Introduction à gprolog

Michel Lévy

11 mai 2007

Table des matières

Introduction

Prolog est lié de manière non exhaustive
- à une modélisation du raisonnement
- au traitement de la langue
- aux bases de données

1 Faits, Règles et Requêtes

1.1 Programme

Un programme Prolog est une liste de clauses. Chaque clause d'un programme est suivie par un point.
Une clause est un fait ou une règle.

Exemple 1.1
`homme(socrate).`
est un fait qui exprime que «Socrate est un homme».
`mortel(X):-homme(X).`
est une règle qui exprime que tout homme est mortel, autrement dit pour tout `X` *X est mortel* si `X` *est un homme.*
Mon premier programme est

```
homme(socrate).
mortel(X):-homme(X).
```

1.2 Question

Pour utiliser des programmes, on pose des questions (encore appelées requêtes ou buts).
Ma première question (suivie de sa réponse) est :
`?-mortel(socrate).` Autrement dit «socrate est-il mortel ?».
`yes`
Ma deuxième question (suivie de sa réponse) est :
`?-mortel(X).` Autrement dit «qui est mortel ?».
```
X = socrate
yes
```
`X = socrate` est une substitution, définissant la valeur de `X`. On note que `mortel(socrate)` est une conséquence du programme.
Prolog trouve les valeurs des variables, qui rendent la question conséquence du programme.

1.3 Logique et Prolog

Notre deuxième programme est :

```
% socrate est un homme
homme(socrate).
homme(platon).
homme(levy).
% socrate est philosophe
philosophe(socrate).
philosophe(platon).
% tout homme philosophe est un sage
sage(X):-homme(X),philosophe(X).
% tout homme est mortel
```

```
mortel(X):-homme(X).
% tout animal est mortel
mortel(X):-animal(X).
% medor est un animal
animal(medor).
animal(minou).
```

1.3.1 Variable, ordre des réponses

Dans une ligne, tout ce qui suit le caractère % est un commentaire. On recommande de commenter les programmes. Posons des questions et observons les réponses.

```
   ?-homme(levy).
yes
   ?-homme(chirac).
no
```

Cela ne veut pas dire que Mr Chirac n'est pas un homme, mais qu'on ne peut pas le déduire du programme ci-dessus. Notez que dans la question, on a noté `chirac` avec une minuscule. En effet en Prolog :
une variable est un mot qui commence par une majuscule.

```
?-homme(Personne).
Personne = socrate ?;
Personne = platon ?;
Personne = levy
yes
```

Prolog donne une première réponse `Personne = socrate ?`, l'utilisateur tape un point-virgule pour obtenir la réponse suivante. Il n'y a pas de point-virgule après la troisième réponse, car Prolog détermine qu'il n'y a pas d'autre réponse.

Pour répondre à cette question, Prolog parcourt les règles du programme commençant par le prédicat homme, **dans l'ordre où ces règles figurent dans le programme**.

1.3.2 Conjonction, Instanciation et Modus ponens

Posons la question et observons-en les réponses

```
?-sage(X).
X = socrate ? ;
X = platon ? ;
no
```

Tout d'abord, on note que Prolog ne sait pas, après avoir fournit la deuxième réponse, que c'est la dernière réponse.

Prolog utilise la règle commençant par sage, et réduit le but `sage(X)` au but `homme(X), philosophe(X)`. Puis il détermine que l'on peut choisir `X = socrate` car socrate est à la fois un homme et un philosophe et en conclut que socrate est un sage. Il fait de même avec `X = platon`. Mais il est clair que `levy` n'est pas un sage, car ce programme ne permet pas de déduire que levy est philosophe.

Quels sont les lois logiques à l'oeuvre dans la déduction de la première réponse ?

L'introduction de la conjonction (la conjonction est notée en prolog par une virgule) :

$$\frac{homme(socrate) \quad philosophe(socrate)}{homme(socrate), philosophe(socrate)} \wedge I$$

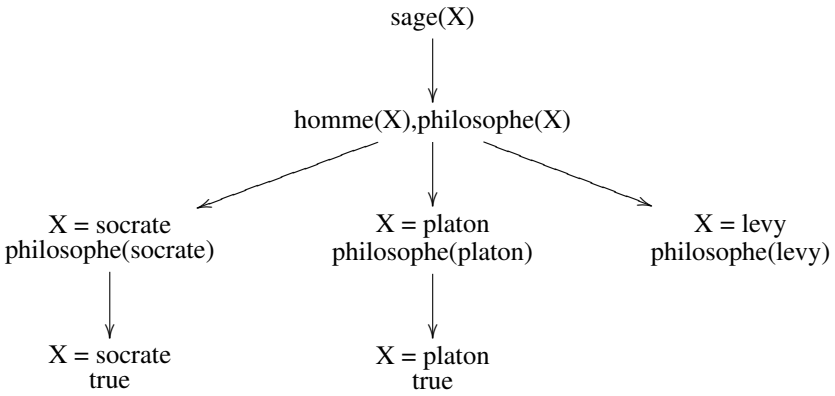
L'élimination d'un «pour tout» (mais le «pour tout» est implicite en Prolog) :

$$\frac{\forall X(sage(X) :- homme(X), philosophe(X))}{sage(socrate) :- homme(socrate), philosophe(socrate)} \forall E$$

Le modus ponens, appelée aussi élimination de l'implication (en Prolog l'implication est notée à l'envers par `:-`)

$$\frac{sage(socrate) :- homme(socrate), philosophe(socrate) \quad homme(socrate), philosophe(socrate)}{sage(socrate)} \Rightarrow E$$

En fait Prolog n'utilise que ces trois lois logiques. La recherche des réponses est représentée par un arbre d'exploration (search tree en anglais) et chaque réponse correspond à une preuve. On donne ici l'exploration complète correspondante aux réponses ci-dessus et les deux preuves correspondantes aux deux réponses.



La preuve correspondante à la réponse `X = socrate` est la combinaison en un arbre de preuve des règles logiques ci-dessus :

$$\frac{\dfrac{\forall X(sage(X) :- homme(X), philosophe(X))}{sage(socrate) :- homme(socrate), philosophe(socrate)} \forall E \quad \dfrac{homme(socrate) \quad philosophe(socrate)}{homme(socrate), philosophe(socrate)} \wedge I}{sage(socrate)} \Rightarrow E$$

Notez que chaque réponse de l'arbre d'exploration correspond à une preuve en déduction naturelle.

Dans la suite, la virgule se prononce «et», le signe `:-` se prononce «si», et les variables des règles sont implicitement universellement quantifiées, c'est-ainsi que l'on retrouve mieux le sens de la règle :

```
sage(X):-homme(X),philosophe(X)
```

en la prononçant :

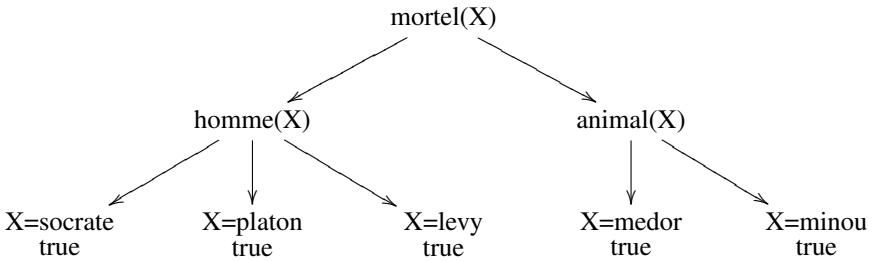
```
pour tout X (sage(X) si homme(X) et philosophe(X))
```

1.3.3 Disjonction

Ce qui est à gauche du si dans une règle est la **tête de la règle**, ce qui est à droite est le **corps de la règle**.

La disjonction est exprimée par la présence de deux règles avec une même tête de règle. En revenant à l'exemple pour prouver que X est mortel, on a deux preuves possibles, prouver que X est un homme ou prouver que X est un animal.

On donne l'arbre d'exploration de la question `mortel(X)` qui met en évidence l'expression de la disjonction par le choix de plusieurs règles applicables à un but :



Les réponses founies par Prolog à la question ci-dessus sont obtenues dans l'ordre gauche-droite des feuilles avec réponse de cet arbre :

```
?-mortel(X).
X = socrate ? ;
X = platon ? ;
X = levy ? ;
X = medor ? ;
X = minou
yes
```

Comme on l'a déjà Prolog parcourt les règles du programme commençant par le prédicat mortel (respectivement homme, animal) **dans l'ordre où ces règles figurent dans le programme**.

On donne la preuve associée à la réponse `X = medor`.

$$\dfrac{animal(medor) \quad \dfrac{\forall X(mortel(X) :- animal(X))}{mortel(medor) :- animal(medor)}\ \forall E}{mortel(medor)}\ \Rightarrow E$$

1.4 La syntaxe de Prolog

La *seule* structure de données de Prolog est celle de termes. Voici des exemples de termes

```
Personne
socrate
2
2+X
+(2,X)
homme(socrate)
mortel(X):-homme(X)
```

Il a quatre sortes de termes : les variables (Personne), les atomes (socrate) , les nombres (2) et les termes composés (les 4 derniers exemples ci-dessus).

On voit que les termes composés peuvent être écrits en notation infixée `2+X` ou préfixée `+(2,X)`.

Comme le montre des 2 derniers exemples, les faits, règles et requêtes sont des termes.

1.4.1 Les caractères

Pour écrire les programme, on dispose des ensembles suivants de caractères : les minuscules, les majuscules, les chiffres et les caractères spéciaux. Les caractères spéciaux sont :
`#, $, &, *, +, -, ., /, :, <, =, >, ?, @, ^, ~`.

1.4.2 Les variables

Une variable est une suite de majuscules, de minuscules, de chiffres et de blancs soulignés qui commence par une majuscule ou par un blanc souligné. Par exemple, `X, Y, Liste, _tag, X_67, Tete` sont des variables.

La variable _ est très particulière. Elle est appelée la variable anonyme.

1.4.3 Les atomes

Un atome est soit :

1. Une chaine de caractères faite de majuscules, de minuscules, de chiffres et du blanc souligné, qui commence par un *minuscule*. Par exemple : `marylin_monroe, joe`.
2. Une suite arbitraire de caractères entre des guillemets simples. Par exemple : `'Je suis un atome', 'l"arbre'`. Observez sur le deuxième exemple, comment mettre un guillemet simple dans un atome.

3. Une suite de caractères spéciaux. Par exemple `;, :-` sont des atomes qui ont un sens prédéfini.
   Attention : =–, =.' sont des atomes, donc si voulez utilisez le signe = comme atome, pensez à le faire précéder et suivre de caractères non spéciaux (dont lettres, chiffres, espace).

Notons que si un atome peut s'écrire sans guillemets simples, il est considéré comme identique au même atome entre guillemets simples : `abcd = 'abcd', <= = '<='`.

1.4.4 Les nombres

On peut utiliser les entiers et les flottants. `33, -33, 33.0, -0.33E+02` sont des exemples de nombres avec les notations usuelles.

Une constante (ou encore un terme atomique) est un atome ou un nombre.

1.4.5 Les termes composés

On utilise une notation standard pour rappeler ce qu'est un terme et définir un terme composé. Dans cette notation, les termes à définir sont des noms commençant par une majuscule, le signe `::=` signifie "est défini par", la barre verticale indique une alternative. Une expression entre crochets peut être omise ou présente, une expression entre accolades peut être répétée zéro ou plusieurs fois.

```
Terme ::= Atome | Variable | Nombre | Terme-composé
Terme-composé ::= Atome( Terme {, Terme})
```

L'atome qui débute un terme composé, est le *foncteur* du terme. Par exemple `aime` est le foncteur du terme composé `aime(vincent,marie)`.

L'arité d'un terme composé est le nombre d'arguments du terme. `femme(marie)` est un terme d'arité 1, `aime(vincent,marie)` est un terme d'arité 2.

Dans un programme Prolog, on peut avoir deux prédicats de même foncteur et d'arités différentes. Par exemple on peut avoir un programme avec des faits comme `aime(vincent,marie)` et `aime(vincent,jean,marie)`. Prolog considère que le prédicat `aime` utilisé avec deux arguments et le prédicat `aime` utilisé avec trois arguments sont deux prédicats différents que l'on notera `aime/2` et `aime/3` pour les distinguer.

Un terme exécutable (callable en anglais) est un atome ou un terme composé.

Toutes les catégories de termes que nous venons de voir ci-dessus sont testables grâce aux prédicats `var/1, atom/1, integer/1, float/1, number/1, atomic/1, compound/1, callable/1` dont la documentation est donnée à l'url `http://www.gprolog.org/manual/html_node/gprolog024.html`.

```
?- var(X).
yes
?- var(7+X).
no
?- compound(7+X).
yes
?- callable(7+X).
yes
?- callable(homme).
yes
```

En cas de doute sur la syntaxe et la terminologie ci-dessus, n'hésitez pas à utiliser ces prédicats.

1.4.6 Syntaxe des faits, règles et questions

Un fait est un atome ou un terme composé, autrement dit un terme exécutable.

Une tête de règle est un terme exécutable.

Pour INF242, on ne considéra que des corps de règles qui sont des suites de termes exécutables séparés par des virgules. Les éléments d'une telle suite sont les éléments du corps de la règle.

Un fait peut-être vu comme une règle de forme particulière, en effet le fait `homme(socrate)` est identique du point de vue de l'exécution à la règle `homme(socrate):-true`, autrement dit un fait est une règle dont le corps ne comporte aucun terme.

Une requête est une suite de termes exécutables séparés par des virgules. Les éléments d'une telle suite sont les éléments de la requête.

1.4.7 Les opérations

Le terme `2+3*4` est un terme infixé. On peut en connaître la notation préfixée grâce au prédicat display

```
?-display(2+3*4).
+(2,*(3,4))
yes
```

Une opération est un foncteur unaire ou binaire, qu'on peut noter en préfixé, infixé ou postfixé *sans parenthèses*. En prolog, on peut facilement ajouter de nouvelles opérations ou modifier les manières d'écrire les opérations existantes.

On précise les notations des opérations en leur associant un rang et une classe. On donne ci-dessous une table avec quelques opérations prédéfinies

Rang	*Classe*	*Noms*
1200	xfx	:-
1000	xfy	,
700	xfx	= < <= => >
500	yfx	+ -
400	yfx	* /
200	fy	+ -

Les rangs sont dans l'ordre inverse des priorités. Le foncteur d'un terme est celui correspondant à l'opération de plus grand rang, autrement dit la moins prioritaire. Par exemple, le foncteur de `2+3*4` est + car l'addition est l'opération binaire de plus grand rang.

Dans la classe, la lettre `f` précise la position de l'opération et les lettres `x, y` la position de ses opérandes. La lettre `y` permet d'avoir comme opérande une suite de termes avec l'opération et la lettre `x` l'interdit : par exemple, il est interdit d'écrire `1 < X < 2` mais on peut écrire le terme `1 + X + 2`.

Si la lettre `y` est à gauche de `f`, l'opération `f` la plus à gauche est prioritaire.

Si la lettre `y` est à droite de `f`, l'opération `f` la plus à droite est prioritaire.

Cela explique comment dans le tableau ci-dessous les termes à gauche avec opérations correspondent aux termes à droite sans opération.

```
1+X+2                    +(+(1,X),2)
1*2+3*4                  +(*(1,2),*(3,4))
p(X) :- q(X),r(X),s(X)    :-(p(X),','(q(X),','(r(X),s(X))))
```

Les guillemets `','` sont obligatoires dans le dernier terme, car la virgule n'est pas un caractère spécial, donc n'est pas un atome, donc ne peut pas être utilisée comme foncteur. Par contre `','` est un atome.

On peut modifier ou ajouter des opérations grâce au prédicat `op/3` et consulter le rang et le type d'une opération grâce au prédicat `current_op/3`.

Résumé : la seule structure de données de Prolog est le terme. La représentation interne d'un terme composé est celle d'un foncteur appliqué à des termes. Mais on peut aussi utiliser et définir des notations préfixées, infixées et suffixées.

2 Unification

2.1 Substitution

2.1.1 Définition

Une substitution (encore appelée un contexte) est une liste d'égalités de la forme `X = T` entre une variable `X` et un terme `T` *distinct* de la variable, telle que deux égalités de la substitution ne commencent pas par la même variable.

`X = 3, X = f(Y)` n'est pas une substitution, car deux égalités commencent par la même variable.
`X = X` n'est pas une substitution.
`X = 3, Y = f(2)` est une substitution.

Dans une substitution, l'ensemble des variables à gauche du signe égal, est le domaine de la substitution. Le domaine de la substitution `X = Z, Y = f(2)` est `X, Y`.

2.1.2 Instance d'un terme

En appliquant la substitution `X = Z, Y = f(2)` au terme `g(X,Y,Z,a)` on obtient le terme `g(Z,f(2),Z,a)`. En appliquant la substitution `X = Z, Y = f(2)` au terme `U` on obtient `U`.

Soit x une variable et σ une substitution. La valeur de x dans σ est :
– x si x n'est pas dans le domaine de σ (voir l'exemple ci-dessus)
– t si $x = t$ figure dans σ

Une variable est *instanciée* par une substitution si la substitution comporte une égalité entre la variable et un terme qui n'est pas une variable.
Exemple :

```
?-X = 4, var(X).
no
```

Juste après `X = 4`,`X` est instanciée, donc `var(X)` échoue.

```
?-X = Y, var(X).
yes
```

Juste après `X = Y`, `X` n'est pas instanciée, donc `var(X)` réussit.

Soit σ une substitution et t un terme. La valeur de t dans σ est le terme noté $t\sigma$ obtenu en remplaçant chacune des variables de t par sa valeur.
Exemple :

```
?- X = 4, Y = 2, write(f(X,Y,g(X))).
f(4,2,g(4))
yes
```

Le prédicat write/1 affiche la valeur de son argument dans la substitution courante (on dit aussi le contexte courant) sur la sortie courante.

2.1.3 Composer les substitutions

Soient σ et τ deux substitutions. La substitution qui à toute variable x associe le terme $x\sigma\tau$ est notée $\sigma\tau$. Cette substitution a comme égalités :
– $x = t\tau$ si $x = t \in \sigma$ et si $t\tau \neq x$
– $x = u$ si x n'est pas dans le domaine de σ et $x = u \in \tau$

Exemple 2.1 *Soit* σ *la substitution* `X = a, Y = f(a,V),Z = g(a, U).`
Soit τ *la substitution* `X = c, U = h(c), T = i(Z).`
La substitution $\sigma\tau$ *est* `X = a, Y = f(a,V), Z = g(a, h(c)),U = h(c), T = i(Z).`

Exemple 2.2 *Soit* σ *la substitution* `X = Y`.
Soit τ *la substitution* `Y = X`.
La substitution $\sigma\tau$ *est* `Y = X`.

La substitution $\sigma\tau$ est obtenue en appliquant τ à σ, c'est une instance de σ. On dit inversement que σ est plus générale que ses instances.

2.1.4 Substitution idempotente

Une substitution est idempotente si les variables des termes à droite des égalités ne sont pas dans le domaine de la substitution.

Par exemple la substitution `X = 3, Y = f(X)` n'est pas idempotente, tandis que la substitution `X = 3, Y = f(3)` est idempotente.

```
?- Y = 4, X = f(Y, Z).
Y = 4, X = f(4, Z).
```

On note que la réponse de prolog est une substitution idempotente.
Les réponses calculées par prolog sont des substitutions idempotentes.
La substitution σ est idempotente si et seulement si $\sigma\sigma = \sigma$ ce qui justifie le choix du mot idempotent.

2.2 Unificateur le plus général

2.2.1 Définition

Commençons par un exemple

```
?- f(a,X) = f(Y, b), write(f(a,X)), write('\n'),write(f(Y,b)).
f(a,b)
f(a,b)
X = b, Y = a
yes
```

Grâce au prédicat égal, Prolog trouve une substitution rendant identiques les deux termes `f(a,X)` et `f(Y,b)`. On dit que cette substitution unifie les deux termes ou encore que c'est un unificateur des deux termes.
Lorsque deux termes ont plusieurs unificateurs, Prolog calcule le plus général et nous décrivons ci-dessous ce calcul. Mais prenons un nouvel exemple :

```
?- f(X,Y) = f(g(Y),h(T)).
X = g(h(T)), Y = h(T)
yes
```

Appelons σ la réponse. Il est clair qu'en remplaçant `T` par un terme quelconque, on obtient d'autres unificateurs, mais la réponse est l'unificateur le plus général (au sens du paragraphe 2.1.3).

Une substitution est un unificateur de deux termes si les instances de ces deux termes par la substitution sont *identiques*. Elle est une solution de l'égalité entre deux termes si c'est un unificateur de ces deux termes. Elle est solution d'une liste d'égalités, si elle est solution de chaque égalité de la liste.

2.2.2 Une solution la plus générale ou plusieurs

Avant de décrire le calcul de la solution la plus générale d'une liste d'égalités, constatons qu'elle n'est pas unique :

```
?- f(A, B) = f(X, Y).
X = A, Y = B.
yes
```

Mais il y a trois autres solutions «la plus générale» obtenues en changeant les égalités entre variables :
`X = A, B = Y`
`A = X, Y = B`
`A = X, B = Y`
Deux unificateurs les plus généraux de deux termes, sont des instances l'un de l'autre, donc sont identiques à l'orientation près des égalités entre variables.
En cas d'égalité entre deux variables, *prolog* met la plus grande (dans l'ordre alphabétique) à gauche.

2.2.3 Calcul de la solution la plus générale

On indique comment calculer la solution la plus générale d'une liste d'égalités entre termes indépendamment du langage prolog puis nous verrons comme procède **gprolog**.

Developper remplacer $f(s_1,\ldots s_n) = f(t_1,\ldots t_n)$ par $s_1 = t_1,\ldots s_n = t_n$

Enlever enlever $t = t$

Orienter remplacer $t = x$ où x est une variable et t n'est pas une variable par $x = t$.

Éliminer une variable S'il y a une égalité de $x = t$ où x est une variable ne figurant pas dans t (test d'occurence) alors on remplace x par t dans toutes les *autres* égalités

Echec par désaccord il n'y a pas de solution s'il y a une égalité entre deux termes, qui ne sont pas des variables, de foncteurs ou d'arité distinctes.

Echec par test occurrence il n'y a pas de solution s'il y a une égalité entre une variable et un terme composé contenant cette variable.

Voici quelques exemples de requêtes et de leur réponses, illustrant l'unification

Requêtes	Réponses
`jean = marie`	`no`
`jean = 'jean'`	`yes`
`f(X) = f(Y, Z)`	`no`
`f(X) = f(1)`	`X = 1`
`k(s(g),t(k)) = k(X,t(Y))`	`X = s(g), Y = k`
`X = g(Y), Y = h(T)`	`X = g(h(T)), Y = h(T)`
`aime(jean, anne) = aime(X, X)`	`no`

2.2.4 gprolog et l'unification

Tout d'abord,comme on l'a vu plus haut, **gprolog** oriente les égalités entre variables (voir 2.2.2). Mais surtout, **gprolog** applique l'algorithme ci-dessus aux égalités, en oubliant (pour aller plus vite) les tests d'occurrence, ce qui a deux conséquences importantes :

1. `gprolog` trouve des solutions là où il n'y en a pas.
   Considérons le programme suivant

   ```
   unifier :- X = f(X).
   ```

   A la requête `unifer`, `gprolog` répond `yes`, car il admet incorrectement que le système `X = f(X)` a une solution. `gprolog` termine par une erreur fatale la requête `X = f(X)`, car il trouve la solution incorrecte `X = f(X)`, qu'il est incapable d'afficher.
2. Quand il n'y a pas de solution, il peut ne jamais terminer, par exemple en cherchant une solution de la requête : `?- X = f(X,Y), Y = f(X,Y).`

C'est au programmeur d'éviter ces deux pièges. En cas de besoin, il peut notamment unifier deux termes en utilisant le prédicat `unify_with_occur_check/2` qui calcule *correctement* l'unificateur principal de ses deux arguments.
Résumé : en absence d'égalités entre une variable et un terme composé comportant cette variable, gprolog sait calculer l'unificateur le plus général de deux termes. C'est au programmeur d'éviter les égalités entre une variable et un terme composé comportant cette variable.

3 Arithmétique en gprolog

3.1 Expressions arithmétiques

Une expression arithmétique est un terme dont les foncteurs sont des opérations arithmétiques. On donne une liste des opérations les plus courantes : `+, -, *, //`(quotient entier), `/`(quotient exact), `rem`(reste).

```
?- X = 2 + Y*4, display(X), nl, write(X), nl.
+(2,*(_17,4))
2+_17*4
X = 2+Y*4
yes
```

On voit que `display(X)` affiche la valeur de `X` (dans le contexte courant) dans la notation préfixée, que `nl` produit un retour à la ligne et que `write(X)` affiche dans la valeur de `X` dans la notation infixée. Mais on voit que les opérations ne sont pas effectuées.

En prolog, les expressions arithmétques sont calculées uniquement lorsqu'elles sont arguments des prédicats arithmétiques ou argument droit du prédicat `is`.

3.2 Prédicats arithmétiques

Soit E une expression arithmétique, on distingue l'expression E et eval(E) la valeur de E quand on effectue les opérations de E. Pourque le nombre eval(E) soit défini, il faut que les variables de E aient comme valeurs des nombres, que les opérations de E soient des opérations connues de gprolog, que ces opérations soient appliquées aux nombres du bon type et qu'il n'y ait pas de division par 0. Pour avoir plus d'information, il faut lire la documentation de gprolog.

1. `E1 =:= E2` réussit si eval(E1) = eval(E2).
2. Les autres prédicats arithmétiques sont =\= (différent), <, <=, > , >= et se comportent comme =:=, leurs arguments sont d'abord évalués puis comparés

On donne des requêtes et leurs réponses, pour illustrer leur comportement

Requêtes	Réponses	Explications
`2 + 3 =:= 5`	`yes`	eval(2+3) = 5
`2 + 3 = 5`	`no`	les termes `2 + 3` et 5 sont distincts
`2 + 3 =:= 6`	`no`	eval(2+3) = 5 et $5 \neq 6$
`X = 2, Y = 3, X + Y =:= 5`	`yes`	eval(X+Y) = 5
`3 + 4 < X + 5`	`Erreur`	la valeur de `X` est inconnue
`X = 2, Y = 3, f(X) + Y <= 6`	`Erreur`	la fonction f est inconnue
`X = 2, Y = 3, Y > 2 /(X - 2)`	`Erreur`	division par 0
`X = 2.0, Y = 4.0, X =:= Y // X`	`Erreur`	les opérandes de // doivent être des entiers
`X = 2, Y = 4, X =:= Y // X`	`X = 2, Y = 4`	eval(X) = eval(Y // X)

3.3 Le prédicat `is/2`

`R is E`
calcule R = eval(E), autrement dit unifie R et eval(E).

Requêtes	Réponses	Explications
`X is 2 + 3`	X = 5	`X = eval(2+3)`
`X is 2 + Y`	`Erreur`	la valeur de `Y` est inconnue
`5 is 2 + 3`	`yes`	5 = eval(2+3)
`2 + 3 is 2 + 3`	`no`	2 + 3 = eval(2+3) échoue

Traitons un exemple

1. Spécification
   `p2(N,R)`
   condition d'utilisation : N entier positif ou nul
   résultat : R = eval(2**N)
   exemple :
   ```
   ?- p2(3,R).
   R = 8
   ```
2. Construction de la solution (par récurrence sur le premier argument)
   p2(0,R) vrai si R = 1
   p2(N,R) vrai si N > 0 et p2(N-1,S) et R = 2*S
   Cette étape est une explication convaincante de la manière de construire le programme sans traiter tous les détails. En effet supposons par récurrence que l'appel récursif soit correct, donc que S = 2**(N-1), on a alors p2(N,R) vrai car R = 2*S = 2**N.
3. Solution
   ```
   p2(0, 1).
   p2(N, R) :- N > 0, NM1 is N - 1, p2(NM1, S), R is 2 * S.
   ```
   On doit tenir compte de ce que les opérations ne sont effectuées que dans les arguments des prédicats arithmétiques et dans l'argument droit du prédicat `is`.
   On doit tenir compte de l'ordre d'évaluation : la valeur de `NM1` doit être connue avant l'appel récursif, la valeur de `R` doit être calculée après celle de `S`.
   La deuxième règle peut être essayée quand `N` vaut 0, il faut donc mettre le test `N > 0` pour interdire un appel récursif de `p2` avec un premier argument négatif.

Essayons le programme maintenant qu'il est écrit :

```
?- p2(3, X).
X = 8 ?;
no
```

Ce comportement n'est pas étonnant, il est du au fait que lorsque le premier argument vaut 0, prolog va quand même essayer la deuxième règle. Dans la suite du cours, on verra comment analyser l'exécution, puis comment la rendre plus efficace en évitant cet essai inutile.

3.4 Arithmétique et Logique

En mathématiques, on mélange sans en être conscient, le raisonnement (la logique) et le calcul. En Prolog, on a la même situation : les règles logiques sont mélangées de façon naturelle aux calculs que nous venons de voir.

L'approche traditionnelle des logiciens, a consisté à axiomatiser complètement les mathématiques, ce qui n'a rien à voir avec l'activité réelle des mathématiciens. L'approche moderne de la logique tente de définir plus clairement la part du raisonnement et celle du calcul afin d'assister par des machines l'activité du mathématicien.

4 Listes en prolog

4.1 Notations et définitions des listes

Commençons par des exemples de listes

- `[]` est la liste vide
- `[1, 2, 3]` est la liste d'éléments 1, 2 et 3
- `[1 | [2, 3]]` est la même liste, la barre sépare le premier élément de la liste et la fin de la liste `[2, 3]`
- `[1, 2 | [3]]` est la même liste, la barre sépare les deux premiers éléments de la liste et la fin de la liste `[3]`

Définition des listes :
L'ensemble des listes est le plus petit ensemble de termes comportant la liste vide et telle que si `Xs` une liste alors `[X | Xs]` est une liste de premier élément `X` et de fin `Xs`.
Notons que `[1 | 2]` n'est pas une liste, car 2 n'est pas une liste, par contre `[1, 2]` est une liste qui s'écrit aussi sous l'une des formes :

```
[1|[2]]
[1|[2|[]]]
```

Il y a un gprolog un prédicat `list/1` qui permet de tester que son argument est une liste.

```
?- list([1 | 2]).
no
?- list([1 , 2]).
yes
?- list([X | Xs]).
no
?- Xs = [5,6],list([X | Xs]).
yes
```

Les listes peuvent être notées aussi avec une notation préfixée totalement parenthésée où :
`[X|Xs] = .(X,Xs)`
On donne des requêtes pour se familiariser avec ces notations :

```
?- [1, 2]= .(1,.(2,[])).
% l'espace après = est obligatoire pour que = soit un atome et non pas =.
yes
?- [1, 2, 3] = [X | Xs]. % les espaces sont optionnels
X = 1, Xs = [2,3]
yes
?-[1, 2, 3, 4] = [X1, X2 | Xs].
X1 = 1, X2 = 2, Xs = [3, 4].
```

4.2 Récurrences sur les listes

4.2.1 Élément d'une liste

1. Spécification
   `element(X,L)`
   condition d'utilisation : L est une liste
   résultat : `X` est élément de la liste `L`
   exemple :
   ```
   ?- element(X,[1, 2]).
   X = 1 ?;
   X = 2
   ```
2. Construction de la solution (par récurrence sur le deuxième argument)
   element(X,L)
   si le premier élément de L est X, c'est-à-dire si L = [X | _]
   ou
   si X est élément de la fin de L, c'est-à-dire si L = [_ | M] et element(X,M)
3. Solution

   ```
   element(X,[X | _]).
   element(X,[_ | M]) :- element(X, M).
   ```

   Essayons le programme tel qu'il est écrit :

```
?- element(X,[1, 2]).
X = 1 ?;
X = 2 ?;
no
```

Lorsque le deuxième argument est un liste, prolog essaie les deux règles, donc il ne peut pas deviner quand il a obtenu la dernière réponse grâce à la première règle, que la deuxième règle va conduire à un échec. Ce prédicat existe en gprolog sous le nom de **member**.

4.2.2 Concaténer deux listes

1. Spécification
   `conc(L1, L2, L12)`
   condition d'utilisation : L1 est un liste
   résultat : L12 est la concaténation des deux listes L1 et L2
   exemple :

   ```
   ?- conc([1,2],[3,4], L).
   L = [1, 2, 3, 4]
   yes
   ```

2. Construction de la solution (par récurrence sur le premier argument)
   conc([],L, R) si R = L
   conc([X| Xs],L, R) si conc(Xs, L, S) et R = [X | S]

3. Solution

   ```
   conc([],L, L).
   conc([X| Xs],L,[X | S]) :- conc(Xs, L, S).
   ```

   On observe que si le troisième argument est une liste, on peut aussi utiliser cette solution pour décomposer une liste en deux listes, comme le montre les requêtes ci-dessous :

   ```
   conc(L1,L2,[2]).
   L1 = [] L2 = [2]; ?
   L1 = [2] L2 = []; ?
   no
   ```

Le prédicat ci-dessus existe en gprolog sous le nom de **append**.

5 Modèles d'exécution

5.1 Renommage des faits et règles

(a) Soit le programme

```
p(X).
```

À ce programme on pose la requête $p(f(X))$. La réponse doit être `yes` car $p(f(X))$ est conséquence de $\forall X\, p(X)$.

Prolog obtient cette réponse de la façon suivante Il change $p(X)$ en $p(Y)$, où Y est une nouvelle variable. Il pose $p(f(X)) = p(Y)$ et l'algorithme d'unification trouve la solution $Y = f(X)$. Le nouveau but est vide donc la réponse est yes. Notons que le changement de variable est obligatoire, car $p(f(X)) = p(X)$ n'a pas de solution.

(b) Soit le programme

```
p(f(X)).
```

À ce programme on pose la requête $p(X)$. La réponse doit être : X est de la forme $f(\_)$.

Prolog obtient cette réponse de la façon suivante Il renomme $p(f(X))$ en $p(f(Y))$. Il pose $p(X) = p(f(Y))$ et l'algorithme d'unification trouve la solution $X = f(Y)$. Le nouveau but est vide et la réponse est $X = f(Y)$, ce que prolog affiche comme $X = f(\_)$ car le nom Y, ne figurant pas dans la question, est sans importance Comme ci-dessus, le changement du nom de variable est une nécessité.

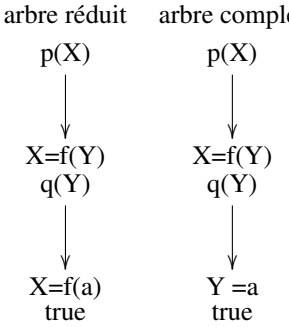
(c) Soit le programme

```
p(f(X)):-q(X).
q(a).
```

À ce programme on pose la requête $p(X)$. La réponse est : $X = f(a)$.

Prolog obtient cette réponse de la façon suivante Il renomme $p(f(X)) : -q(X)$ en $p(f(Y)) : -q(Y)$. Il pose $p(X) = p(f(Y))$, donc il obtient le nouveau but $q(Y)$ sachant que $X = f(Y)$. En résolvant $q(Y) = q(a)$, il trouve $Y = a$ donc en composant les deux substitutions, il calcule la réponse $X = f(a)$.

On explique la demarche de Prolog de réduction des buts à la l'aide des règles à l'aide d'arbres de buts, appelés arbres d'exploration. On distingue un arbre d'exploration **réduit** dans lequel les substitutions produites par les unifications ne sont pas conservées et un arbre d'exploration **complet** dans lequel on garde ces substitutions qui ne sont composées qu'au moment du calcul des unifications et des réponses.



Notez la différence entre les deux arbres. Dans l'arbre réduit, on garde les valeurs des variables du but initial (la valeur de X dans l'exemple) tandis que dans l'arbre complet, on garde les substitutions calculées par les unifications et la valeur de X est obtenue en composant ces substitutions. Le but vide est noté true.

À la main (examen et TD), on ne demande que de retrouver les arbres réduits. Prolog implémente l'arbre d'exploration complet, mais visualise dans les outils de trace les arbres réduits.

5.2 Retour-arrière

Soit le programme

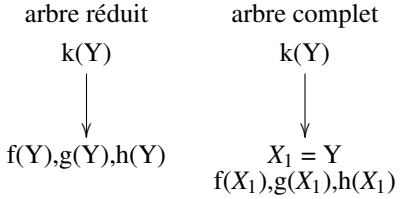
```
f(a).
f(b).
g(a).
g(b).
h(b).
k(X) :- f(X),g(X),h(X).
```

A ce programme, nous posons la requête `k(Y).`

Prolog essaie d'appliquer une règle à ce but. Il n'utilise pas directement la règle, mais, comme on l'a vu ci-dessus, il utilise une copie avec une *nouvelle variable* : $k(X_1) : -f(X_1), g(X_1), h(X_1)$.

Prolog résoud $k(X_1) = k(Y)$, trouve la solution $X_1 = Y$, et l'applique à la partie droite de règle. Notons que dans ce cas, il y a aussi la solution $Y = X_1$. Mais que Prolog choisit toujours l'orientation qui met la nouvelle variable à gauche ce qui évite d'avoir à conserver la valeur de Y dans l'arbre réduit.

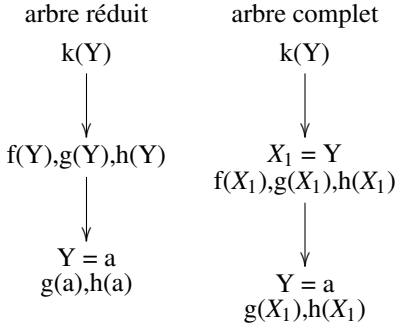
Ainsi Prolog remplace la requête initiale par la requête $f(Y), g(Y), h(Y)$. Graphiquement on représente cette situation par un arbre réduit et un arbre complet



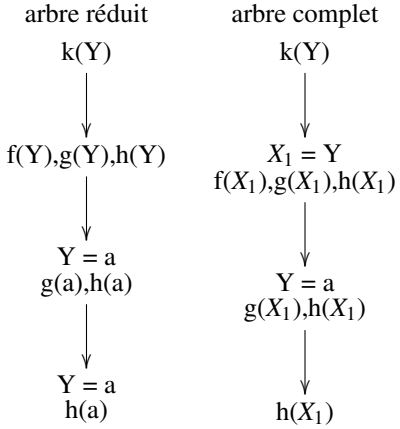
Chaque fois que Prolog a une liste de buts à prouver, il le fait de gauche à droite, donc il cherche une preuve du but $f(Y)$.

Notons que sur l'arbre complet, il faut effectuer la substitution pour trouver ce but, ce qui explique que nous, comme prolog, ne demandons et ne visualisons que les arbres réduits, sauf dans ce paragraphe, où nous voulons expliquer comment Prolog s'exécute.

Prolog explore la base de connaissances «de haut en bas» et trouve ainsi le fait $f(a)$ qui s'unifie avec $f(Y)$ en donnant la liaison $Y = a$. Il reste donc à prouver la requête $g(a), h(a)$. La représentation graphique de la preuve est maintenant :



Le fait $g(a)$ étant dans la base, il reste à prouver le but $h(a)$, comme le montre l'état de l'exploration obtenu après application du fait $g(a)$.



La substitution vide (qui ne change rien) n'est pas écrite, c'est pourquoi il n'y a pas de substitution dans la feuille de l'arbre complet.

Comme la base ne comporte, ni fait, ni tête de règle unifiable avec $h(a)$, Prolog constate que ce but est improuvable et recherche en *remontant* le chemin qu'il vient de suivre s'il y a une requête avec une *autre* possibilité pour unifier son premier but avec un fait ou une tête de règle.

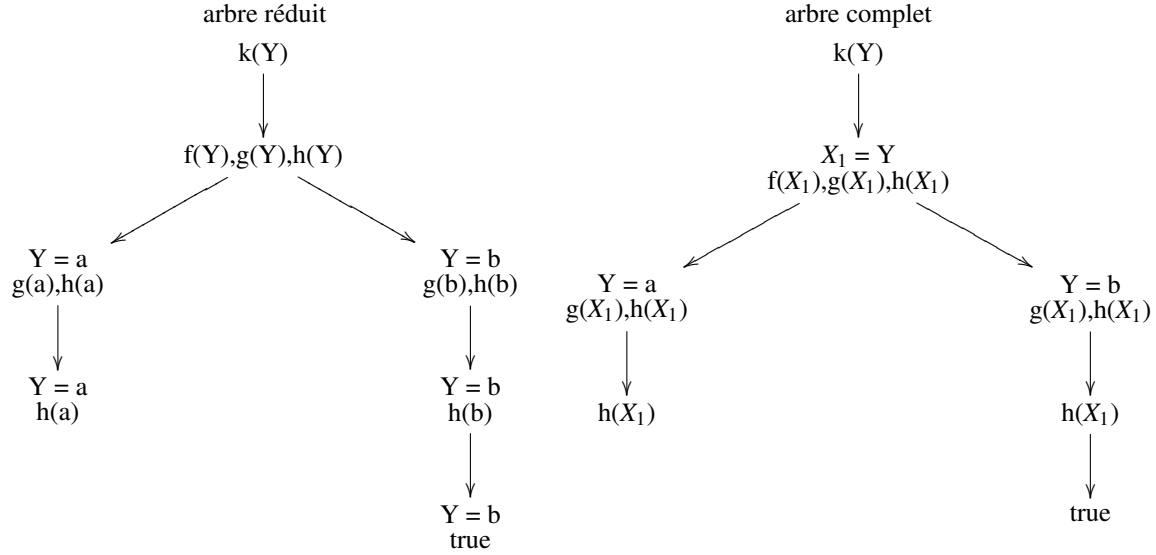
Il n'y a pas d'autre possibilité pour but $g(a)$, mais il y a une autre possibilité pour le but $f(Y)$. Les points de cette exploration , où il y a plusieurs possibilités pour unifier le premier but, sont appelés des *points de choix*. Prolog conserve une trace des points de choix et des choix déjà effectués, de sorte qu'il peut remonter au point de choix précédent avec encore un choix possible. Ce comportement est appelé **le retour-arrière**.

Donc Prolog revient en arrière au plus récent point de choix, où la requête était $f(Y), g(Y), h(Y)$. Il essaie de satisfaire à nouveau le premier but en explorant *plus loin* la base de connaissances. Il voit qu'il peut unifier le premier but avec le fait $f(b)$ en posant $Y = b$. Il reste à prouver la requête $g(b), h(b)$.

Puisque $g(b)$ est un fait, il reste à prouver la requête $h(b)$. Puisque $h(b)$ est aussi un fait, Prolog obtient une requête vide, autrement dit inconditionnellement vraie (ce qui est la raison pour laquelle nous l'avons écrite true). Cela veut dire que Prolog a prouvé tout ce qui était nécessaire pour établir que la requête initiale $k(Y)$ est satisfaisable.

Sur l'arbre réduit, on trouve la réponse $Y = b$ attachée la requête vide, sur l'arbre complet en composant les substitions qui mènent à cette requête, on trouve aussi $Y = b$. Cela veut dire que $k(b)$ est conséquence de la base de connaissances.

En représentant ces dernières étapes de l'exploration, on obtient :



Puisqu'il n'y a plus de points de choix avec des possibilités ouvertes, Prolog a terminé l'exploration associée à la requête initiale. Bien sûr, s'il y avait eu d'autres règles pour le prédicat `k/1`, Prolog les aurait essayées exactement de la manière que nous avons décrite : c'est-à-dire, **en explorant de haut en bas la base de connaissances, de gauche à droite les requêtes, et en revenant en arrière au précédent point de choix à chaque échec.**

5.3 Le renommage en gprolog

Pour renommer facilement les règles, gprolog (comme les autres prolog) remplace les variables par des entiers qu'il affiche comme des variables Prolog en les préfixant par un blanc souligné. On s'en aperçoit quand on utilise les traces gprolog (activées par le prédicat `trace/0` et arrêtées par `notrace/0`) ou quand on fait tracer les arbres d'exploration, comme dans l'exemple suivant :

```
?- question.
k(Y).
Y = _21
k(_21).
  Y = _21
  f(_21), g(_21), h(_21).
    Y = a
    g(a), h(a).
      Y = a
      h(a).
    Y = b
    g(b), h(b).
      Y = b
      h(b).
        Y = b
        succes
yes
```

Si l'on saute les deux premières lignes , on voit l'arbre d'exploration **réduit** de la requête `k(_21)`. Autrement dit non seulement les règles mais aussi les variables de la requête initiale sont renommées avec des entiers. Malheureusement, cela ne facilite pas l'utilisation des outils de trace.

Aussi quand nous demandons en exercice de construire des arbres d'exploration, on demande de renommer une règle appliquée à un but de profondeur *i* en indiçant les variables de la règle par *i*, comme sur le dernier exemple du paragraphe 5.2.

5.4 Renommage et retour-arrière

1. Spécification
   `d(L,LD)`
   condition d'utilisation : L est une liste d'entiers
   résultat : LD est la liste obtenue en multipliant par 2 chaque élément de L
   Par exemple :

   ```
   ?- d([5,7],LX).
   LX = [10,14]
   ```

2. Solution

   ```
   d([],[]).
   d([E|S],[D|R]) :- d(S,R), D is 2*E .
   ```

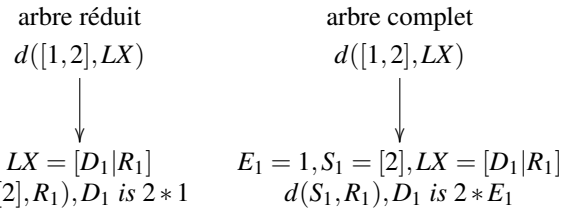
Traçons un arbre d'exploration de la requête $d([1,2],LX)$. La première règle n'est pas applicable. Car l'équation $d([],[]) = d([1,2],LX)$ n'a pas de solution.

On renomme la deuxième règle en $d([E_1|S_1],[D_1|R_1]) : -d(S_1,R_1), D_1$ *is* $2*E_1$ et l'on résoud l'équation $d([E_1|S_1],[D_1|R_1]) = d([1,2],LX)$.

On obtient la solution $E_1 = 1, S_1 = [2], LX = [D_1|R_1]$.

En appliquant cette solution au corps de la règle, on voit que la requête initiale revient donc à trouver des valeurs des variables R_1 et D_1 telles que la requête $d([2],R_1), D_1$ *is* $2*1$ réussisse, sachant que $LX = [D_1|R_1]$.

On représente cette situation par deux arbres réduit et complet. On rappelle qu'on ne vous demande que les arbres réduits, mais que les arbres complets représentent mieux ce qui se passe effectivement.



Prolog cherche maintenant une preuve de $d([2],R_1)$. La première règle n'est pas applicable.

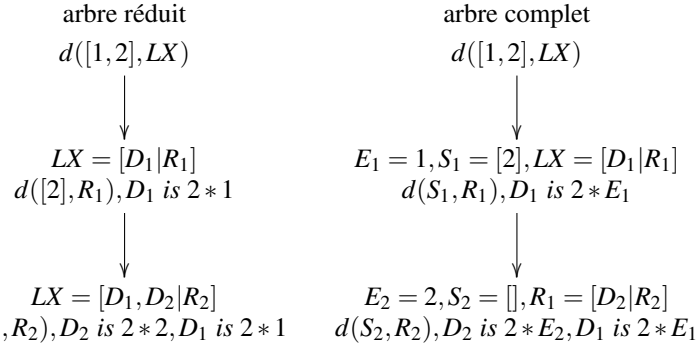
On renomme la deuxième règle en $d([E_2|S_2],[D_2|R_2]) : -d(S_2,R_2), D_2$ *is* $2*E_2$ et l'on résoud l'équation $d([E_2|S_2],[D_2|R_2]) = d([2],R_1)$

On obtient la solution $E_2 = 2, S_2 = [], R_1 = [D_2|R_2]$.

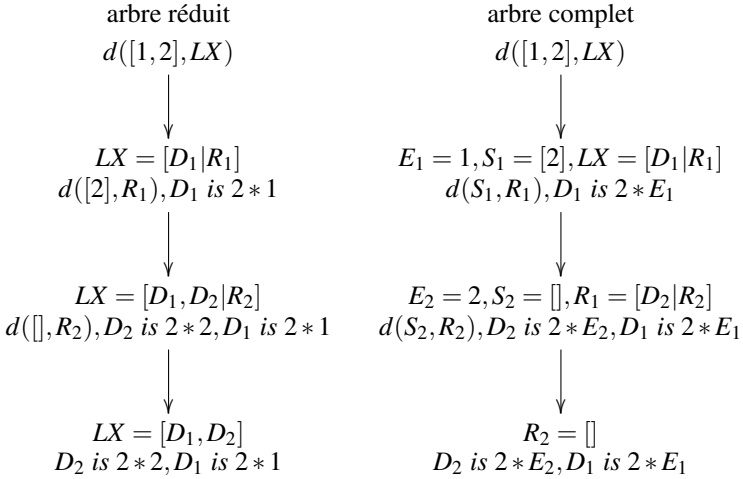
En appliquant cette solution au corps de la règle, on voit que la requête précédente revient donc à trouver des valeurs des variables R_2 et D_2 telles que la requête $d([],R_2), D_2$ *is* $2*2, D_1$ *is* $2*1$ réussisse, sachant que $LX = [D_1, D_2|R_2]$.

Notons qu'en absence de renommage, on aurait posé des conditions impossibles à remplir en imposant en particulier la conjonction des deux buts D *is* $2*2, D$ *is* $2*1$.

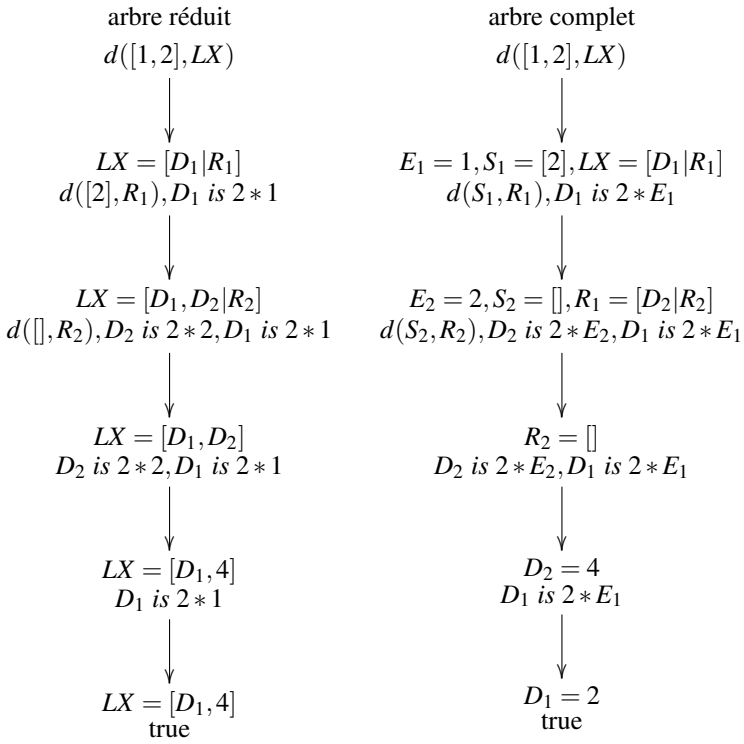
L'oubli du renommage peut empêcher de découvrir des preuves donc a pour conséquence d'enlever des réponses légitimes et parfois d'enlever toutes les réponses



Prolog cherche maintenant une preuve de la requête $d([],R_2), D_2$ *is* $2*2, D_1$ *is* $2*1$. La première règle est applicable, et donne la réponse $R_2 = []$. L'exploration devient alors :



Prolog exécute alors les deux buts de la derniere requête, ce qui successivement lie la variable D_2 à la valeur 4, puis la variable D_1 à la valeur 2. Comme il ne reste plus de choix possible, l'exploration complète de la requête initiale est représentée par les deux arbres :



Sur l'arbre complet, on retrouve la réponse en composant les substitutions de haut en bas, alors que sur l'arbre réduit, on l'obtient directement puisqu'on a appliqué ces substitutions au fur et à mesure.

5.5 true, yes ou no

On rappelle que la **dernière** réponse à une requête est suivie d'un point d'interrogation si prolog ne sait déterminer qu'il s'agit de la dernière réponse (voir 1.3.2) et ce point d'interrogation est suivi d'un no, signifiant que Prolog échoue dans la recherche d'autre réponse. Par contre prolog fait suivre la **dernière** réponse d'un yes lorsqu'il sait que c'est la dernière réponse (voir 1.3.3).

Ce comportement surprend le débutant et nous allons examiner un petit exemple. Voici notre programme :

```
a(1,2).
a(2,3).
a(1,3).
```

Première requête :

```
a(1,2).
true ?;
no
```

Prolog a démontré la question, d'où la réponse **true** puis il essaie les autres règles de la relation **a**, pour savoir s'il y a une autre preuve de la question. Comme il n'y a pas d'autres preuves, la réponse suivante est no.

Deuxième requête :

```
a(1,3).
yes
```

Prolog a démontré la question, et comme il n'y a pas de règle suivante pour **a**, il sait que c'est la dernière réponse.

Troisième requête :

```
a(2,3).
yes
```

Prolog a démontré la question et il sait que c'est la dernière preuve possible. Cela peut surprendre. Mais gprolog recherche les règles en s'aidant du foncteur du premier argument (ou de l'entier premier argument dans notre exemple), ce qui accélère l'exécution : dans notre cas, cela permet à prolog de savoir avant unification que la troisième règle n'est pas applicable.

Comme me l'a indiqué Paulo Moura, la plupart des compilateurs prolog, dont gprolog, utilise ce mécanisme pour accélerer l'exécution des requêtes.

5.6 Prolog et les preuves

L'interêt majeur de prolog est qu'en plus d'être un langage de programmation simple, l'exécution d'un programme est une preuve.

5.6.1 Les réponses de prolog aux requêtes sont correctes

Si la substitution σ est une réponse du programme P à la requête R alors Rσ est **une conséquence** du programme P. Ce que nous venons d'affirmer, n'est vrai qu'aux deux conditions suivantes :

1. les programmes doivent être en prolog «pur», autrement dit ils ne doivent utiliser **aucun** prédicat prédéfini
2. les unifications qui ont permis de calculer la réponse ne doivent pas avoir unifié incorrectement une variable et un terme composé contenant cette variable.

La preuve de ce résultat est une conséquence immédiate de ce que les trois règles logiques des preuves prolog sont correctes. En respectant quelques précautions, ce résultat peut être généralisé aux programmes faisant des évaluations d'expressions arithmétiques.

5.6.2 Les réponses de prolog aux requêtes sont les plus générales

Supposons que la substitution σ est une réponse du programme P à la requête R.
Supposons que cette substitution est une instance d'une substitution τ telle que Rτ est une conséquence de P, autrement dit la réponse σ ne serait pas la plus générale.
Alors les deux substitutions sont égales à l'orientation prés des égalités entre deux variables, autrement dit τ n'est pas plus générale que σ.

La preuve de ce résultat est une conséquence de ce que l'algorithme d'unification calcule l'unificateur le plus général de deux termes.

5.6.3 Quand une requête termine, prolog calcule toutes ses réponses

Soit σ une substitution telle que Rσ est conséquence du programme P. Si la requête R **termine**, alors prolog calcule une réponse plus générale que σ.

Cette complétude est due à deux mécanismes de Prolog :
– le calcul des unificateurs les plus généraux.
– l'essai de toutes les règles pour effacer un but

Comme il est évident dans l'énoncé de cette propriété, si la requête R ne se termine pas, on ne peut rien dire sur ce qui est calculé par prolog. Prenons un exemple
Soit P le programme

```
frere(X,Y):- frere(Y, X).
frere(michel, claude).
```

Il est clair que de ce programme, on peut déduire `frere(michel, claude)` et `frere(claude, michel)`. Mais la requête prolog `frere(X,Y)` ne nous donne aucune de ces deux réponses, car elle va réduire la requête initiale à `frere(Y,X)` puis à `frere(X,Y)` : cette requête ne se termine pas en théorie et en pratique se termine par un débordement de la pile.

La morale de cet exemple est qu'il faut pour chaque prédicat

1. définir ses conditions d'utilisation
2. prouvez que toute exécution du prédicat, respectant ces conditions d'utilisations, termine

Cela ne doit vous empêcher d'expérimenter ce qui se passe, quand on ne respecte pas les conditions d'utilisation préconisées. On a dit que le prédicat `member(X,L)` (element en français, voir 4.2.1) ne doit être utilisé que si L est une liste. Expérimentons que qui se passe, lorsque L n'est pas une liste :

```
?- member(4,L).
L = [4|_] ? ;
L = [_,4|_] ? ;
L = [_,_,4|_] ? ;
L = [_,_,_,4|_] ?
yes
```

Après le dernier point d'interrogation, on a tapé «return» pour arrêter l'énumération des réponses qui sont toutes les listes ayant comme élément 4.

6 Compléments sur les termes

Pour décomposer ou construire des termes composés, on dispose de 3 prédicats.

6.1 functor

Le prédicat functor permet de décomposer un terme ou de construire un terme.

```
?- functor(f(a,b),F,A).
A = 2
F = f
yes
?- functor(a,F,A).
A = 0
F = a
yes
?- functor([a,b,c],X,Y).
X = '.'
Y = 2
yes
?- functor(T,f,2).
T = f(_,_)
```

La requête `functor(T,F,A)` réussit si F est le foncteur et A l'arité du terme T. Si T n'est pas une variable non instanciée, on obtient son foncteur et son arité, comme le montrent les 3 premières requêtes. Si T est une variable non instanciée, la dernière requête montre que l'on construit un terme.

6.2 arg

Le prédicat arg permet de connaître les arguments d'un terme.

```
?- arg(2,aime(vincent,marie),X).
X = marie
yes
?- arg(2,aime(vincent,X),marie).
X = marie
yes.
?- arg(2,heureux(juliette),X).
no
```

La requête `arg(N,T,A)` unifie l'argument N de T avec A. Elle échoue si cette unification échoue ou si T n'a pas N arguments. La requête termine par une erreur si N n'est pas un entier ou si T n'est pas instanciée.

6.3 Le prédicat =..

```
?- f(a,b)=..X.
X = [f,a,b]
Yes
?- X =.. [f,a,b].
X = f(a,b).
Yes
```

La requête `T=..[F|LA]` unifie le terme T avec un terme de foncteur F et de listes d'arguments LA.

Ecrivons le prédicat `variables(T,L)` qui réussit si et seulement si `L` est la liste des variables du terme `T`.

Avant de passer à la rédaction de ce prédicat, on rappelle la définition des prédicats `var, nonvar` et on donne des exemples de leur fonctionnement.
La requête `var(X)` réussit si et seulement si `X` est une variable non instanciée, c'est-à-dire dont la valeur n'est ni un nombre, ni un atome, ni un terme composé. La requête `nonvar(X)` réussit si et seulement si `var(X)` échoue. On donne quelques exemples d'usage du prédicat `var` :

```
?- var(X).
yes
?- X=2,var(X).
no
?- X=Y, var(X).
yes
```

Nous sommes près à écrire le prédicat `variables` en français puis à l'aide de règles prolog.

- Si `T` est une variable alors `[T]` est sa liste de variables
- Si `T` est un terme composé dont la liste d'arguments est `LA`, alors la liste des variables de `T` est celle de la liste des variables de la liste de termes `LA`.
- La liste des variables de la liste vide est vide
- La liste des variables d'une liste de termes `[T | S]` est la concaténation de la liste des variables du terme `T` et de la liste des variables de la liste de termes `S`

Quand on traduit cette définition en prolog, on doit introduire le prédicat
`variables_de_la_liste(LT, LVT)`
condition d'utilisation : `LT` est une liste
résultat : `LVT` est la liste des variables de la liste `LT`.

```
/* variables(T,L) réussit si et seulement si L est la liste des variables
      du terme T */
variables(T,[T]):- var(T).
variables(T,L) :- nonvar(T),T =..[F|LA],variables_de_la_liste(LA,L).
variables_de_la_liste([],[]).
variables_de_la_liste([T|S],LVTS) :-
       variables(T,LVT),variables_de_la_liste(S,LVS),append(LVT,LVS,LVTS).
```

7 Contraintes sur les domaines finis

7.1 Introduction

A quoi sert la programmation avec contraintes La programmation avec des contraintes est une technique logicielle en développement pour la description et la résolution de problèmes importants, particulièrement combinatoires dans les domaines de la planification et de l'ordonnancement.

Elle est employée avec succès dans les domaines suivants

- applications graphiques, pour exprimer des relations géométriques entre divers éléments d'une scène
- traitement des langues naturelles (pour améliorer la performance des analyseurs)
- système de base de données (pour assurer ou restaurer la cohérence des données)
- problèmes de recherche opérationnelle (en particulier les problèmes d'optimisation)

et bien d'autres domaines non cités.

Prolog et les contraintes Le langage Prolog est né en 1970. Son apogée a eu lieu dans les années 80, en liaison avec des applications au traitement des langues naturelles et à l'intelligence artificielle. Son déclin provisoire est suivi d'une résurrection avec l'introduction des *contraintes* dans le langage.

Une *contrainte* c'est une relation entre des variables, chacune prenant une valeur dans un *domaine donné*. Une contrainte restreint les valeurs possibles que les variables peuvent prendre.

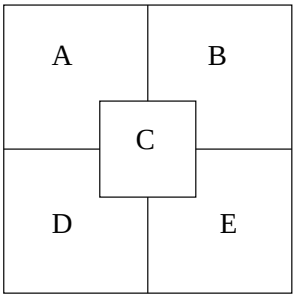
Donnons un exemple tiré de `gprolog`. La contrainte `fd_domain([X,Y,Z],1,10)` impose que ces trois variables prennent des valeurs entières entre 1 et 10, la conjonction de cette première contrainte et de la contrainte `X+Y #= Z` restreint les domaines de X et Y à l'intervalle fermé 1..9 (car X est au plus égal au maximum de Z moins le minimum de Y) et le domaine de Z à l'intervalle fermé 2..10 (car le minimum de Z est au moins la somme des minimums de X et de Y).

Prolog et les contraintes ouvre le nouveau domaine de la programmation logique avec contraintes (PLC) : Prolog est étendu avec des résolveurs de contraintes, travaillant sur des données qui peuvent être des entiers, des booléens, *des domaines finis d'entiers*, des rationnels, des intervalles de rationnels.

Les résolveurs de contraintes peuvent être ajoutés à d'autres langages que Prolog, mais l'algorithme d'unification de Prolog est déjà un résolveur des contraintes d'égalité sur les termes, donc la résolution de contraintes s'intègre naturellement à Prolog.

7.2 Un problème combinatoire trivial

On donne la carte ci-dessous et on souhaite en connaître un coloriage en 3 couleurs (s'il existe) de sorte 2 régions frontalières aient des couleurs distinctes.



Il a deux manières de résoudre ce type de problème :

1. Énumérer tous les coloriages possibles et tester chaque coloriage obtenu pour voir s'il vérifie les conditions données. Ce qui nous donnera au plus 3^5 coloriages à examiner.
2. Poser d'abord les contraintes du problèmes, et seulement après, énumérer les coloriages en coupant l'énumération dès que les contraintes ne sont plus satisfaites

L'utilisation des contraintes en `gprolog` permet de pratiquer facilement la deuxième démarche.

Énumerer d'abord, tester ensuite On illustre la "mauvaise" approche du problème par le prédicat `coloriageET`, ET abrégeant énumérer puis tester. Cette "mauvaise" approche était en fait l'approche usuelle suivie en prolog avant l'intégration des contraintes dans prolog.

```
/* coloriageET(A,B,C,D,E)
     résultat : A,B,C,D,E ont des valeurs entières entre 1 et 3 telles
     que 2 régions voisines n'ont pas les mêmes valeurs */
   coloriageET(A,B,C,D,E):-
        % on enumère
        member(A,[1,2,3]),
        member(B,[1,2,3]),
        member(C,[1,2,3]),
        member(D,[1,2,3]),
        member(E,[1,2,3]),
        % on teste
        A=\=B,A=\=C,A=\=D,
        B=\=C,B=\=E,
        C=\=D,C=\=E,
        D=\=E.
```

La requête `coloriageET(A,B,C,D,E)` va énumérer A = 1, B = 1 puis faire inutilement toutes les énumérations des valeurs de C, D, E avant de commencer à trouver le début d'une solution acceptable A = 1, B = 2. L'arbre d'exploration jusqu'à la première solution est très grand, comme vous pouvez l'expérimenter.

Contraindre d'abord, énumérer ensuite La bonne est approche est représenter par le prédicat `coloriageCE`, CE abrégeant contraindre puis énumérer.

```
/* coloriageCE(A,B,C,D,E)
     résultat : A,B,C,D,E ont des valeurs entières entre 1 et 3 telles
     que 2 régions voisines n'ont pas les mêmes valeurs */
   coloriageCE(A,B,C,D,E):-
        % On impose que les variables A,B,C,D,E valent 1, 2 ou 3
        fd_domain([A,B,C,D,E],1,3),
        % On imppose que deux régions voisines soient de couleurs différentes
        A#\=B,A#\=C,A#\=D,
        B#\=C,B#\=E,
        C#\=D,C#\=E,
        D#\=E,
        % On énumère les valeurs des variables dans l'ordre A,B,C,D,E
        fd_labeling([A,B,C,D,E]).
```

Comment interfèrent les contraintes et l'énumération au moment de l'exécution de `fd_labelling` :
Prolog choisit A = 1, cela interdit 1 dans les domaines de B,C,D
Prolog choisit B = 2, cela interdit 2 dans les domaines de C, E
Prolog choisit C = 3 (1 et 2 sont interdits par les contraintes), cela interdit 3 dans domaine de D et E
Les valeurs de D et E sont alors déterminées : D = 2, E = 1.

Les contraintes de gprolog Sur cet exemple, on a vu

- la contrainte `fd_domain` qui permet de contraindre les variables à être dans un intervalle entre deux entiers positifs ou nuls.
- la contrainte `#=` qui impose que deux expressions arithmétiques aient des valeurs distinctes
- la contrainte `fd_labeling` qui force l'énumération des variables dans leurs domaines

Les lettres `fd` abrège finite domain. Dans la version officielle, `gprolog` ne permet de poser des contraintes que sur les variables dont le domaine est un ensemble fini d'entiers positifs ou nuls.

7.3 Puzzle crypto-arithmétique

On pose l'opération

```
 SEND
+MORE
-----
MONEY
```

Le but du jeu consiste à affecter un chiffre à chaque lettre (deux lettres distinctes étant affectées à des chiffres distincts) de sorte que l'addition soit vérifiée et que S et M ne soient pas affectés par 0.

Comme précédemment, on peut rechercher la solution par programme avec la mauvaise méthode énumerer d'abord, tester après et avec la bonne méthode contraindre d'abord, énumérer progressivement après.

Énumérer d'abord, tester ensuite

```
/* sendmoryET([S,E,N,D,M,O,R,Y])
   résultat : les valeurs des variables sont des chiffres qui rendent correcte
     l'opération SEND+MORE=MONEY  */

   sendmoryET([S,E,N,D,M,O,R,Y]):-
        % On enumére les affectations possibles
        L09 =[0,1,2,3,4,5,6,7,8,9],
        member(S,L09),
        member(E,L09),
        member(N,L09),
        member(D,L09),
        member(M,L09),
        member(O,L09),
        member(R,L09),
        member(Y,L09),
        % Les variables sont instanciées
        % On teste que toutes les lettres sont associées à des chiffres distincts
        differents([S,E,N,D,M,O,R,Y]),
        % La somme est correcte
        M =\=0, S =\=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.

% differents(L)
% précondition : L est une liste de nombres
% résultat : yes ssi les éléments de L sont distincts
differents([]).
differents([E|S]):- non_element(E,S),differents(S).

% non_element(E,L)
% précondition : E est un nombre et L une liste de nombres
% résultat : yes ssi E n'est pas élément de L
non_element(_,[]).
non_element(E,[F|S]):- E=\=F, non_element(E,S).
```

Puisqu'il y a 8 lettres auxquelles on peut affecter 10 valeurs, les tests seront effectuées au plus 10^8 fois. En comptant 10^{-5} secondes par test, cela fait 10^3 secondes de calculs, donc au moins 15 minutes. Le test effectif a donné la seule solution en *38 minutes*.

Contraindre d'abord, énumérer ensuite

```
/* sendmoryET([S,E,N,D,M,O,R,Y])
   résultat : les valeurs des variables sont des chiffres qui rendent correcte
     l'opération SEND+MORE=MONEY  */

   sendmoryCE([S,E,N,D,M,O,R,Y]):-
        % Definitions du domaine des variables
        fd_domain([S,E,N,D,M,O,R,Y],0,9),
        % Pose des contraintes
        % Toutes les valeurs des variables sont distinctes
        fd_all_different([S,E,N,D,M,O,R,Y]),
        % M et S ne valent pas 0
        M #\=0, S #\=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,
        % La somme est correcte
        Send + More #= Money,
        % Par fd_labelingff (first fail), énumération des solutions en commençant par la variable de
        % plus petit domaine
        % et s'il y a deux variables de plus petit domaine, en prenant celle
        % la plus à gauche
        fd_labelingff([S,E,N,D,M,O,R,Y]).
```

L'énumération choisie n'est pas nécessairement la meilleure et gprolog permet des choix différents que l'on trouve dans la documentation de `fd_labeling`. Dans de gros problèmes combinatoire, l'ordre d'énumération est capitable dans l'obtention rapide de solutions.

On se fatiguera pas à chercher le meilleur ordre possible, car l'ordinateur, qui avait trouvé la réponse en *38 minutes*, la trouve maintenant en *2 ms*.

7.4 Les reines sur un échiquier

Un problème test pour les contraintes est celui consistant à trouver les positions que peuvent occuper 8 reines sur un échiquier de 8×8 cases de sorte qu'aucune reine ne puisse en prendre une autre. En généralisant à N reines posées sur un échiquier $N\times N$ on peut alors «mesurer» les performances du résolveur de contraintes.

Aux échecs une reine peut prendre sur la ligne, la colonne ou les deux diagonales sur lesquelles elle est placée. On peut coder le problème avec deux variables par reine indiquant la ligne et la colonne de la reine. Mais comme deux reines ne peuvent pas occuper la même colonne, on peut ne prendre qu'une variable pour chaque colonne, la variable indiquant la position de la reine dans la colonne. C'est un principe général à suivre pour tout problème combinatoire : prendre un codage qui diminue le nombre de variables et qui code une ou des contraintes du problème.

On généralise immédiatement le problème à N reines

```
/* reines(L,N)
   précondition : N entier >= 0
   résultat L =  liste de longueur N,
   l'élément en position I de L est la ligne de la reine en colonne I,
   les reines de L ne sont pas en prises */
```

```
reines(L,N) :-
  length(L,N),
  % on précise le domaine des variables de la liste L %
  fd_domain(L,1,N),
  % on POSE les contraintes exprimant que les reines ne sont pas en prises
  non_en_prises(L),
  % on énumère
  fd_labeling(L).
```

Les reines de la liste vide ne sont pas en prise. Les reines de la liste [X|S] ne sont pas en prise si la reine à la ligne X ne prend pas les reines de S (supposées être sur les colonnes suivantes) et si les reines de S ne sont pas en prises entre elles.

On utilise le prédicat auxilliaire `ne_prend_pas(X,S,D)`, qui réussit si la reine à la ligne X ne prend pas les reines de la liste S placées (D-1) colonnes après la reine X. Précisons ce prédicat. Supposons que la reine X est dans la colonne I, les reines de S sont alors dans les colonnes consécutives I+D, I+D+1, ...etc. Soit Y la reine en colonne I+D.

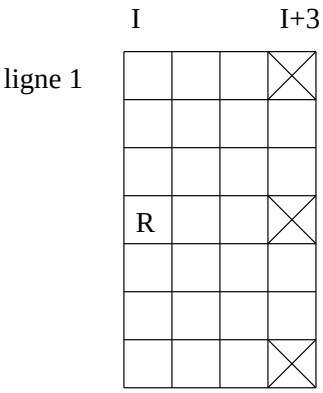
Pour que les reines X et Y soient dans des lignes différentes, il faut poser

`Y #\=X`

Pour que les reines X (colonne I) et Y (colonne I+D) soient dans des diagonales différentes, il faut poser

`Y #\= X - D, Y #\= X + D`.

Représentons la situation pour X = 4, D = 3, les cases interdites sont les cases des lignes 4, 4-3, 4+3.



D'où les programmes correspondant aux deux prédicats `non_en_prises` et `ne_prend_pas`.

```
/* non_en_prises(L)
  précondition : L est une liste de variables à domaine fini
  résultat : les contraintes telles que L est une liste de reines
  non_en_prises */
non_en_prises([]).
non_en_prises([X|S]) :- ne_prends_pas(X,S,1),non_en_prises(S).

/* ne_prends_pas(X,L,D)
   précondition : L est une liste de variables à domaine fini
       X est une variable à domaine fini
       D est un entier
   résultat : les contraintes telles que  X ne prend les reines
       de la liste L placées (D-1) colonnes après D celle de X */
ne_prends_pas(_, [], _).
ne_prends_pas(X, [Y | S], D) :-
   Y #\=X, Y #\= X - D, Y #\= X + D,
   Dplus1 is D+1, ne_prends_pas(X, S, Dplus1).
```

7.5 Classification des contraintes

7.5.1 Définition des domaines

`fd_domain(LVars,Bi,Bs)` Lvars est une liste de variables, la réussite de la requête lie chacune des variables de la liste à l'intervalle Bi..Bs bornes comprises. Bi et Bs doivent être des entiers.

```
?- fd_domain([X, Y],1, 4).
X = _#3(1..4)
Y = _#25(1..4)
```

La réponse doit être interprétée ainsi : `X` et `Y` sont des variables dont la valeur est comprise entre 1 et 4.

7.5.2 Contraintes arithmétiques

Principe des réductions de domaine Commençons par des exemples.

```
?- fd_domain([X,Y],0,7),X * Y #= 6, X + Y #=5, X #< Y.
X = 2 Y = 3
```

Le domaine initial des variables X et Y est fixé, puis il est réduit au fur et à mesure de l'accumulation des contraintes.

On n'indique pas l'algorithme qui réduit les domaines, mais on donne des indications sur son principe de fonctionnement.

1. D'après la première contrainte X et Y sont entre 0 et 7.
2. D'après la deuxième contrainte X et Y ne sont pas nuls donc valent au moins 1, d'après cette deuxième constrainte X et Y valent au plus 6.
3. D'après la troisième contrainte et les bornes précédentes, X vaut au plus 5 moins la valeur minimum de Y, donc au plus 4, et de même pour Y.
4. *Réutilisons* à nouveau la deuxième contrainte, X est au moins égal à 6/4, donc X vaut au moins 2. *Réutilisons* à nouveau la troisième contrainte, puisque X vaut au moins 2, Y vaut au plus 3. En échangeant les rôles de X et Y, on voit qu'après les 3 premières contraintes X et Y sont entre 2 et 3. Donc la dernière contrainte donne la solution

En présence d'un ensemble de contraintes, l'algorithme utilise chaque contrainte en effectuant les réductions de domaine, jusqu'au moment où plus *aucune* contrainte ne peut reduire le domaine des variables.

Réduction partielle des domaines Le signe caractéristique des contraintes à domaine fini est le `#`.
La liste des prédicats utilisables est : `#=, #\=, #<, #=<, #>, #>=`
Ces prédicats peuvent être écrits entre leurs opérandes qui sont des expressions arithmétiques et nous avons déjà vus plusieurs exemples de leurs utilisations.

On en présente un exemple surprenant qui met en évidence que gprolog utilise deux représentations des domaines finis.

```
?- fd_domain([X],1,1000), X#\=3.
X = _#3(1..2:4..127@)
```

Après la première contrainte, la valeur de X est comprise entre 1 et 1000. Cet intervalle est représenté par un couple d'entiers. Après la deuxième contrainte, la valeur de X doit être différente de 3. `gprolog` représente alors X par un vecteur de 128 bits. Le signe `@` indique que des valeurs ont été perdues suite à ce changement de réprésentation.

En résumé, il y a deux représentations des domaines finis

1. par un intervalle entre Bi et Bs avec $0 \leq Bi$ et $Bs \leq 2^{28} - 1$
2. par un vecteur de bits d'indice entre 0 et une valeur vector_max. Initialement vector_max = 127. Il est possible par la contrainte fd_set_vector_max(+integer) de changer la valeur de vector_max, ce qui est le premier moyen d'éviter de perdre des valeurs.

Comment chaque contrainte est-elle utilisée ? Soit X une variable de domaine DX, Y une variable de domaine DY, l'exécution de la contrainte r(X,Y) *peut* enlever de DX les valeurs v telles que r(v,Y) n'est vérifiée pour aucune valeur de Y dans le domaine DY et peut réduire de même le domaine DY. Par exemple :

```
?- fd_domain([X,Y],1,10), X#\=2, X+Y#=5.
X = _#3(1:3..4)
Y = _#25(1..4)
```

Noter que la réduction opérée par la dernière contrainte est *partielle*. Puisque X est différent de 2, d'après la dernière contrainte, Y ne peut pas être égal à 3, mais cette valeur n'est pas enlevée du domaine de Y.

Les contraintes de réductions partielles (avec un seul #) ne permettent que la réduction des intervalles entre les valeurs minimum et maximum des variables.

Réduction complète des domaines La liste des prédicats utilisables est : `#=#, #\=#, #<#, #=<#, #>#, #>=#`
Reprenons l'exemple précédent avec la réduction complète :

```
?- fd_domain([X,Y],1,10), X#\=2, X+Y#=#5.
X = _#3(1:3..4)
Y = _#25(1..2:4)
```

Comparons sur d'autres exemples la réduction partielle et complète

```
| ?- fd_domain([X,Y],1,100),X*Y #= 51.
X = _#3(1..51)
Y = _#25(1..51)
yes
| ?- fd_domain([X,Y],1,100),X*Y #=# 51.
X = _#3(1:3:17:51)
Y = _#25(1:3:17:51)
yes
```

La réduction complète prend plus de temps que la réduction partielle. Si votre problème n'a besoin que d'intervalles, il est inutile de l'utiliser.

7.5.3 Information sur les domaines

1. `fd_min(X, N)`
   X doit être une variable associée à un domaine fini
   N est la valeur minimale du domaine de X
2. `fd_max(X, N)`
   X doit être une variable associée à un domaine fini
   N est la valeur maximale du domaine de X
3. `fd_size(X, N)`
   X doit être une variable associée à un domaine fini
   N est le nombre d'éléments du domaine de X
4. `fd_dom(X, Values)`
   X doit être une variable associée à un domaine fini
   Values est la liste des valeurs du domaine de X

7.5.4 Contraintes d'énumération

1. `fd_labeling(Vars)`
   Vars doit être une liste de variables associées à des domaines finis
   Spécification : affecte une valeur à chaque variable de la liste Vars.

```
?- fd_domain([X],1,2),fd_labeling([X]).
X = 1 ?;
X = 2
yes
```

2. `fd_labelingff(Vars)`
   Vars doit être une liste de variables associées à des domaines finis
   Spécification : affecte une valeur à chaque variable de la liste Vars en choisissant d'abord la variable de plus petit domaine et entre deux variables de même plus petit domaine, celle la plus à gauche.

   ```
   ?- fd_domain([X],1,3), fd_domain([Y],1,2),fd_labelingff([X, Y]).
   X = 1 Y = 1 ? a
   X = 2 Y = 1
   X = 3 Y = 1
   X = 1 Y = 2
   X = 2 Y = 2
   X = 3 Y = 2
   ```

   Attention la variable `Y`, qui a le plus petit domaine, est bien énuméré d'abord : pour `Y = 1`, `X` prend les valeurs `1, 2, 3` puis on recommence avec `Y = 2`.

Le prédicat `fd_labeling` peut être utilisé avec des options qui modifient l'ordre d'énumération. L'ordre d'énumération est très important dans la résolution des problèmes combinatoires, puisque, par exemple, le choix d'une valeur pour une variable qui entre dans beaucoup de contraintes, va conduire plus vite à l'échec ou au succés des contraintes comportant cette variable.

Nous renvoyons à la documentation `gprolog` pour compléter les informations sur les contraintes. En particulier nous n'avons pas évoqué ci-dessus les contraintes dites symboliques (comme `fd_all_different` qui figure dans les exemples) qui posent des contraintes sur des listes de variables.

8 Coupure

Lorsque prolog exécute une requête, en général, il en recherche toutes les réponses, en appliquant au premier but de la requête, tous les faits unifiables au but et toutes les règles dont la tête est unifiable au but.

Cette application exhaustive des faits et règles peut conduire à des programmes inefficaces. Prolog inclut un prédicat, la *coupure*, permettant d'éviter cette application exhaustive .

Ce prédicat a deux intérêts principaux, il permet

1. d'exprimer qu'un but ne peut pas être déduit d'un programme.
2. d'empêcher l'application de règles qui conduiront certainement à des échecs, donc de rendre plus efficace les programmes

Cependant utiliser la coupure est *dangereux* car on perd des réponses aux requêtes. Aussi il faut soit éviter de l'utiliser, soit justifier avec le plus grand soin les raisons qui vous obligent à l'utiliser.

8.1 Définition et effets de la coupure

La coupure est un atome qu'on peut utiliser en écrivant les règles Prolog. Par exemple :

```
p(X) :- b(X),c(X),!,d(X),e(X).
```

est une règle bien écrite. Tout d'abord la coupure est un but qui réussit. Mais, et c'est le plus important, l'exécution de la coupure a des effets de bord. Supposons qu'un but utilise cette règle. Appelons ce but, le but parent. La coupure supprime les choix possibles restant entre l'exécution du but parent et l'exécution de la coupure, autrement dit tous les choix restant pour le but `p(X)` et les deux buts `b(X), c(X)` qui précédent la coupure.

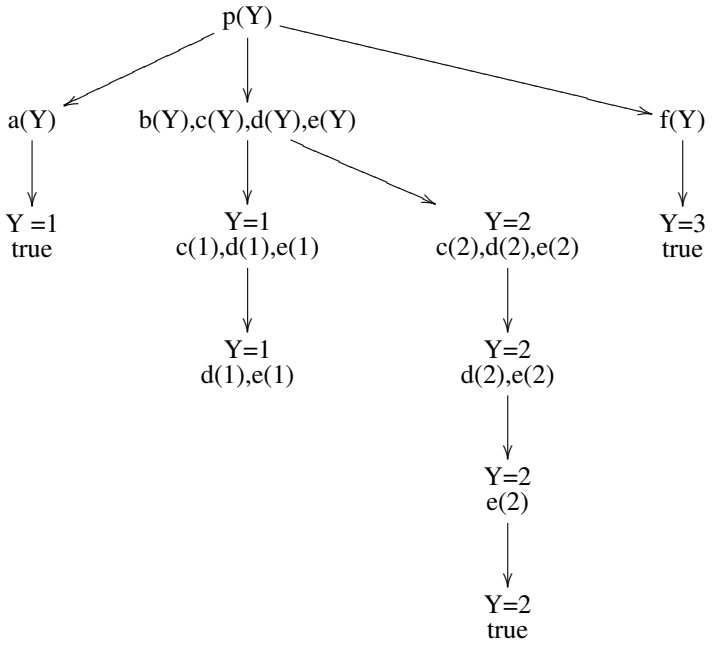
Observons ce qui se passe sur un exemple. Considerons d'abord le programme *sans* coupure suivant :

```
p(X) :- a(X).
p(X) :- b(X),c(X),d(X),e(X).
p(X) :- f(X).
a(1).
b(1).
b(2).
c(1).
c(2).
d(2).
e(2).
f(3).
```

Si on pose la requête `p(Y)` on obtient les réponses suivantes :

```
Y = 1?;
Y = 2?;
Y = 3
no
```

Donnons l'arbre d'exploration de cette requête. L'arbre montre comment Prolog trouve les 3 solutions.



Maintenant, on ajoute une coupure dans la deuxième règle :

```
p(X) :- b(X),c(X),!,d(X),e(X)
```

Si on pose la requête `p(Y)` on obtient les réponses suivantes :

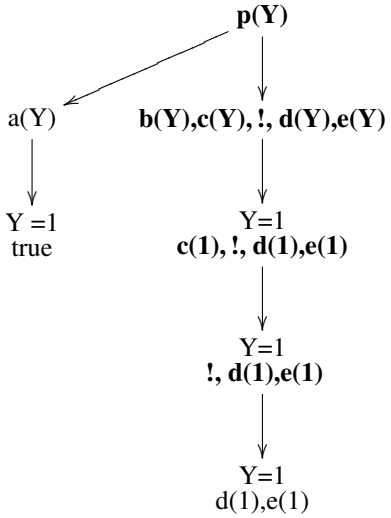
```
Y = 1;
no
```

Que s'est-il passé ?

1. Le but `p(Y)` a été tout d'abord unifié avec la première règle, ce qui a donné un nouveau but `a(Y)`. Ce nouveau but a été unifié avec le fait `a(1)` ce qui a donné comme précédemment la solution `Y = 1`.
2. Puis `p(Y)` a été unifié avec la deuxième règle, ce qui a donné la nouvelle requête `b(Y),c(Y),!,d(Y),e(Y)`. Le premier but a été unifié avec le fait `b(1)`, ce qui donne la nouvelle requête `c(1),!,d(1),e(1)`. Puis le but `c(1)` est effacé grâce au fait `c(1)`, ce qui donne la nouvelle requête `!,d(1),e(1)`.

3. La coupure est alors exécutée avec succès, ce qui supprime les choix restants entre l'appel du but qui a introduit la coupure et l'exécution de cette coupure. La nouvelle requête est alors `d(1),e(1)`.
4. Puisqu'aucune tête de règle n'est unifiable avec `d(1)`, la requête échoue. Puisqu'il n'y a plus de choix possible en revenant en arrière, on n'obtient une seule solution.

Observons l'arbre d'exploration de la même requête en présence de la coupure :



Tous les choix restants des sommets en gras sont supprimés.

8.2 Exemples d'utilisation

8.2.1 Exprimer une négation

Le prédicat non unifiable On a dit que la requête `X \= Y` réussit si et seulement si la requête `X = Y` ne réussit pas. On peut programmer en gprolog ce prédicat. On le renomme `non_unifiable`, car gprolog interdit de rédéfinir un prédicat prédéfini, et on le définit pas les règles :

```
non_unifiable(X,Y) :- X = Y, !, fail.
non_unifiable(_,_).
```

Effectuons la requête `non_unifiable(X,Y)`. Si `X` et `Y` sont unifiables, la première règle appliquée, va exécuter la coupure (donc empêcher l'emploi de la deuxième règle) et le but `fail` échoue comme son nom l'indique. Si `X` et `Y` ne sont pas unifiables, la première règle échoue *avant* la coupure, et la deuxième règle réussit. La coupure a exprimé une négation.

Le prédicat nonvar La requête `nonvar(X)` réussit si et seulement si `var(X)` échoue. On le renomme `nonvariable` afin de pouvoir le définir en gprolog et on le définit par les règles :

```
nonvariable(X) :- var(X),!, fail.
nonvariable(_).
```

Effectuons la requête `nonvariable(X)`. *Si* `X` n'est pas instanciée, la coupure est exécutée, l'atome `fail` cause un échec "définitif", puisque la coupure empêche le choix de la deuxième règle, *sinon* `nonvariable(X)` réussit. La coupure a exprimé une négation.

Le prédicat list On peut tester si un terme est une liste au moyen du prédicat `list`, que nous désignons en français par `est_liste`.

Rappelons que toute liste est

1. la liste vide notée `[]`.
2. un terme `[E | S]` où `E` est un terme, `S` est une liste

La requête `est_liste(T)` doit réussir si et seulement si la valeur du terme `T` est une liste. Donc elle doit échouer si `T` est une variable non instanciée, car la valeur de cette variable est alors elle-même. Dans le cas contraire, il suffit de faire appel à la définition inductive ci-dessus. Le prédicat est défini par le programme :

```
est_liste(L) :- var(L),!,fail.
est_liste([]).
est_liste([_| S]) :- est_liste(S).
```

Effectuons la requête `est_liste(L)` avec `L` instanciée. Le but `var(L)` échoue, et prolog essaie les deux autres règles. Effectuons la requête `est_liste(L)` avec `L` *non* instanciée. Le but `var(L)` réussit, le but `!`, c'est-à-dire la *coupure*, réussit, ce qui coupe les choix restants du but `est_liste(L)`, puis le but `fail` échoue, comme son nom l'indique.

8.2.2 Optimiser un programme

Expression conditionnelle Considérons la spécification suivante :
`max(X,Y,Z)`
condition d'utilisation : `X` et `Y` sont des nombres. résultat : `Z` est le maximum de `X` et `Y`.
On souhaite le comportement suivant :

```
?- max(2,3,2).
no
?- max(2,3,3).
yes
?- max(2,3,X).
X = 3
yes
```

Construction d'une solution : `max(X,Y,Z)` vrai si `X =< Y` alors `Z = Y` sinon `Z = X`
Solution :

```
max(X,Y,Z) :- X =< Y, Z = Y.
max(X,Y,Z) :- X > Y, Z = X.
```

La solution a l'inconvénient de mal exprimer la construction «sinon», autrement dit de comparer deux fois `X` et `Y`.
La coupure permet d'exprimer cette construction :

```
max(X,Y,Z) :- X =< Y,!, Z = Y.
max(X,_,X).
```

Peut-on faire mieux et gagner une unification avec le programme :

```
max(X,Y,Y) :- X =< Y,!.
max(X,_,X).
```

Non, car ce programme est incorrect, en effet il déclare que le maximum de `X` et `Y` est `Y`, avant d'avoir comparer ces deux nombres. Essayons ce *tout dernier* programme :

```
?-max(2,3,2).
yes
```

La tête de la première règle n'est pas unifiable avec la requête, et la deuxième règle donne une réponse incorrecte au regard de la spécification.
Pour conclure cet exemple, on recommande de programmer de façon sûre, puis d'optimiser le programme obtenu avec prudence, en respectant la spécification.

Factorielle encore plus rapide Dans les travaux pratiques, on a proposé une version déjà rapide du calcul de la fonction factorielle. Rappelons la spécification et le programme obtenu.
`fact(N,R)`
condition d'utilisation : `N` est un entier positif ou nul
résultat : R = eval(N !)
On avait obtenu le programme :

```
fact(0,1).
fact(1,1).
fact(N,R):- N > 1, factAcc(N,N,R).
factAcc(2,A,A).
factAcc(N,A,R) :- N > 2, NM1 is N-1, A1 is A * NM1, factAcc(NM1,A1,R).
```

Comme sur l'exemple précédent, on voit que les règles traduisent (mal) une expression conditionnelle :
`fact(N,R)` si N = 0 et R = 1 ou alors si N = 1 et R = 1 ou alors N > 1 et factAcc(N,N,R).
Il en est de même des règles pour le prédicat factAcc.

En ajoutant des coupures, on traduit le fait que pour toute valeur de N, une seule des trois règles de fact est choisie, ce qui donne pour fact le programme :

```
fact(0,1):- ! .
fact(1,1):- ! .
fact(N,R):- factAcc(N,N,R).
```

Pour optimiser factAcc sans commettre d'erreur, il faut comme pour l'exemple précédent, réécrire le programme de façon à ce que le choix de la règle ne dépende que de la valeur du premier argument :

```
/* factAcc(N, A, R)
   condition d'utilisation : N >= 2
   résultat : R = eval(A*(N-1)!) */
   factAcc(2,A,R) :-  R = A.
   factAcc(N,A,R) :- N > 2, NM1 is N-1, A1 is A * NM1, factAcc(NM1,A1,R).
```

Puisque factAcc n'est exécuté que lorsque N >= 2, on voit que le programme peut être optimisé en :

```
factAcc(2,A,R) :-  !, R = A.
factAcc(N,A,R) :-  NM1 is N-1, A1 is A * NM1, factAcc(NM1,A1,R).
```