
?- maplist(add, [z, z, s(s(z))], [s(z), s(z), z], L).
L = [s(z), s(z), s(s(z))].
```

- Pour $n = 1$:

```
?- maplist(entier, [z, s(z), miaou]).
false.
```

Remarque : le prédicat `maplist/1` est tel que `maplist(P, LST)` si `P(X)` pour tout élément `X` de `LST`. Il permet donc d'effectuer un **test universel** (analogue au `List.for_all` d'OCaml).

4.3.2. Nombres et contraintes arithmétiques

Pour le moment, nous avons évité de travailler avec les **entiers natifs** de PROLOG. Ils existent bien mais sont limités dans la façon dont ils se comportent avec l'algorithme de résolution.

Nous allons utiliser ici la bibliothèque `clpfd` (*Constraint Logic Programming over Finite Domains*) pour manipuler les entiers et la plupart des opérations et relations sur ces derniers.

L'inclusion de cette bibliothèque se fait par

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

Il ne faut pas oublier le `:-` lorsque l'inclusion s'effectue depuis un fichier `.pl`.

La *programmation par contrainte* est une branche de la programmation logique dans laquelle un problème est spécifié par des contraintes que doivent vérifier des objets comme des nombres ou des booléens.

Ici, nous allons considérer des problèmes sur les **entiers** soumis à des **contraintes arithmétiques**. La bibliothèque `clpfd` est parfaitement adaptée à ce contexte.

Une *expression arithmétique* est une expression pouvant prendre l'une des formes suivantes :

1. `C` ;
2. `X` ;
2. `-E` ;
3. `abs(E)` ;
4. `E1 + E2` ;
5. `E1 * E2` ;
6. `E1 - E2` ;
7. `E1 // E2` ;
8. `E1 ^ E2` ;
9. `min(E1, E2)` ;
10. `max(E1, E2)` ;
11. `mod(E1, E2)` ;

où `C` est une constante entière, `X` est une variable et `E`, `E1` et `E2` sont, récursivement, des expressions arithmétiques.

Leur sémantique est intuitive. En particulier, `^` est l'opération d'exponentiation et `//` est l'opération de division entière.

Exemple

```
1 // ((1 - X) ^ 2) + 1 // (1 - X * Y)
```

Une *contrainte arithmétique* est une expression pouvant l'une des formes suivantes :

1. `E1 #= E2` ;
2. `E1 #\= E2` ;
2. `E1 #>= E2` ;
3. `E1 #=< E2` ;
4. `E1 #> E2` ;
5. `E1 #< E2` ;

où `E1` et `E2` sont, récursivement, des expressions arithmétiques.

Leur sémantique est intuitive (attention néanmoins à la relation `#=<` notée de manière inhabituelle.

Exemple

```
X - Y #>= Z ^ 2 + 1
```

Nous pouvons utiliser l'approche par contrainte pour redéfinir de manière efficace les fonctions précédentes sur les entiers de Peano, en se servant des entiers manipulés via `clpfd`.

Exemple

```
fact(0, 1).
fact(N, F) :-
    N #> 0,
    N1 #= N - 1,
    F #= N * F1,
    fact(N1, F1).
```

```
?- fact(7, X).
X = 5040 ;
false.
```

```
?- fact(25, X).
X = 15511210043330985984000000 ;
false.
```

```
?- fact(X, 720).
X = 6 .
```

```
?- maplist(fact, [1, 2, 3, 4, 5, 6], L).
L = [1, 2, 6, 24, 120, 720] ;
false.
```

```
?- fact(X, X).
X = 1 ;
X = 2 ;
false.
```

```
?- X #=< 1024, fact(X, X * 2).
X = 3 ;
false.
```

Observons quand dans la toute dernière requête, la variante `fact(X, X * 2), X #=< 1024.` mène à une résolution infinie. Il est important de disposer les **contraintes les plus contraignantes en 1er** de sorte à **restreindre le plus possible l'espace de recherche**.

Exercice : implanter le prédicat `fibo/2` de manière similaire en utilisant `clpfd` et expérimenter.

Le prédicat infixe `in/2` permet de spécifier un contrainte par un **intervalle d'entiers**. Plus précisément,

```
X in A..B
```

exprime le fait que la variable `X` doit être supérieure à l'entier `A` et inférieure à l'entier `B`.

Exemples

```
?- X in 1..3, Y in 2..4.
X in 1..3, Y in 2..4.
```

```
?- X in 0..10, Y in 0..10, X #=< Y.
X in 0..10, Y#>=X, Y in 0..10.
```

Le prédicat `label/1` prend une liste de variables contraintes et **recherche explicitement leurs valeurs**.

Exemples

```
?- X in 1..2, Y in 2..3, label([X, Y]).
X = 1, Y = 2 ;
X = 1, Y = 3 ;
X = Y, Y = 2 ;
X = 2, Y = 3.
```

```
?- X in 0..2, Y in 0..2, X #=< Y, label([X, Y]).
X = Y, Y = 0 ;
X = 0, Y = 1 ;
X = 0, Y = 2 ;
X = Y, Y = 1 ;
X = 1, Y = 2 ;
X = Y, Y = 2.
```

Le prédicat infixe `ins/2` permet de spécifier un contrainte par un **intervalle d'entiers** sur toutes les **variables d'une liste**. Plus précisément,

```
LST ins A..B
```

exprime le fait que toutes les variables de la liste de variables `LST` doivent être supérieures à l'entier `A` et inférieures à l'entier `B`.

Le prédicat `all_distinct/1` permet de spécifier le fait que toutes les variables d'une liste doivent prendre des **valeurs distinctes**. Plus précisément,

```
all_distinct(LST)
```

exprime le fait que pour toutes variables `X1` et `X2` différentes, toute spécialisation de `X1` doit être différente de toute spécialisation de `X2`.

Exemple

```
?- [X, Y, Z] ins 0..1, all_distinct([X, Z]), label([X, Y, Z]).
X = Y, Y = 0, Z = 1 ;
X = 0, Y = Z, Z = 1 ;
X = 1, Y = Z, Z = 0 ;
X = Y, Y = 1, Z = 0.
```

Comme nous l'avons vu dans les nombreux exemples parcourus jusqu'à présent, si une requête admet plusieurs solutions, celles-ci sont calculées et imprimées une à une.

Il est cependant utile de pouvoir **calculer toutes les solutions directement**. Une bonne manière de faire consiste à construire une liste contenant toutes les substitutions vérifiant les contraintes. Pour cela, nous utilisons le prédicat `findall/3` tel que

```
findall(LST_VARS, C, RES)
```

si `LST_VARS` est une liste de variables, `C` une contrainte mettant en jeu des variables de `LST_VARS` et `RES` est la liste dont ses éléments sont des listes assignant des valeurs aux variables de `LST_VARS` qui rendent vraie la contrainte `C`.

Ceci est applicable dans le contexte des entiers mais pas uniquement.

Exemples

```
?- findall([X, Y], ([X, Y] ins -3..3, X + Y #= 0, label([X, Y])), R).
R = [[-3, 3], [-2, 2], [-1, 1], [0, 0], [1, -1], [2, -2], [3, -3]].
```

En reprenant l'exemple généalogique :

```
?- findall([X, Y], grand_parent(X, Y), R).
R = [[marvin, shelly], [marvin, stan], [grandma, stan], [grandma, shelly]].
```

Dans certains cas, suivant comment un prédicat est écrit et la façon dont l'algorithme de résolution travaille, plusieurs solutions identiques peuvent être collectées par `findall/3`.

Exemple

```
?- findall([X, Y], fratrie(X, Y), R).
R = [[randy, jimbo], [jimbo, randy], [shelly, stan], [stan, shelly], [stan, shelly], [shelly, stan]].
```

Il devient ainsi utile de **supprimer les doublons** dans une liste. Pour cela, nous utilisons le prédicat `sort/2` tel que `sort(LST_1, LST_2)` si la liste `LST_2` est la version triée et sans doublon de la liste `LST_1`.

Exemple

```
?- findall([X, Y], fratrie(X, Y), R1), sort(R1, R).
R1 = [[randy, jimbo], [jimbo, randy], [shelly, stan], [stan, shelly], [stan, shelly], [shelly, stan]],
R = [[jimbo, randy], [randy, jimbo], [shelly, stan], [stan, shelly]].
```

4.3.3. Exemples

Commençons par le cryptarithme

```
  S E N D
+ M O R E
----------
M O N E Y
```

qui consiste à rendre le calcul valide où chaque lettre désigne un chiffre différent.

Nous pouvons proposer le prédicat suivant pour le résoudre :

```
puzzle(S, E, N, D, M, O, R, Y) :-
    Vars = [S, E, N, D, M, O, R, Y],
    Vars ins 0..9,
    all_distinct(Vars),
                  S * 1000 + E * 100 + N * 10 + D
                + M * 1000 + O * 100 + R * 10 + E
    #= M * 10000 + O * 1000 + N * 100 + E * 10 + Y,
    M #\= 0,
    S #\= 0,
    label(Vars).
```

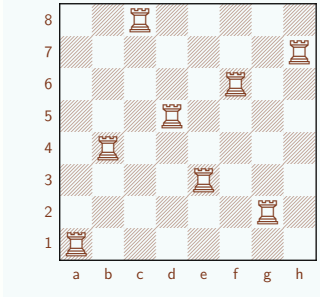
```
?- puzzle(S, E, N, D, M, O, R, Y).
S = 9, E = 5, N = 6, D = 7, M = 1, O = 0, R = 8, Y = 2 ;
false.
```

Il est possible d'utiliser la programmation par contrainte pour construire des nouveaux cryptarithmes avec des règles similaires (où une seule solution est possible). Mais cela est une autre histoire...

Exemple

Un *placement de tours* sur un échiquier de dimension n consiste à positionner n tours blanches de sorte qu'aucune n'en protège une autre.

L'exemple ci-contre montre une solution possible lorsque $n = 8$.



Nous cherchons un moyen de définir un prédicat qui calcule toutes les solutions étant donné n.

Comme d'habitude ici, le problème devient très simple une fois qu'il est bien modélisé.

Ici, nous pouvons remarquer qu'une solution est spécifiée entièrement par les indices des rangées de chacune des tours de la gauche vers la droite. Ceci est rendu possible par le fait qu'étant donnée une rangée, il y a exactement une tour qui l'occupe. La solution de l'exemple est ainsi représentée par la liste `[1, 4, 8, 5, 3, 6, 2, 7]`.

Nous avons ramené le problème initial à un problème spécifié par des contraintes. Une liste `R` est une réponse valide pour le problème des n tours si

- la longueur de `R` est n ;
- les éléments de `R` sont compris entre 1 et n ;
- les éléments de `R` sont tous distincts.

Ceci nous mène au prédicat `placement_tours/2` qui est tel que `placement_tours(N, R)` si la liste `R` est une solution au problème des `N` tours. Il est défini par

```
placement_tours(N, R) :-
    length(R, N),
    R ins 1..N,
    all_distinct(R),
    label(R).
```

```
?- placement_tours(3, R).
R = [1, 2, 3] ;
R = [1, 3, 2] ;
R = [2, 1, 3] ;
R = [2, 3, 1] ;
R = [3, 1, 2] ;
R = [3, 2, 1].
```

Voici un prédicat pour calculer toutes les solutions

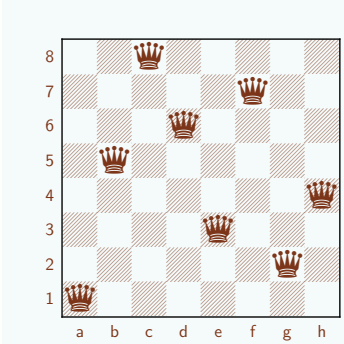
```
placement_tours_toutes(N, SOL) :-
    findall([R], placement_tours(N, R), SOL).
```

```
?- placement_tours_toutes(2, R).
R = [[[1, 2]], [[2, 1]]].
```

Exercice : changer un peu l'ordre des conditions définissant `placement_tours` et observer puis expliquer ce qu'il se passe.

Exercice : continuer ceci afin de dénombrer les solutions au problème en fonction de n.

Exemple



Un *placement de dames* sur un échiquier de dimension n consiste à positionner n dames noires de sorte qu'aucune n'en protège une autre.

L'exemple ci-contre montre une solution possible lorsque $n = 8$.

Nous cherchons un moyen de définir un prédicat qui calcule toutes les solutions étant donné n.

Tout comme pour le problème des n tours, nous allons coder une solution par la liste des rangées occupées par les dames, de la gauche vers la droite. Ainsi, la solution de l'exemple précédent est codée par la liste `[1, 5, 8, 6, 3, 7, 2, 4]`.

Soit le prédicat `independante/3` tel que `independante(LST, D, DIST)` si une dame placée sur la 1re colonne et en rangée `D` ne protège aucune des dames disposées à partir de la 2e colonne sur les rangées spécifiées par la liste `LST` avec `DIST = 1`. Il se définit par

```
independante([], _, _).
independante([D | DAMES], DAME, DIST) :-
    DAME #\= D,
    DAME #\= D + DIST,
    DAME #\= D - DIST,
    DIST_2 #= DIST + 1,
    independante(DAMES, DAME, DIST_2).
```

Exemples

- ☐
```
?- independante([4, 2, 5], 1, 1).
true.
```
- ☐
```
?- independante([4, 2, 5], 2, 1).
false.
% Protection orthogonale de la dame 2.
```
- ☐
```
?- independante([4, 2, 5], 3, 1).
false.
% Protection diagonale de la dame 4.
```
- ☐
```
?- independante([4, 2, 5], 4, 1).
false.
% Protection orthogonale de la dame 4.
```
- ☐
```
?- independante([4, 2, 5], 5, 1).
false.
% Protection orthogonale de la dame 5.
```
- ☐
```
?- independante([4, 2, 5], 6, 1).
true.
```
- ☐
```
?- independante([4, 2, 5], 7, 1).
true.
```
- ☐
```
?- independante([4, 2, 5], 8, 1).
false.
% Protection diagonale de la dame 5.
```

Soit le prédicat `independantes/1` tel que `independantes(DAMES)` si aucune dame de la liste de dames spécifiées par la liste `DAMES` n'en protège une autre. Il se définit par

```
independantes([]).
independantes([D | DAMES]) :-
    independante(DAMES, D, 1),
    independantes(DAMES).
```

Soit finalement le prédicat `placement_dames/2` tel que `placement_dames(N, R)` si la liste `R` est une solution au problème des `N` dames. Il est defini par

```
placement_dames(N, DAMES) :-
    length(DAMES, N),
    DAMES ins 1..N,
    all_distinct(DAMES),
    independantes(DAMES),
    label(DAMES).
```

Nous avons ainsi par exemple,

```
?- placement_dames(8, D).
D = [1, 5, 8, 6, 3, 7, 2, 4] ;
D = [1, 6, 8, 3, 7, 4, 2, 5] ;
D = [1, 7, 4, 6, 8, 2, 5, 3] ;
D = [1, 7, 5, 8, 2, 4, 6, 3] ;
% ...
```

```
?- findall([R], placement_dames(8, R), L), length(L, LEN).
L = [[[1, 5, 8, 6, 3, 7, 2|...]], [[1, 6, 8, 3, 7, 4|...]], % ...
LEN = 92.
```

INF6120 --- Programmation fonctionnelle et logique

Samuele Giraudo

Département d'informatique, Université du Québec à Montréal

giraudo.samuele@uqam.ca

Baccalauréat en informatique et génie logiciel

Automne 2024