6

Syntaxe de Prolog IV

LA SYNTAXE DE PROLOG IV est décrite dans deux chapitres. L'un d'eux (celui-ci) décrit la syntaxe de Prolog IV sous une forme simplifiée, afin de ne pas noyer le lecteur sous les nombreuses particularités de la syntaxe.

L'autre décrit la syntaxe précise de prolog ISO avec tous ses détails. Il est intitulé «Syntaxe ISO», et il est possible de l'ignorer dans un premier temps.

Le but de ce chapitre est donc de présenter brièvement la syntaxe de Prolog IV. La description donnée ici est approximative et volontairement simplifiée à l'extrême.

Une annexe, sous la forme de questions-réponses, se veut donner quelques informations supplémentaires en rapport plus ou moins direct avec la syntaxe de Prolog IV. D'autres annexes donnent des tables d'opérateurs prédéfinis dans les deux modes de fonctionnement de Prolog IV que sont `prolog4` et `iso`.

6.1 Mini-glossaire

Atome : désigne un symbole. C'est un objet de base du langage, généralement exprimé à l'aide de caractères alpha-numériques. C'est une expression consacrée dans la communauté prolog. On peut trouver également le vocable *identificateur*.

Notation décimale : se dit d'un nombre exprimé au moyen d'un nombre fini de chiffres et du point décimal [1] (par exemple 3.141592).

Notation exponentielle : se dit d'un nombre exprimé au moyen de la notation décimale et suivi d'un exposant (signé ou pas) d'une puissance de 10. ($1.71\texttt{e}98$) On appelle aussi cette notation *notation flottante*.

Notation fractionnaire : se dit d'un nombre exprimé au moyen de chiffres et de la barre de fraction «`/`» (par exemple `22/7`).

Codage flottant IEEE : se dit du codage d'une entité numérique (généralement exprimé au moyen de la notation flottante) en un nombre flottant IEEE (qui peut être simple ou double), comme dans la plupart des langages de programmation. Ces nombres flottants sont en quantité finie, et ont une représentation décimale finie. Il ne peuvent donc représenter qu'un sous-ensemble

1. Et non pas de la virgule décimale !

fini de l'ensemble **D** des nombres décimaux, lui-même sous-ensemble de l'ensemble **Q** des nombres rationnels.

Codage rationnel précis : se dit du codage d'une entité numérique sans perte d'information. En Prolog IV, tout nombre entier ou rationnel peut être codé en précision parfaite[2].

Caractère graphique : tout caractère pris dans l'ensemble «`#$&*+-./:<=>?@^~\`».

Le choix du codage numérique employé dépend du contexte et de la notation employée.

6.2 Les modes d'utilisation de Prolog IV

La norme ISO pour prolog a été publiée récemment. Prolog IV, basé sur prolog, respecte cette norme. Bien sûr, pour que Prolog IV soit un langage moderne et intéressant, de nombreuses extensions (quand ce ne sont pas des remaniements) ont été apportées à prolog. Certaines de ces extensions sont incompatibles avec la norme, d'où les divers modes de fonctionnement du logiciel Prolog IV.

- Le mode *iso-strict* permet de faire tourner des programmes écrits en «Prolog ISO strict».
- Le mode *iso* comporte les extensions Prolog IV (comme les contraintes) sans altérations syntaxiques.
- Le mode *prolog4* est le mode *iso* plus des extensions syntaxiques rendant Prolog IV plus agréable à utiliser. Dans ce mode, on s'éloigne de l'interprétation habituelle d'une règle.

Le mode `iso-strict` est peu intéressant en lui-même (son existence est imposée par la norme). Les modes `iso` et `prolog4` se distinguent par leur prompt. En effet, dans l'attente d'une requête, «`?-`» s'affiche en mode `iso` et «`>>`» s'affiche en mode `prolog4`.

6.3 La logique dans Prolog

La syntaxe de Prolog IV est celle de la logique du premier ordre avec égalité[3]. Il est donc normal de retrouver les concepts de logique même s'ils ne sont pas omniprésents à l'esprit du programmeur. Objet incontournable de la logique, la formule logique est constituée de symboles conventionnels représentants les divers connecteurs *et* ($\wedge$), *ou* ($\vee$), les constantes *true* et *false*, la quantification existentielle ($\exists$), l'égalité ($=$), et de formules atomiques. On passera sous silence la (véritable) négation[4] ($\neg$), qui n'est généralement pas implantée dans prolog.

Montrons rapidement l'équivalence entre ces divers symboles logiques et ceux

2. qui n'est limitée que par la place mémoire.
3. On peut rappeler ici que PROLOG vient de «PROgrammation en LOGique».
4. qui n'est pas du tout la négation par échec, laquelle est à la fois dans tous les prolog et incomplète.

utilisés dans la syntaxe de Prolog IV :

Symbole logique	Symbole Prolog	Construction logique	Construction Prolog
$true$	`true`	$true$	`true`
$false$	`false`	$false$	`false`
$\vee$	`;`	$p \vee q$	p `;` q
$\wedge$	`,`	$p \wedge q$	p `,` q
$\exists$	`ex`	$(\exists x)p$	X `ex` p
$=$	`=`	$t_1 = t_2$	t_1 `=` t_2

Bien sûr, il reste encore à définir comment noter les autres objets de la logique que sont les constantes, variables, termes et relations.

6.4 Les objets du langage

Dans tous les modes de fonctionnement, les objets syntaxiques du langage sont les mêmes, ce qui ne veut pas dire que ce qui est codé ne dépend pas de ce mode ! Les espaces sont significatifs et servent à rendre distinctes deux entités qui auraient pu n'en faire qu'une.

Quand une entité est lue, sa nature dépend du premier caractère. Ces différents cas sont explicités ci-après :

6.4.1 Les noms (atomes et variables)

Si le premier caractère lu est une lettre ou le caractère «_»[5],
les lettres, les chiffres et «_» qui suivent s'agglutinent, comme dans la plupart des langages de programmation.

Le résultat produit dépend de la nature du premier caractère :

- Si c'est une lettre majuscule ou bien «_», l'entité produite est une variable. Exemples : `Var _1 X X_2 TOTO Tete_regle`.
- Si c'est une lettre minuscule, l'entité produite est un atome (un symbole). Par exemple : `var conc x x_15 tOTO`.

6.4.2 Les nombres

Si le premier caractère lu est un chiffre,
les chiffres qui suivent s'agglutinent à celui-ci et formeront un nombre entier. Précisons tout de suite que les nombres entiers sont de tailles (presque) illimitée en Prolog IV. Donnons comme exemples :

`17` *et* `12345678901234567890123456`

Après ces chiffres peut se trouver un point suivi d'autres chiffres, eux-mêmes éventuellement suivis de la lettre «`e`» puis d'un signe «`+`» ou «`-`» facultatif puis d'une suite de chiffres. Ce qui est lu est alors une nombre écrit en notation décimale (ou flottante). Par exemple `1.23 4.56e7 8.9e-10`.

Le codage par Prolog IV du nombre écrit en notation décimale dépend du mode syntaxique.

- En mode `iso`, c'est un nombre flottant binaire (en double précision).

5. blanc souligné, dit encore *underline* ou *underscore*.

– En mode `prolog4`, c'est un nombre rationnel (`1.2` est $12/10$, soit $6/5$). Dans ce mode, la notation fractionnaire est reconnue ; on peut donc également taper `12/10`.

On forme un nombre négatif en mettant un signe « `-` » immédiatement avant le nombre quel qu'il soit (sans aucun espace entre).

6.4.3 Les caractères graphiques

Si le premier caractère lu est un caractère graphique,

(donc parmi l'ensemble «`#$&*+-./:<=>?@^~\`»),

tous les caractères graphiques lus ensuite s'agglutinent. Ils forment une entité graphique, laquelle sera codée comme un atome, bien que n'étant pas alphanumérique.

6.4.4 Autres atomes

Chacunes des entités suivantes, lues, forment un atome : `!` `[]` `;`

Notes :

– `!` est utilisée pour représenter la coupure de l'espace de recherche (cut).

– `[]` est utilisée pour représenter la liste vide.

– `;` est utilisée pour représenter la disjonction de littéraux.

6.4.5 Les séparateurs

Les caractères suivants ne peuvent s'agglutiner, ni entre eux, ni avec aucun autre caractère. Ils forment chacun une entité simple ; ce sont des auxiliaires permettant des constructions décrite plus loin :

`(` `,` `)` `[` `|` `]` `{` `}`

Les espaces et les commentaires (voir plus loin) sont aussi des séparateurs.

6.4.6 Les atomes quotés (chaînes de caractères à apostrophe)

Si le premier caractère lu est une apostrophe « `'` »,
tous les caractères qui suivent sont agglutinés, et ce jusqu'à la prochaine apostrophe. L'entité est codée en tant qu'atome. Les apostrophes qui délimitent la chaîne font pas partie de l'atome codé. Il faut aussi savoir que :

– On ne peut pas mettre de retour chariot dans une chaîne [6].

– Pour que l'apostrophe figure dans la chaîne, il suffit d'en taper deux.

– Bien que les entités syntaxiques `'toto'` et `toto` diffèrent, les atomes codés sont identiques.

Voici des exemples de chaînes à apostrophe :

```
'cette chaine est un atome'
'X = [\n'
'l''apostrophe est : '''
```

6. On peut mettre la séquence «`\n`» pour représenter un retour-chariot.

6.4.7 Les commentaires

Les commentaires servent comme dans tout langage de programmation à poser des informations textuelles qui ne sont pas destinées à l'analyseur ou le compilateur du langage. Bien que ces derniers ignorent le contenu de ces commentaires, ceux-ci ont toutefois l'effet non-négligeable d'être interprétés comme un caractère espace. Un commentaire peut donc être mis à tout endroit où un caractère blanc pourrait être placé.

Il y a deux types de commentaires en Prolog IV : le commentaire *ligne* et le commentaire *bloc* :

- Si le caractère lu est un «`%`» tous les caractères jusqu'à la fin de ligne comprise sont ignorés.
- Si les deux caractères lus sont «`/*`», tous les caractères jusqu'à la prochaine séquence «`*/`» comprise sont ignorés.

Quelques exemples :

```
        % commentaire sur une seule ligne

  /* commentaire sur une ligne aussi */

    /* long commentaire que l'on
    ecrit sur
         plusieurs lignes */
```

Les espaces et donc les commentaires sont des séparateurs.

6.4.8 Expressions simples

On appelle expression bien formée (EBF) une suite d'entités syntaxiques, agencées selon certaines règles. Voici quelques unes de ces règles :

- Atomes et nombres sont des EBF.
- Les variables sont des EBF.
- Si t est une EBF, (t) est une EBF.
- Si t_1,..., t_n avec $n \geq 1$ sont des EBF, et *atome* un atome, alors *atome*`(`t_1 `,` ... `,` t_n`)` est une EBF appelée notation fonctionnelle. Il faut impérativement que l'atome soit collé à la parenthèse ouvrante !
- Si t_1,..., t_n avec $n \geq 0$ sont des EBF, alors `[`t_1 `,` ... `,` t_n`]` est une EBF appelée liste.
- Si t_1,..., t_n, t_{n+1} avec ($n \geq 1$) sont des EBF, alors `[`t_1 `,` ... `,` t_n `|` t_{n+1}`]` est une EBF appelée paire-pointée.

Voici des exemples d'expressions bien formées :

```
123     1.23e18    --     .+     @=<
toto    toto(a)    toto(a,b)     ;(nl,'nl')
[a]     [a|b]      [a,b,c]       [a,b,c|d]
```

On appellera *terme syntaxique* ou plus simplement *terme* une expression bien formée.

6.4.9 Opérateurs

Un opérateur est un atome ayant été déclaré[7] comme pouvant être utilisé lors de constructions syntaxiques spéciales (autres que celles des expressions simples). A cet atome est associée un nombre représentant une priorité[8], et un spécificateur[9] qui décrit la façon dont l'opérateur capte ses opérandes. Un opérateur peut être déclaré préfixé, infixé ou postfixé. On peut également spécifier l'associativité.

Dans Prolog IV un certain nombre d'opérateurs sont prédéfinis en mode `iso-strict` et `iso` ; En voici quelques uns :

Priorité	Spécificateur	Opérateurs
1200	xfx	:-
1200	fx	:- ?-
1100	xfy	;
1000	xfy	,
700	xfx	=
500	yfx	+ -
400	yfx	* /
200	fy	-

D'autres opérateurs sont prédéfinis pour le fonctionnement en mode `prolog4`. Des tables d'opérateurs sont données en annexe de ce chapitre.

Maintenant que sont introduits les opérateurs, voici d'autres règles qui viennent compléter la construction d'EBF :

- Si t_1 et t_2 sont des EBF, *atome* un opérateur infixé, alors « t_1 *atome* t_2 » est une EBF. En ce qui concerne la lecture d'expressions en prolog, cette expression est indiscernable de l'expression fonctionnelle «$atome(t_1, t_2)$ ».
- Si t une EBF, *atome* un opérateur préfixé, alors «*atome* t» est une EBF. En ce qui concerne la lecture d'expressions en prolog, cette expression est indiscernable de l'expression fonctionnelle «$atome(t)$ ».
- Si t une EBF, *atome* un opérateur postfixé, alors «t *atome*» est une EBF. En ce qui concerne la lecture d'expressions en prolog, cette expression est indiscernable de l'expression fonctionnelle «$atome(t)$ ».

Il faut bien sûr que la priorité de l'opérateur principal (s'il existe) des t_i soit plus forte (de valeur inférieure donc) que celle d'*atome* pour avoir le résultat présenté.

On peut toujours outrepasser les priorités et l'associativité des opérateurs par le biais d'expressions parenthésées, qui sont, tout comme les variables et constantes, de priorité maximale (c.à.d. zéro).

7. au moyen de la primitive ou de la directive op/3
8. Plus petit est le nombre, plus prioritaire est l'opérateur.
9. Le spécificateur est un atome pris dans la liste `fx fy xfx xfy yfx xf yf` . Le `f` désigne l'opérateur et le `x` ou `y` représente les opérandes. On imagine donc assez bien comment on spécifie qu'un opérateur est en position préfixée, infixée ou postfixée. Pour savoir que représente exactement les `x` et `y`, il faut se reporter au chapitre décrivant la syntaxe Prolog ISO.

Voici d'autres exemples d'expressions bien formées, avec des opérateurs déclarés :

```
123 + B * 3/4                X.-.3.*.log(Y)
1 u 2 u cc(4,7)              A:3:I
X ~ cc(-2, 3/4)              X = Y - 7
(Cond -> then ; else)        ?- requete
regle(X) :- b(X), (c(X) ; d(X))
```

6.4.10 Opérateurs du mode `prolog4`

Les opérateurs prédéfinis de la norme ISO sont bien sûr présents dans Prolog IV. Toutefois, en mode `prolog4`, d'autres ont été ajoutés, dans le but de rendre plus lisibles des expressions et formules qui apparaitraient trop textuelles sinon.

Priorité	Spécif.	Opérateurs	Commentaires
1000	xfy	ex	quantificateur existentiel
700	xfx	~	compatibilité termes/pseudo-termes
700	xfx	gelin lelin gtlin ltlin	contraintes linéaires
700	xfx	ge le gt lt	relations sur les réels (•)
700	xfx	and or xor impl equiv	relations booléennes (•)
650	yfx	u	union d'ensembles (†)
640	yfx	n	intersection d'ensembles (†)
600	yfx	o	concaténation de listes ($conc$) (†)
680	xfx	bimpl	pseudo-opérations booléennes (•)
650	yfx	bequiv	,, ,,
600	yfx	bor bxor	,, ,,
550	yfx	band	,, ,,
500	yfx	.+. .-.	$+$ et $-$ binaires ($plus$, $minus$) (•)
400	yfx	.*. ./.	$\times$ et $\div$ ($times$, div) (•)
200	fy	.+. .-.	$+$ et $-$ unaires ($uplus$, $uminus$) (•)
200	xfx	.^.	puissance ($power$) (•)
150	yfx	:	indexation de liste ($index$) (†)

Le • indique que la relation associée à l'opérateur travaille dans le solveur approché sur les réels.

Le † indique que la relation associée à l'opérateur travaille dans le solveur approché sur les arbres.

L'opérateur « : », déclaré dans la norme ISO et inutilisé , est ici redéfini et joue le rôle d'opération d'indexation sur les listes.

6.4.11 Entités étendues

Ces extensions concernent essentiellement les nombres et différentes façons de les construire. On trouvera aussi une façon de noter les identificateurs afin qu'ils ne soient pas compris comme étant un nom de relation prédéfinie utilisée avec la notation fonctionnelle (pseudo-termes).

Les entrées possibles de nombres

Les nombres rationnels et les nombres flottants peuvent être également notés de la façon décrite ci-après, quel que soit le mode syntaxique `iso` ou `prolog4` :

Soient les constructions suivantes :

1. `` `?$N_1$` ``
2. `` `?$N_1$ . $N_2$` ``
3. `` `?$N_1$ . $N_2$ {[eE] {[+-]} $N_3$}` ``
4. `` `?$N_1$ / $N_2$` ``
5. `` `?$N_1$ + $N_2$ / $N_3$` ``
6. `` `?$N_1$ - $N_2$ / $N_3$` ``

Avec les conventions suivantes :

- Ce qui est entre accolades (qui ne doivent pas etre tapées) est facultatif.
- Parmi tous les caractères qui sont entre crochets (qui ne doivent donc pas pas etre tapées) on doit faire le choix d'un seul.
- (`) est la back-quote (accent grave) et non pas l'apostrophe ou accent aigu.
- Les N_i sont des suites de chiffres décimaux.
- «?» représentant une des lettres «`r`», «`<`», «`>`» ou «`f`», avec :

  «`r`» pour construire un nombre rationnel (précis),

  «$<$» et «$>$» pour construire un nombre rationnel représentable par un flottant simple (par défaut ou par excès selon le symbole utilisé), le plus proche du nombre rationnel fourni.

  «`f`» pour construire un nombre flottant (pour les seules formes 1 à 3).

Le nombre construit est positif. Un signe `-` placé immédiatement devant lui change son signe.

Les identificateurs

Puisque certains noms sont prédéfinis, on est immanquablement amené à utiliser, dans une règle ou une requête, — peut-être pour des besoins de méta-programmation — un nœud d'arbre ayant un nom et une arité réservés par une relation. Le moyen d'empêcher le compilateur de traiter ce nœud comme un pseudo-terme est de mettre un caractère d'échappement «`^`» au début de l'identificateur, quitte à rajouter une paire d'apostrophes autour du tout pour qu'il reste syntaxiquement correct. Ce caractère ne fait bien sûr pas partie de l'atome codé.

Le terme :	doit être écrit :
`toto(a+b)`	`toto(^+(a,b))`
`toto(n(a,u))`	`toto('^n'(a,'^u'))`
`toto(list)`	`toto('^list')`

6.5 Lecture de règles et de requêtes

6.5.1 Lecture de termes

Le terme est le seul type de donnée prolog. Une formule (requête, règle) est donc (au moins syntaxiquement) un terme. Il existe en prolog des primitives de lecture de termes. Pour être lisible par ces primitives, le terme doit être suivi d'une marque de terminaison constituée du caractère point (`.`) suivit d'un blanc (retour-chariot, espace, tabulation, commentaire-ligne [10], ...)

On appelle littéral tout terme d'une des formes suivantes :

- $atome$,
- $atome(t_1, \ldots, t_n)$, les t_i étant des termes.
- `!` (dit *cut*) qui est la fameuse coupure des points de choix.

Requêtes

Une requête est une formule, c.à.d. une combinaison de littéraux agencés par des conjonctions, disjonctions, et quantifications existentielles, tout ceci éventuellement imbriqué sur plusieurs niveaux ; on peut être amené à utiliser des parenthèses pour exprimer la formule voulue. Toute formule doit se terminer par la marque de terminaison (point suivi d'espace.) Voici quelques exemples de requêtes :

```
write('Bonjour a tous'), nl.
X ex Y = cos(X), int(Y).
Y ex Ville ex habite(jean,Ville), habite(Y, Ville),
              dif(Y, jean).
X = 1 ; X = 2.
```

Règles

Il existe deux formes de règle[11] (du point de vue syntaxique) :

- Celle qu'on appelle *fait*, c.à.d. juste un littéral qui est la tête de la règle (et qui peut contenir des pseudo-termes). Le fait est terminé par la marque de terminaison.
- Celle qu'on appelle *règle* proprement dit, qui est une construction de la forme :

  tête-de-règle `:-` *queue-de-règle* `.`

  où *tête-de-règle* est un littéral, et où *queue-de-règle* a exactement la syntaxe d'une formule représentant une requête possible.

Un fait peut être vu comme une règle dont la queue ne contient que le littéral `true`.

Quelques exemples de règles :

```
premier(31).
fait(noir(nuit)).
```

10. un commentaire bloc ne peut convenir pour la bonne raison qu'il ne peut être reconnu comme tel. En effet, les caractères `./*` s'agglutinent pour former un atome.
11. essentiellement pour des raisons historiques.

```
element_de(X, [Y|L]) :- element_de(X, L).
distance(X1,Y1, X2,Y2, D) :-
    D ~ sqrt(square(X2.-.X1).+.square(Y2.-.Y1)).
somme([X | L], X.+.Y) :- somme(L, Y).
```

6.5.2 Transformation des règles (et requêtes) en codage interne

Tout terme lu sensé repésenter une règle destinée à augmenter la base de règles est codé. Il n'est bien entendu pas question de détailler ce codage, mais il est nécessaire de connaître certains points de cette transformation, dépendante du mode syntaxique, et décrite pour chacun d'entre eux.

Mode `prolog4`

- Les entités numériques entrées avec la notation flottante sont codés par des nombres précis (rationnel).
- Tout littéral où figure la quantification existentielle (`X ex` p) est transformé en p', tiré du littéral p en remplaçant dans celui-ci toute occurence de la variable `X` par une variable neuve ne figurant pas déjà dans la règle. Ce processus est effectué en postordre dans la formule (du plus profond vers la surface).
- Tout terme fonctionnel figurant dans un littéral est traduit en un nœud de terme logique, si le nom et l'arité du nœud n'est pas celui d'une relation prédéfinie. Sinon, le terme en question est un pseudo-terme, et la transformation suivante est appliquée sur le littéral :

  Le pseudo-terme est remplacé dans le littéral par une variable neuve.

  Un littéral construit à l'aide du pseudo-terme est inséré *avant* celui qu'on est en train de traiter.

  La variable neuve est insérée *avant* les autres arguments du nouveau littéral.

  Le littéral
  $$p(a_1, \ldots, rel(b_1, \ldots, b_n), \ldots, a_m)$$
  est donc remplacé par :
  $$rel(X, b_1, \ldots, b_n), p(a_1, \ldots, X, \ldots, a_m)$$
  la variable `X` ne figurant pas déjà dans la règle.

  Les littéraux insérés sont également traités de cette façon. L'ordre d'expansion des pseudo-termes entre eux est indéfini.
- Si le premier caractère d'un identificateur est un « `^` », alors ce qui est codé est l'identificateur privé de ce caractère, sauf dans le seul cas de l'identificateur « `^` », qui reste inchangé. Cette règle de transformation est appliquée après toutes les autres.

Mode `iso`

- Les entités numériques entrées avec la notation flottante sont codées par un nombre flottant IEEE double précision.

– Tout terme fonctionnel figurant dans un littéral est traduit en un nœud de terme logique.

Différence entre le mode `prolog4` *et le mode* `iso`

Voici essentiellement ce qu'on perd quand on est en mode `iso` :

– Pas de pseudo-termes dans l'interprétation des requêtes et des règles lues (toute structure est un constructeur). Il faut donc utiliser la notation relationnelle pour écrire des contraintes, il n'est pas possible d'en imbriquer.

– Pas de quantification existentielle.

– Une entité ayant la notation flottante est traduite en nombre flottant IEEE double précision, et non pas en nombre rationnel de précision parfaite ; il faut donc utiliser les entités étendues pour pouvoir utiliser des nombres rationnels dans ce mode.

– Des opérateurs en moins (ceux qui sont des raccourcis de relations).

ANNEXE A : Questions et Réponses

Attention : le caractère «`» *est un* guillemet inversé, *comme l'accent grave.*

Q: Peut-on entrer un nombre flottant binaire (de valeur 1.2 par exemple) en étant dans le mode `prolog4`?

> **R**: Il faut le rentrer sous la forme `` `f1.2` `` (`-` `` `f1.2` `` si on voulait entrer -1.2).

Q: Comment peut-on en mode `iso` entrer un nombre fractionnaire (je sais qu'on fait `1/10` en mode `prolog4`)?

> **R**: Il faut le rentrer sous la forme `` `r1/10` `` (`-` `` `r1/10` `` si on voulait entrer la fraction négative $-\frac{1}{10}$).

Q: Quelles différences il y a t-il entre les primitives `write` et `writeq`?

> **R**: Ce qu'affiche `write` est destiné à la lecture par des humains, `writeq` est destiné à la lecture par un programme (par `read` par exemple.) `write` n'affiche aucune apostrophe autour des entités écrites. `writeq` en mettra partout où c'est nécessaire, ainsi que des espaces, afin de lever toute ambiguïté à la relecture.

Q: Les nombres flottants n'ont pas l'air d'être pris en compte dans mes contraintes?

> **R**: En effet, seuls les nombres précis (quelle que soit la notation employée) sont utilisés dans dans les solveurs numériques.

Q: Alors à quoi peuvent bien servir ces nombres flottants?

> **R**: Ils font partie de la norme prolog, et peuvent servir dans des calculs par le biais des primitives `is/2` et assimilées, ainsi que dans des appels de fonctions écrites dans un autre langage de programmation (comme C).

ANNEXE B : Tables d'opérateurs (mode `iso`)

Voici la table standard des opérateurs prédéfinis de la norme prolog ISO, avec leurs priorités, formats et associativités.

Priorité	Spécificateur	Opérateurs
1200	xfx	:- -->
1200	fx	:- ?-
1100	xfy	;
1050	xfy	->
1000	xfy	,
900	fy	\+
700	xfx	= \=
700	xfx	== \== @< @=< @> @>=
700	xfx	=..
700	xfx	is =:= =\= < =< > >=
500	yfx	+ - /\ \/
400	yfx	* / // rem mod << >>
200	xfx	**
200	xfy	^
200	fy	- \
100	xfx	@
50	xfx	:

ANNEXE C : Tables d'opérateurs (mode `prolog4`)

Cette table complète en l'augmentant la table des opérateurs du mode `iso`, quand on entre dans le mode `prolog4`.

La seule modification de la table d'opérateurs du mode `iso` est la redéfinition de l'opérateur «`:`» .

Priorité	Spécif.	Opérateurs	Commentaires
1000	`xfy`	`ex`	quantificateur existentiel
700	`xfx`	`~`	compatibilité termes/pseudo-termes
700	`xfx`	`gelin lelin gtlin ltlin`	contraintes linéaires
700	`xfx`	`ge le gt lt`	relations sur les réels (•)
700	`xfx`	`and or xor impl equiv`	relations booléennes (•)
650	`yfx`	`u`	union d'ensembles (†)
640	`yfx`	`n`	intersection d'ensembles (†)
600	`yfx`	`o`	concaténation de listes ($conc$) (†)
680	`xfx`	`bimpl`	pseudo-opérations booléennes (•)
650	`yfx`	`bequiv`	,, ,,
600	`yfx`	`bor bxor`	,, ,,
550	`yfx`	`band`	,, ,,
500	`yfx`	`.+. .-.`	$+$ et $-$ binaires ($plus$, $minus$) (•)
400	`yfx`	`.*. ./.`	$\times$ et $\div$ ($times$, div) (•)
200	`fy`	`.+. .-.`	$+$ et $-$ unaires ($uplus$, $uminus$) (•)
200	`xfx`	`.^.`	puissance ($power$) (•)
150	`yfx`	`:`	indexation de liste ($index$) (†)

Le • indique que la relation associée à l'opérateur travaille dans le solveur approché sur les réels.

Le † indique que la relation associée à l'opérateur travaille dans le solveur approché sur les arbres.

La table suivante contient quelques commentaires sur les opérateurs existants en mode `iso`.

Priorité	Spécificateur	Opérateurs	Commentaires
500	`yfx`	`+ -`	$+$ et $-$ binaires ($pluslin$, $minuslin$)
400	`yfx`	`* /`	$\times$ et $\div$ ($timeslin$, $divlin$)
200	`fy`	`+ -`	$+$ et $-$ unaires ($upluslin$, $uminuslin$)

Toutes les relations associées à ces opérateurs travaillent dans le solveur linéaire.