Université d'Aix-Marseille - L3 info - Logique : introduction à Prolog

Prolog, créé à Marseille par Colmerauer et Roussel au début des années 70, est un langage de *programmation logique*, qui est une forme particulière de *programmation déclarative*. Il est basé sur la résolution de clauses de Horn de la logique des prédicats. Ces notions ayant été introduites précédemment pendant le cours de logique, nous les considérerons comme acquises lors de cette présentation (très succinte) de Prolog.

1 La programmation déclarative

La programmation déclarative est un paradigme de programmation qui se distingue très nettement de la programmation impérative (C, Python, etc), de la programmation objet (C++, Java, etc) et de la programmation fonctionnelle (Lisp, Caml, Haskell, etc). Un programme déclaratif est une suite de déclarations qui constitue une base de connaissances dont on ne présuppose pas forcément l'utilisation qu'il en sera fait : *on y affirme ce qui est vrai mais on ne dit pas ce qu'il faut en faire*. En programmation déclarative, on donne précisément dans un certain formalisme (un langage de spécification) les propriétés/caractéristiques des choses dont on veut faire le traitement, puis un algorithme général (capable de résoudre tout problème spécifié dans ce formalisme) se charge de faire lui-même le traitement et de fournir le résultat. En un certain sens, l'idée fondamentale est de ne plus avoir à programmer au sens où on l'entendait jusqu'alors, c'est-à-dire, de ne plus avoir à décrire le calcul à effectuer pour obtenir un résultat.

La programmation déclarative entre dans le champ de l'intelligence artificielle (IA), plus précisément de son aspect "raisonnement automatique" dont l'objet est de simuler le raisonnement humain pour résoudre les problèmes informatiques. Ce n'est pas une forme de programmation pour faire spécifiquement de l'IA, puisqu'on peut résoudre les mêmes problèmes que dans les langages impératifs ou fonctionnels, mais c'est une application de l'IA à la programmation : c'est l'ordinateur qui trouve le résultat sans qu'on lui ait fourni la méthode (l'algorithme) mais uniquement les connaissances du domaine suffisantes pour produire ce résultat.

Pour obtenir une information pouvant être déduite de cette base de connaissances, on a besoin d'un *moteur d'inférence* et d'une *requête*. Le moteur d'inférence met en oeuvre une procédure d'extraction de l'information à partir de la requête mais contrairement aux bases de données, l'information extraite n'est pas forcément explicitée par la base, elle peut être déduite soit à partir des données explicitement présentes, soit grâce aux règles de production qui s'appliquent récursivement à ces données (cf figure 1).

Par exemple, un programme déclaratif pourrait être celui qui déclare "Socrate est un homme" et "Tous les hommes sont mortels". Les requêtes pourraient être en substance :

- "Socrate est-il un homme ?" et la réponse serait "oui" car c'est une donnée explicite de la base.
- "Qui est un homme ?" et la réponse serait "Socrate" car c'est extrayable d'une donnée explicite de la base.
- "Qui est mortel ?" et la réponse serait "Socrate" car ce serait une conséquence des deux déclarations (le moteur d'inférence a fait la déduction).

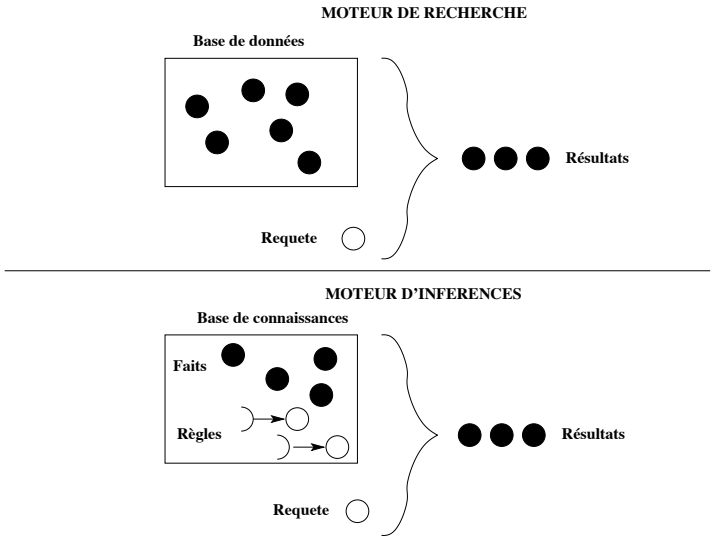

FIGURE 1 – Comparaison base de données/base de connaissances

2 La programmation logique

La programmation logique permet d'écrire des programmes déclaratifs en décrivant des connaissances dans le cadre de la logique du premier ordre.

Dans ce cadre, l'exemple précédent s'exprimerait par les formules "homme(socrate)" et "$\forall$x, homme(x) $\Rightarrow$ mortel(x)". Les requêtes ressembleraient à :

— "homme(socrate) ?" et la réponse serait "vrai".

— "$\exists$x, homme(x) ?" et la réponse serait "socrate".

— "$\exists$x, mortel(x) ?" et la réponse serait "socrate" car "mortel(socrate)" serait une conséquence logique des deux formules.

Prolog est un langage de programmation logique qui permet une certaine forme d'expression logique des connaissances et utilise un moteur d'inférence particulier, que nous allons décrire plus loin. La puissance de Prolog en terme de calculabilité est celle de la machine de Turing (ou du lambda-calcul). Le grand avantage de Prolog est la rapidité et la simplicité avec laquelle on peut écrire des programmes mais son gros inconvénient est que le temps de calcul correspondant à une requête peut être beaucoup plus long que dans les autres paradigmes de programmation si on ne maîtrise pas la façon dont fonctionne le moteur d'inférence. En effet, comme ce n'est pas nous qui indiquons comment le résultat doit être calculé, il est fréquent que la méthode utilisée par Prolog soit plus lente que l'algorithme le plus efficace qui existe.

3 La syntaxe de Prolog

Un programme Prolog est un ensemble de *clauses* (de Horn). Une clause peut être un *fait* (une déclaration inconditionnelle) ou une *règle* (une déclaration conditionnelle). Chaque clause met en relation des *atomes logiques*. Chaque atome logique est constitué d'un *prédicat* qui porte sur un certain nombre de *termes*, qui peuvent être des *variables*, des *termes atomiques* ou des *termes*

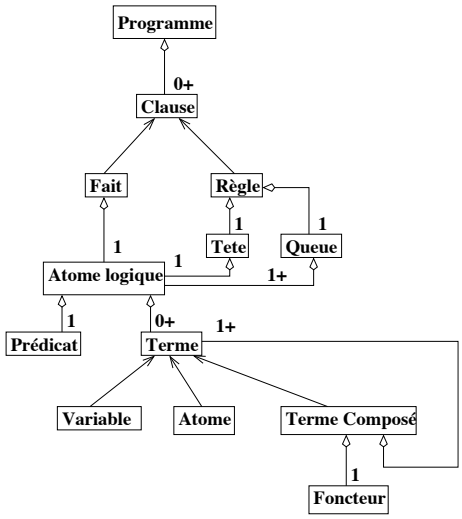

FIGURE 2 – Récapitulatif des liens entre les différentes unités syntaxiques de Prolog

composés.

Un *terme atomique* correspond à une constante dans la logique des prédicats. En Prolog, ça peut être un identificateur (on parle alors d'*atome*), un entier, une chaîne de caractères. Syntaxiquement, un atome est une suite de lettres, de chiffres et de _ qui commence par une minuscule ou _. Ex : `dupont`, `marseille`.

Une *variable* correspond à la même notion qu'en logique des prédicats : il s'agit d'un objet inconnu. Syntaxe Prolog : une variable est une suite des lettres et de chiffres qui commence par une majuscule. Ex : `Nom`, `Ville`.

Un *terme composé* correspond à la notion de fonction en logique des prédicats. Il s'agit d'un terme composé de plusieurs autres termes. Il est constitué d'un *foncteur* (ou *symbole de fonction*) qui porte sur une suite d'objets qui le compose. Syntaxe Prolog : un terme composé est un identifiant suivi d'une suite de termes ou de variables séparés par des virgules et encadrés par des parenthèses. Ex : `nom(jean, dupont)`, `personne(nom(jeanne, martin), Age)`.

Un *atome logique* est la plus petite unité qui peut être évaluée à vrai ou faux. Il est composé d'un prédicat qui porte sur des termes, comme dans la logique des prédicats (mais si le prédicat ne porte sur aucun terme, alors il correspond à une proposition). Syntaxe Prolog : la même que pour les termes composés. Ex : `mange(lapin, carotte)`.

Une *clause* correspond à une clause de Horn en logique des prédicats, c'est-à dire une disjonction d'atomes logiques dont un seul peut être positif. Une clause $p_1 \vee \neg p_2 ... \vee \neg p_n$ peut se réécrire $p_2 \wedge p_3 \wedge ... \wedge p_n \Rightarrow p_1$. En Prolog, on l'écrira sous la forme : p_1 `:-` p_2`, ...,` p_n`.` où les p_i sont des atomes logiques de Prolog. p_1 est la *tête de la clause* et p_2`, ...,` p_n`.` est la *queue de la clause*. S'il n'y a qu'un seul atome logique (positif) alors la clause est un fait et s'écrit : p_1`.` (un atome logique suivi d'un point). L'apparition d'une variable dans un terme implique qu'elle est quantifiée universellement dans la clause. Ex : `beau_parent(X, Y) :- parent(X, Z), maries(Z, Y).` équivaut à : $\forall X, Y, Z : parent(X, Z) \wedge maries(Z, Y) \Rightarrow beau\_parent(X, Y)$

Etant donné un programme Prolog (un ensemble de clauses), on peut obtenir un résultat à

partir d'une requête formulée sous la forme d'une conjonction d'atomes logiques

`p1 , ... , pn .`. Le moteur d'inférence va indiquer avec quelles valeurs instancier les variables de cette conjonction d'atomes logiques pour qu'elle soit une conséquence logique de l'ensemble des clauses du programme. S'il n'y a pas de variable, le moteur va simplement indiquer si la requête est ou pas une conséquence logique. Avec ce programme :

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

les requêtes `?- homme(socrate).` ou `?- mortel(socrate).` retournent `yes`, les requêtes `?- homme(X).` ou `?-mortel(X).` retournent `X=socrate` et les requêtes `?- homme(aristote).`, `?- mortel(platon).` ou `?- philosophe(X).` retournent `no`.

4 Sémantique dénotationnelle de Prolog

Avant de présenter la façon dont Prolog répond effectivement à une requête sur un programme, nous allons définir la *dénotation d'un programme*, c'est-à-dire, l'ensemble (potentiellement infini) des atomes logiques que représente le programme.

On rappelle qu'une *substitution* σ est une fonction d'un ensemble de variables vers un ensemble de termes. Ex : { X → a, Y → f(a,Z), U → V}. On peut étendre l'application t^σ d'une substitution σ à un terme quelconque t ainsi :

— si t est une variable, on applique normalement σ.
— si t est un terme atomique alors $t^\sigma = t$.
— si t est un terme composé $f(t_1, ..., t_n)$ alors $t^\sigma = f(t_1^\sigma, ..., t_n^\sigma)$.

Ex : $f(b, X, Y, U)^\sigma = f(b, a, f(a, Z), V)$.

t^σ est appelé *instance* de t. t_1 est plus général que t_2 s'il existe une substitution telle que t_2 est l'instance de t_1.

Un *unificateur* σ de deux termes t_1 et t_2 est une substitution telle que $t_1^\sigma = t_2^\sigma$. Ex : {X → h(U), Y → b, V → a} est un unificateur des termes f(X, g(a, Y)) et f(h(U), g(V, b)).

Un *unificateur le plus général* σ de deux termes t_1 et t_2 est un unificateur de t_1 et t_2 telle que tout autre unificateur ρ de t_1 et t_2 est tel que t_1^σ est plus général que t_1^ρ. L'algorithme de Robinson permet de déterminer l'unificateur le plus général de 2 termes ou de déterminer qu'il n'y en a pas (cf figure 3).

La *dénotation* d'un programme Prolog est l'ensemble des atomes logiques qui peuvent être déduits par une suite d'applications des règles à partir de ses faits. Plus précisément, on le définit inductivement ainsi :

— l'ensemble des faits donnés par le programme sont des atomes logiques qui font partie de la dénotation du programme.
— si les atomes logiques a_1, ... a_n font partie de la dénotation, que la règle c :- b_1, ..., b_n appartient au programme et que σ est le plus grand unificateur de a_1 avec b_1, a_2 avec b_2, ..., a_n avec b_n alors c^σ appartient à la dénotation du programme.

Poser une requête sous la forme d'une conjonction d'atomes logiques r_1, ..., r_k à un programme Prolog revient à demander s'il existe un unificateur (le plus général) σ tel que r_1^σ, ... et r_k^σ font partie de la dénotation du programme, et quel est cet unificateur.

On peut interpréter la dénotation d'un programme Prolog dans le cadre de la logique des prédicats. En effet, un fait est une clause positive unaire et une règle c :- b_1, ..., b_n peut se réécrire

```
fonction upg(t1, t2)
   si t1 = t2 alors retourne ∅
   si t2 est une variable alors permuter t1 et t2
   si t1 est une variable alors
      si t2 est un sous-terme de t1
         retourne ECHEC
      sinon
         retourne { t1 → t2}
   sinon
      si t1 = f(u1, ..., un) et t2 = f(v1, ..., vn)
         σ → ∅
         pour i de 1 a n
            ρ → upg( u_i^σ, v_i^σ)
            si ρ = ECHEC alors retourne ECHEC
            σ → ρ o σ
      sinon
         retourne ECHEC
```

FIGURE 3 – Algorithme de Robinson

en une clause CNF $c \vee \neg b_1 \vee ... \vee \neg b_n$. Appliquer une règle Prolog consiste à faire la résolution entre cette clause CNF et chacune des clauses unaires a_i qui s'unifie deux à deux avec les b_i pour obtenir au final la résolvante c^σ.

5 Sémantique opérationnelle de Prolog

La manière inductive dont a été définie la dénotation d'un programme peut suggérer que Prolog fonctionnerait en *chaînage avant*, c'est-à-dire en partant des faits et en appliquant les règles jusqu'à obtenir des atomes logiques s'unifiant avec la requête. C'est l'inverse qui est réalisé (chaînage arrière) : on part de la requête et on cherche d'abord quelles règles permettraient de l'obtenir.

Il existe a priori plusieurs façons de déterminer si une requête est déduisible d'un programme Prolog et plusieurs substitutions valides. L'algorithme de la figure 4 définit comment Prolog trouve la substitution (si elle existe) d'une conjonction d'atomes logiques r_1, ..., r_k. Au départ, il crée la résolvante (suite de buts) $<r_1, ..., r_k>$ et σ est vide. L'algorithme essaie de réduire la résolvante à l'ensemble vide ainsi : on trouve une clause t :- q_1, ..., q_n dont la tête t s'unifie avec r_1 grâce à l'unificateur σ et on recommence avec la résolvante $<q_1^\sigma, ..., q_n^\sigma, r_2^\sigma, ..., r_k^\sigma>$, jusqu'à obtenir la résolvante vide ou rencontrer un échec (aucune clause ne peut effacer un but). Dans le premier cas, on retourne σ, qui contient la façon dont on doit instancier chaque variable de la requête pour qu'elle soit une conséquence logique du programme. Dans le second cas (échec), on revient en arrière sur le dernier choix de clause et on en essaie une autre. S'il n'en reste plus, on s'arrête sur un échec. Le moteur d'inférence de Prolog considère les clauses dans l'ordre dans lequel elles apparaissent dans le programme.

```
resolution(<r_1, r_2, ..., r_k>, sigma) :
   Si k = 0  (= plus aucun but) alors retourne sigma
   Soit L = <c_1, ..., c_n> la liste des clauses dont la tete s'unifie avec r_1
   Si L est vide alors retourne ECHEC
   i = 1
   resultat = ECHEC
   Tant que i < n + 1 et resultat == ECHEC :
      t = tete(c_i)
      <q_1, ...., q_z> = queue(c_i)
      sigma2 = pgu(t, r_1)
      resolvante = applique_substitution(sigma2, <q_1, ..., q_z, r_2, ..., r_k>)
      resultat = resolution(resolvante, sigma2 o sigma)
      i = i + 1
   retourne resultat
```

FIGURE 4 –

Exemple : la requête `?- demi_carre(X).` avec le programme suivant :

```
triangle(a, b, c).
triangle(a, d, e).

egaux(a, b).
egaux(b, c).
egaux(a, c).
egaux(a, d).

angle_droit(a, d).

triangle_rectangle(t(C1, C2, C3)) :- triangle(C1, C2, C3), angle_droit(C1, C2).
triangle_rectangle(t(C1, C2, C3)) :- triangle(C1, C2, C3), angle_droit(C2, C3).
triangle_rectangle(t(C1, C2, C3)) :- triangle(C1, C2, C3), angle_droit(C1, C3).

triangle_isocele(t(C1, C2, C3)) :- triangle(C1, C2, C3), egaux(C1, C2).
triangle_isocele(t(C1, C2, C3)) :- triangle(C1, C2, C3), egaux(C2, C3).
triangle_isocele(t(C1, C2, C3)) :- triangle(C1, C2, C3), egaux(C1, C3).

triangle_equilateral(t(C1, C2, C3)) :- triangle(C1, C2, C3), egaux(C1, C2),
                                       egaux(C2, C3), egaux(C1, C3).

demi_carre(T) :- triangle_rectangle(T), triangle_isocele(T).
```

produit l'arbre de recherche de la figure 5 et retourne l'instanciation T=t(a, d, e).

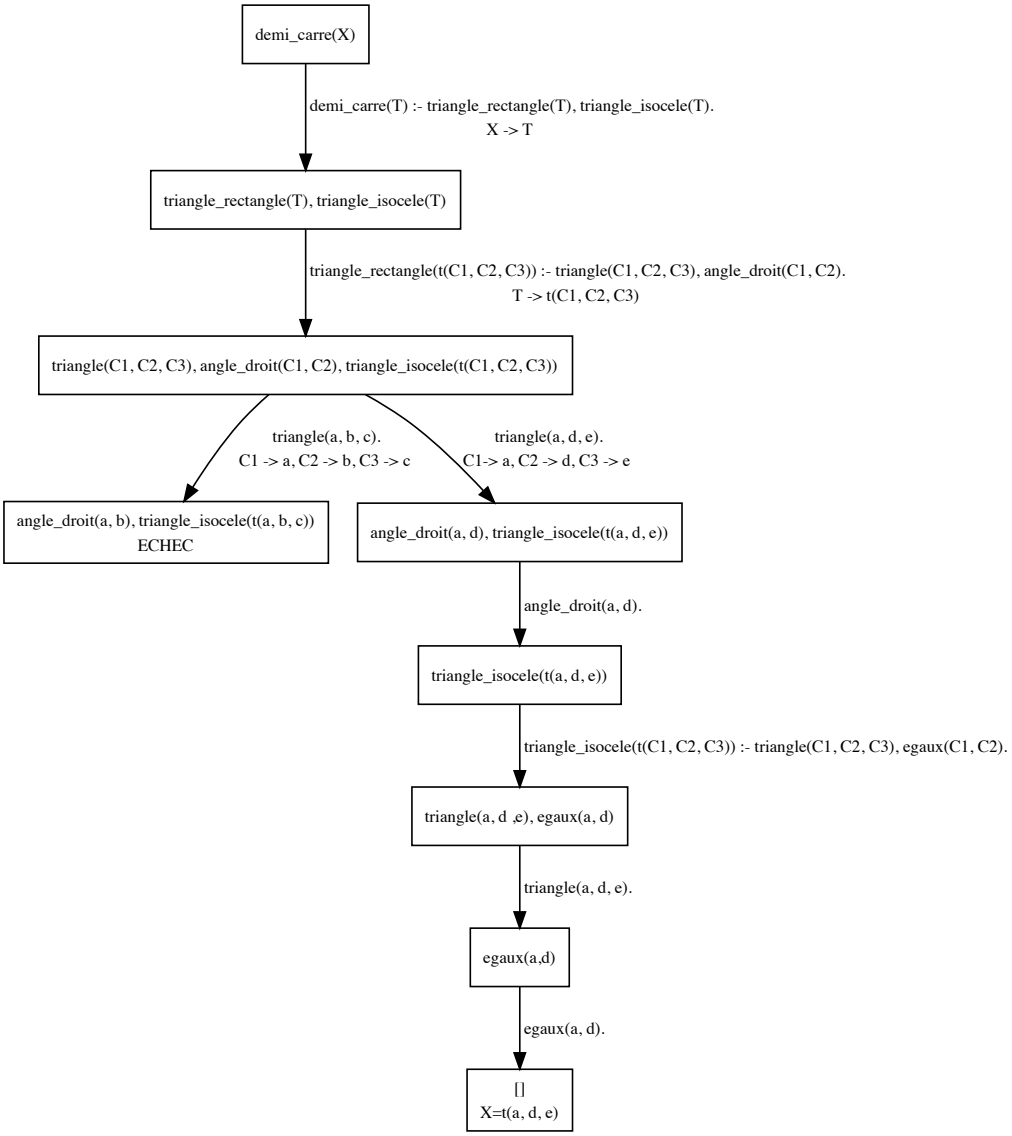

FIGURE 5 – Arbre de recherche de la requête `?- demi_carre(X).`

Il est important que le programmeur connaisse la méthode de résolution utilisée par Prolog s'il veut éviter des boucles infinies ou améliorer le temps de réponse à une requête. Par exemple, le programme suivant boucle indéfiniment si on lui fait la requête `?- egaux(b,a).` (cf figure 6 à gauche) :

```
egaux(X, Y) :- egaux(Y, X).
egaux(a, b).
```

Mais si on intervertit les deux clauses, il donne la réponse `yes` (cf figure 6 à droite). De même, si on veut définir une clause pour chercher un chemin entre deux sommets dans un graphe, il ne faut pas écrire

`chemin(X, Y) :- chemin(X, Z), arc(Z,Y).`, qui boucle infiniment mais

`chemin(X, Y) :- arc(X,Z), chemin(Z, Y).`.

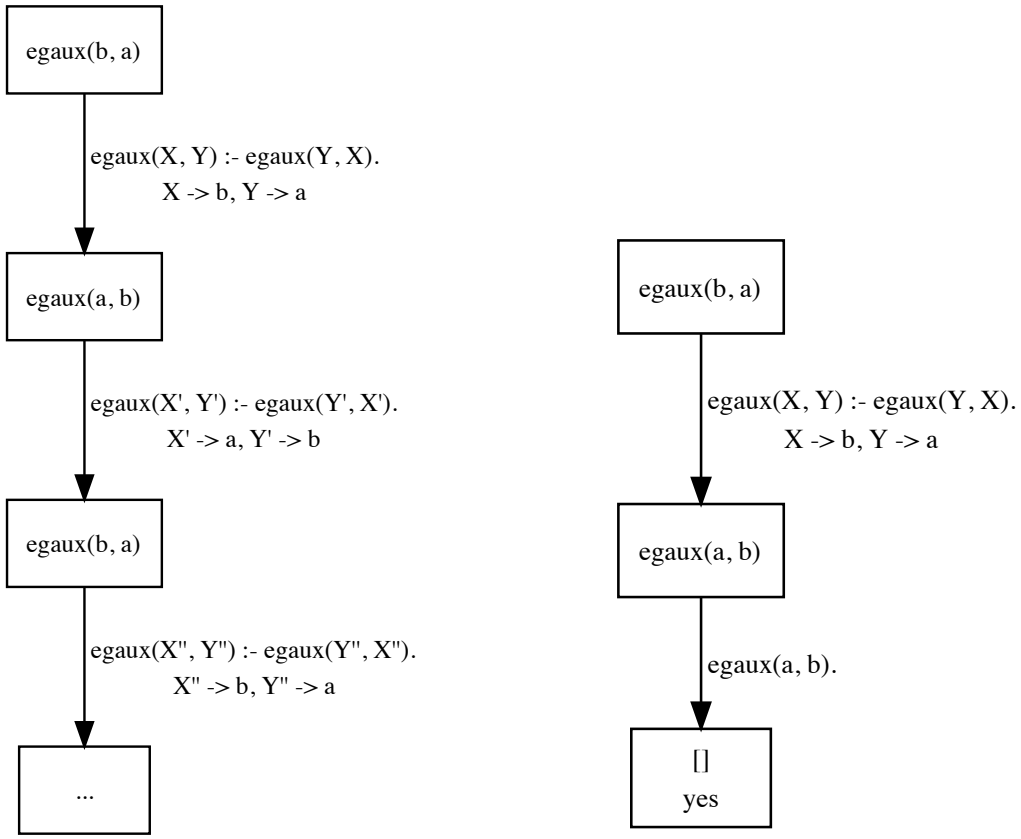

FIGURE 6 – Arbre de recherche de la requête `?- egaux(b, a).` en fonction de l'ordre des clauses.

6 Structures de données supplémentaires et prédicats prédéfinis en Prolog

Ce que nous avons présenté jusqu'à présent constitue le "noyau dur" de Prolog, qui suffit pour résoudre tout type de problèmes mais pas forcément efficacement. Nous allons maintenant introduire quelques éléments complémentaires de Prolog qui rendent la programmation plus efficace et plus agréable.

Prolog contient d'autres types de termes atomiques que l'atome : l'entier (relatif), le nombre flottant, la chaîne de caractères (à écrire entre apostrophes). Il s'agit juste d'une autre manière d'écrire des atomes : `123`, `3.14159`, `'nicolas.prcovic@univ-amu.fr'`.

Il contient aussi le type liste, analogue à celui des langages fonctionnels (Caml, Haskell, ...).

6.1 Les listes

Une liste se définit inductivement comme étant :

— soit la liste vide,

— soit un couple composé d'un élément (de n'importe quel type), appelé la *tête de la liste*, et d'une liste, appelée *reste de la liste*.

Sans type liste prédéfini, on pourrait par exemple définir des listes grâce à l'atome `vide` pour représenter la liste vide et grâce au terme `liste(Tete, Reste)`. Une liste composée de la suite d'éléments `12`, `abc` et `g(a,b)` serait représentée par le terme `liste(12, liste(abc, liste(g(a, b), vide)))`. À la place, Prolog permet la syntaxe `[Tete|Reste]` pour désigner une liste, et `[]` pour désigner une liste vide. On peut donc à la place écrire `[12 | [abc | [g(a, b) | []]]]`. On peut aussi utiliser la simplification d'écriture `[Element1, Element2, Element3]` à la place de `[Element1|[Element2|[Element3|[]]]]`, ce qui nous permet de réécrire le **même** terme ainsi : `[12, abc, g(a, b)]`. Une liste est un terme composé comme les autres (mais avec une syntaxe particulière) et est traité comme tel. Par exemple, le résultat de l'unification de `[a|R]` avec `[X, 5, Y]` est la substitution `X=a` et `R=[5,Y]`.

6.2 Les opérations arithmétiques

La logique des prédicats permet de définir l'arithmétique sur les entiers mais de façon lourde par rapport à nos habitudes. Par exemple, on peut représenter les entiers (dits de Peano) grâce à l'atome `0` et le terme `s(X)` qui signifie "successeur de X". L'entier 3 peut donc se représenter par le terme `s(s(s(0)))`. Ensuite, on peut définir les prédicats `add(X, Y, Somme)`, etc qui permettent d'effectuer des opérations entre entiers de Peano. Mais c'est inefficace et lourd à manipuler. C'est pourquoi Prolog fournit à la place des "outils" de calcul qui rompent partiellement avec son aspect déclaratif mais permettent d'obtenir rapidement le résultat d'une expression arithmétique.

Les 4 opérateurs arithmétiques (`+, -, *, /`) peuvent servir de foncteur. Ex : le terme `+(5, *(X, 3))` est valide. Mais on peut les utiliser aussi sous leur forme infixe équivalente : `5 + X * 3`. Attention : le terme `5 + X * 3` n'est pas une expression arithmétique qui sera évaluée automatiquement par Prolog mais juste une autre façon d'écrire le terme `+(5, *(X, 3))`.

Un (pseudo-)prédicat va permettre le calcul arithmétique : le prédicat prédéfini `is` d'arité 2 et s'écrivant en notation infixe : <T1> `is` <T2> fait le calcul de l'expression arithmétique <T2> et essaie d'unifier le nombre obtenu avec <T1>. Il y a erreur à l'exécution si <T2> n'est pas évaluable en un nombre ou si <T1> n'est ni une variable, ni un nombre. Il y a échec si <T1> est un nombre

ou une variable déjà instanciée à un nombre et que l'expression évaluée est différente.
Exemples :

- la requête `?- X is 1 + 3, Y is X * (X + 9).` réussit et retourne `X=4` et `Y=52`.
- `?- 3 is 1 + 1.` échoue (mais ne provoque pas d'erreur).
- `?- 1 + 1 is 2.` provoque une erreur car $1 + 1$ est un terme alors qu'il faudrait une variable ou un nombre.
- `?- Y is X * (X + 9), X is 1 + 3.` provoque une erreur car `X` n'est pas instancié quand Prolog tente d'évaluer `X * (X + 9)`.
- `?- X is 1 + a.` provoque une erreur car `1 + a` n'est pas une expression arithmétique.

Prolog définit aussi des (pseudo-)prédicats qui sont des opérateurs infixes de comparaisons : `>`, `<`, `>=`, `=<`, `==` et `\==` permettent de comparer des nombres (ou des variables instanciées à des nombres). Il y a une erreur à l'exécution si un des éléments comparés n'est pas un nombre.
Exemple de programme permettant le calcul du N^e nombre de Fibonacci :

```
/* fibo(N, F) est vrai si F est le Nieme nombre de Fibonacci.*/
/* N doit etre instancie' lors d'une requete */
fibo(0, 0). /* f_0 = 0 */
fibo(1, 1). /* f_1 = 1 */
fibo(N, Fn) :- /* si n > 1, f_n = f_(n-1) + f_(n-2) */
    N > 1,
    Nmoins2 is N - 2,
    fibo(Nmoins2, FnMoins2),
    Nmoins1 is N - 1,
    fibo(Nmoins1, FnMoins1),
    Fn is FnMoins1 + FnMoins2.
```