www.prolog-heritage.org

Version restaurée à partir des archives de PrologIA gracieusement cédées à l'association PROLOG HERITAGE. Manuel mis en consultation libre sur Internet au printemps 2010.

Contact : association@prolog-heritage.org

LE MANUEL DE PROLOG IV

Tutoriel
Concepts de base
Primitives
Prolog ISO
Syntaxe
Environnement

PrologIA

Parc Technologique de Luminy – Case 919

13288 Marseille cedex 09 – FRANCE

PROLOG IV est un produit de la société PrologIA, conçu en coopération étroite avec l'équipe *Programmation par Contraintes* du LIM, le Laboratoire d'Informatique de Marseille qui associe l'Université de la Méditerranée, l'Université de Provence et le Centre National de la Recherche Scientifique dans l'URA 1789.
Ont participé à sa réalisation Frédéric BENHAMOU, Pascal BOUVIER, Alain COLMERAUER, Henri GARRETA, Jean-Luc MASSAT, Guy Alain NARBONI, Stéphane N'DONG, Jean-François PIQUE, TOURAÏVANE, Michel VAN CANEGHEM, Eric VÉTILLARD, avec l'aide de Bruno GILLETA, Robert PASERO et Jianyang ZHOU.
Son développement a bénéficié d'études menées dans le cadre de deux projets ESPRIT : le projet de recherche ACCLAIM 7195, portant sur la programmation concurrente par contraintes, et le projet PRINCE 5246, portant sur l'environnement et la partie contraintes d'un Prolog à caractère industriel et financier.

Pour toutes questions concernant ce manuel et ce logiciel, contactez :

PrologIA
Parc Technologique de Luminy – Case 919
13288 Marseille cedex 09 – FRANCE

Tél : 33 91 26 86 36
Fax : 33 91 41 96 37
E-mail: `prologia@prologianet.univ-mrs.fr`

Table des matières

Introduction

Généralités

La programmation Logique avec Contraintes, aujourd'hui un domaine scientifique à part entière, matérialise la convergence de deux tendances majeures dans la recherche en informatique de ces dernières décennies.

La première, en amont, concerne le souci constant de faire évoluer le langage Prolog, inventé par Alain Colmerauer à Marseille au tout début des années 70. Cette évolution naturelle s'est orientée vers la conception et la réalisation d'un système de développement de plus en plus ouvert, de plus en plus efficace et de plus en plus tourné vers la conception d'applications réelles. Ce que nous appelons ici application réelle regroupe tout type de développement destiné à résoudre un ou plusieurs problèmes physiques dans un environnement économique complexe. Autrement dit, il ne s'agit pas seulement d'expérimenter, il s'agit de produire, que ce soit des applications utilisées quotidiennement dans l'entreprise, des résultats de recherche performants, ou encore un enseignement adapté à de nouvelles exigences professionnelles. L'un des chemins de cette évolution est celui des contraintes, qui ont permis d'enrichir considérablement le langage en redéfinissant son principe même de fonctionnement.

Cette révolution ne s'est toutefois pas faite en un jour. La définition de Prolog II, au début des années 80 comportait déjà les idées majeures et la machinerie théorique qui allait plus tard donner naissance aux langages de programmation par contraintes. La seconde innovation marquante devra attendre la fin de ces mêmes années 80 et voir l'avènement d'un traitement des problèmes numériques, notamment, en accord avec la philosophie générale du langage, à savoir sa déclarativité et son caractère relationnel.

La seconde tendance mentionnée plus haut est diamétralement opposée. Il s'agit en effet, non pas d'améliorer le langage Prolog pour le rendre plus adapté au développement d'applications industrielles, mais de montrer que ces applications, souvent déjà exprimées en termes de contraintes, ont tout à gagner de la déclarativité, du caractère relationnel et du non-déterminisme inhérent à Prolog. Pour permettre à Prolog d'être efficace dans cet environnement, une évolution est souhaitable, et cette évolution se trouve converger vers les principes d'amélioration du cœur même du langage que nous avons mentionnés plus haut.

Le résultat pratique de ces réflexions s'est traduit à la fin des années 80 par le développement de langages de programmations par contraintes parmi lesquels Prolog III et d'autres langages ont démontré les potentialités du concept dans le monde industriel.

Prolog IV n'est pourtant pas une nouvelle version de Prolog III. Ce précédent langage s'est amélioré au fil du temps, son environnement de programmation s'est enrichi, les plates-formes matérielles et logicielles sur lesquelles il est disponible se sont diversifiées, mais le cœur du langage est resté inchangé depuis son lancement, en 1989. Prolog IV est le résultat à la fois des recherches en amont qui se sont poursuivies depuis lors, et des besoins fondamentaux des utilisateurs tels qu'ils sont apparus au fil du temps. Enfin, une autre étape fondamentale dans l'utilisation industrielle des langages dérivés de Prolog atteint sa phase terminale, puisque l'établissement d'une norme ISO pour Prolog vient d'être publiée.

Pour toutes ces raisons, la famille des Prolog marseillais se devait donc de s'enrichir d'un nouveau membre, mais à l'image de ses prédécesseurs, celui-ci se devait de réunir un certain nombre de qualités indispensables : efficacité du système, rigueur théorique et nouveauté des concepts.

L'efficacité du système est une condition cruciale à la réussite d'un langage, mais aussi à la réussite de tous les langages construits autour de Prolog. Celle-ci est assurée en Prolog IV grâce à un compilateur optimisé et à l'utilisation de résolveurs de contraintes extrêmement performants que nous présenterons plus en détail dans la suite de ce document.

Contrairement à quelques idées reçues, il serait illusoire de croire que la rigueur théorique se résume à un souci d'esthétique. Les langages de programmation par contraintes tels que Prolog IV, par une volonté constante d'amélioration de leur puissance d'expression et de leurs performances deviennent complexes dans leur conception. L'unique garantie que l'exécution des programmes calcule les résultats attendus (et parfois des résultats tout simplement corrects) réside dans la précision de la définition d'une sémantique formelle en amont du développement. Dans ce contexte, la notion de formalisme s'interprète très pragmatiquement pour le programmeur en termes de correction, de robustesse et de réutilisabilité, en un mot de qualité et de fiabilité du logiciel. Le cas de Prolog IV est exemplaire. L'introduction d'un certain degré d'approximation pour le traitement des contraintes, la cohabitation et la coopération de résolveurs de contraintes complexes au cœur même des mécanismes fondamentaux du langage ont constitué un perpétuel challenge à la fois pour la définition d'un langage simple et cohérent et pour l'élaboration d'un canevas formel qui rende compte très précisément du comportement du système lors de l'exécution des programmes. Cette tâche seule a nécessité plusieurs années de recherche.

Les principaux fondements théoriques ainsi que la sémantique formelle de Prolog IV, dus à Alain Colmerauer, sont présentés dans ce document. Le lecteur peu rompu à la pratique de ce genre de littérature pourra bien entendu tirer un parti optimal du langage sans prendre connaissance de cette partie du

manuel de référence dans la mesure où tous les concepts importants sont également exposés de manière plus pratique et peut être plus intuitive tout au long des manuels de référence et d'utilisation. Il n'en reste pas moins que cet exposé donnera un éclairage différent au lecteur curieux et constituera une base de travail privilégiée pour préparer un cours sur le langage, en second cycle des Universités, par exemple.

Enfin, la nouveauté des concepts marque la lignée des Prologs marseillais. Prolog IV n'échappe pas à la règle : il réalise à la fois l'unification des résultats obtenus jusqu'à ce jour et généralise la résolution de contraintes en y incorporant de nouvelles techniques. Tous les résolveurs de contraintes ont été revus, que ce soit par souci d'efficacité (contraintes linéaires) ou par souci de simplification et respect de la norme ISO (contraintes sur les listes). Un nouveau résolveur, dont les fondements proviennent de l'arithmétique d'intervalles permet d'aborder les contraintes non-linéaires sur les réels, et unifie les contraintes sur les domaines finis et les Booléens. Cette généralisation autorise, en outre, le mélange harmonieux de contraintes sur des algèbres jusqu'alors totalement séparées.

Qu'est ce que la programmation par contraintes ?

Avant d'introduire brièvement les caractéristiques du langage, nous revenons sur le concept même de programmation par contraintes et nous proposons quelques éléments de réponses à certaines questions préalables qui peuvent se poser à l'utilisateur peu familier du domaine.

Que sont les contraintes ? On appelle *contrainte* l'expression de toute relation qui lie un certain nombre d'objets qui prennent leurs valeurs dans un certain domaine. A ce titre, les équations de la physique qui imposent à certaines quantités d'être égales, les impératifs qui imposent des relations particulières aux éléments d'une machine, au déroulement d'un process, ou qui imposent des conditions pour la réalisation d'une tâche sont des contraintes. Le but de la programmation par contraintes est de permettre au programmeur de se concentrer sur l'expression de ces relations (la modélisation du problème) par opposition à la recherche de solutions au système constitué de l'expression de ces relations (la résolution du problème). On demande généralement à ces solutions soit d'être réalisables (elles obéissent aux contraintes) soit d'être optimales par rapport à une certaine fonction de coût.

Si le but est d'écrire des équations, des inéquations ou plus généralement des relations pour que le système se charge ensuite de les résoudre, comment se justifie alors l'activité de programmation ? C'est là l'apport fondamental de cette technologie. En effet, nombre de systèmes spécialisés ont pour vocation d'effectuer efficacement ce genre de traitement. La popularité et l'efficacité de la programmation par contraintes viennent du fait que le programmeur dispose en plus d'un langage de programmation complet et généraliste dans lequel ces techniques de résolutions sont profondément intégrées. Ce langage est très souvent indispensable pour :

1. exprimer des contraintes,
2. contrôler leur prise en compte par le système,

3. contrôler certaines phases clefs de leur résolution,
4. programmer d'autres parties du système qui ne se réduisent pas à cette résolution.

Dans ce cas, si les notions de langage hôte et de résolution de contraintes sont distinctes, pourquoi avoir choisi Prolog ? Là encore, la réponse est tout à fait naturelle. Comme nous l'avons esquissé précédemment, les deux étapes principales du traitement de ce type de problème sont :

1. l'expression du problème,
2. le parcours d'un arbre de recherche pour isoler les solutions

A chacune de ces étapes correspond une des deux caractéristiques majeures et discriminantes du langage Prolog : la *déclarativité* et le *non-déterminisme*. Le caractère déclaratif et relationnel de Prolog en fait un outil absolument naturel pour la modélisation. Il n'est pas nécessaire de traduire en *actions* l'expression du problème, puisque le langage lui même en permet un codage déclaratif pratiquement immédiat. D'autre part, il n'est pas non plus nécessaire de coder le parcours de l'arbre de recherche, ce type de traitement faisant déjà partie intégrante du modèle d'exécution du langage.

Présentation succincte du langage

Plus précisément, quelles sont les caractéristiques du langage et quelles sont les nouveautés par rapport à Prolog III ? Dans ce qui suit, on abordera à la fois des généralités conceptuelles (que sont les nombres en Prolog IV, quels sont les types d'algorithmes utilisés pour résoudre les contraintes, etc.) mais également quelques points plus précis. Lors d'une première lecture, le lecteur pourra sauter les parties qu'il juge trop techniques en fonction de son degré de familiarité avec Prolog et la programmation par contraintes et se référer à la section introductive du manuel.

Modes syntaxiques

Tout d'abord la partie Prolog. Celle-ci a évolué pour se conformer au standard ISO. Prolog IV est donc le premier langage de la lignée à ne plus proposer comme option une syntaxe dite «marseillaise». On se rendra donc ici à l'évidence avec quelque nostalgie, les discussions passionnées sur la syntaxe des variables et sur les mérites respectifs des symboles « : - » et « - > » font aujourd'hui partie de l'histoire de l'informatique. Concédons que c'est bien là l'un des objectifs de toute normalisation et il y a fort à parier que la plupart des utilisateurs ne s'en plaindront pas.

En fait, on dispose en Prolog IV de deux modes, appelés *mode ISO* et *mode étendu*. Sans rentrer dans les détails, le mode ISO, comme mentionné précédemment se conforme à la norme, y compris donc la définition d'une longue liste de prédicats prédéfinis que l'on trouvera sous le même nom dans le manuel de référence. Ce qui est plus intéressant est que l'on peut, en mode ISO, utiliser toute la puissance de Prolog IV et en particulier les contraintes. Pour se faire, il suffit d'utiliser les prédicats de contraintes prédéfinis dont on trouvera également le détail dans le manuel de référence. De ce fait, la *totalité* des fonctionnalités de Prolog IV sont accessibles en mode ISO. Ceci introduit donc la

question suivante : « pourquoi un second mode syntaxique ? ». La réponse est très simple. Lorsque l'on utilise les contraintes de manière intensive, un grand nombre de facilités syntaxiques sont souhaitables. La première de ces facilités concerne l'utilisation de termes pour représenter des expressions numériques. Par exemple on souhaite pouvoir écrire la contrainte $2x + y = 5z - t$ de la façon suivante :

```
2*X + Y = 5*Z - T
```

et non pas, comme il faut le faire en mode ISO :

```
timeslin(X1,2,X),
addlin(X2,X1,Y),
timeslin(Y1,5,Z),
minuslin(Y2,Y1,T),
X2 = Y2.
```

La remarque importante ici est que la première manière d'écrire a également un sens en mode ISO, mais ce n'est pas celui que l'on recherche ici. En effet, le traitement de cette égalité a pour effet de chercher des valeurs pour les variables qui vérifient l'égalité entre les deux termes Prolog `2*X + Y` et `5*Z - T`, c'est à dire une égalité entre deux arbres. Cette équation n'a bien entendu pas de solutions (dans le jargon logique souvent utilisé en Prolog, on dirait que les deux termes ne sont pas *unifiables*). On voit donc immédiatement l'intérêt d'un mode plus approprié au traitement des contraintes. Notons au passage que de nombreuses autres possibilités sont également offertes dans le mode *étendu* de Prolog IV, comme l'utilisation de pseudo-termes (voir la définition de ces objets syntaxiques dans le manuel de référence). Intuitivement, l'utilisation de ces pseudo-termes permet de simplifier un certain nombre de notations en les rendant plus « fonctionnelles » comme par exemple :

```
Y = cos(X)
```

au lieu de la notation sous forme relationnelle :

```
cos(Y,X)
```

ou de fixer des domaines de valeurs possibles pour les variables (très grossièrement, on peut faire un parallèle avec des types) :

```
[X,Y,Z,T] ~ [int n ge(0), le(0), co(1,2), list]
```

Encore une fois sans rentrer dans les détails, on exprime ainsi de manière très concise que `X` doit être un entier positif ou nul, que `Y` doit être un nombre réel strictement négatif, que les valeurs de `Z` sont à prendre dans l'intervalle semi-ouvert $[1, 2)$ et enfin que `T` doit représenter une liste.

Contraintes sur les arbres et les listes

Prolog IV hérite des précédents Prolog de la même lignée (Prolog II, Prolog II+ et Prolog III) du traitement de contraintes sur les arbres rationnels (une certaine famille d'arbres infinis) qui inclut le traitement des équations et des diséquations (il s'agit de la relation $\neq$ ou pour parler différemment du célèbre prédicat `dif/2` de Prolog II).

Les listes sont représentées par des arbres binaires de manière habituelle (et conforme à la norme ISO). La notion même de liste a été homogénéisée par rapport à Prolog III et on ne fait plus de différence entre les listes sur lesquelles on peut définir des contraintes de concaténation et les listes usuelles. Les deux principales contraintes restent la concaténation et la contrainte de taille (ou longueur) d'une liste, auxquelles a été ajoutée la très utile contrainte `index/3` qui exprime qu'un objet est le *énième* élément d'une liste.

Mentionnons également ici que la notion de *retardement*, que l'on retrouvera lorsque l'on abordera les contraintes linéaires, a été conservée. Rappelons en deux mots qu'une contrainte est retardée si elle ne satisfait pas un certain nombre de conditions qui permettent son traitement immédiat. Lors de l'exécution d'un programme, une telle contrainte sera ajoutée au système de contraintes courant si ces conditions sont satisfaites ultérieurement. La définition précise des conditions qui régissent le retardement des contraintes sur les listes sera donnée dans le chapitre consacré à ces contraintes, mais remarquons d'ores et déjà que ces conditions diffèrent de celles qui conditionnent le retardement en Prolog III. Grossièrement toute contrainte du type :

```
L = L1 o L2.
```

qui se lit « la liste `L` est formée de la concaténation des listes `L1` et `L2` » est ajoutée au système de contraintes courant si deux au moins des tailles des listes `L, L1` et `L2` sont connues. La contrainte est retardée sinon.

Il existe de nombreuses autres différences avec Prolog III concernant les listes. Ces différences seront détaillées dans la suite de ce document.

Contraintes numériques

C'est de très loin le domaine le plus riche de Prolog IV. Formellement c'est extrêmement limpide, la puissance absolue conférée par l'abstraction mathématique permet de définir toute étiquette numérique d'un arbre rencontré en Prolog IV comme un nombre réel. Cela dit, comme tout mortel rompu aux approximations que la réalité impose à nos réalisations humaines le sait fort bien, la représentation des nombres réels en machine n'est pas sans poser de nombreux problèmes pratiques dont le moindre n'est sans doute pas que, par essence, tous les calculs (ou presque) opérés par un ordinateur sont faux. Ces problèmes sont résolus très imparfaitement par la représentation des nombres réels par des nombres flottants, dont la structure suit généralement les recommandations de la norme IEEE pour ces mêmes flottants. C'est le choix (réaliste) de la norme Prolog qui impose donc le comportement que vous rencontrerez en Prolog IV en mode ISO.

En mode étendu, le comportement est totalement différent. La notion même de nombre flottant disparaît, et chaque valeur numérique est représentée par un rationnel en précision infinie, c'est à dire par un couple d'entiers numérateur-dénominateur où les entiers sont aussi grands que nécessaire (dans la limite de la mémoire disponible bien sûr). Jusque là, rien de très différent de ce à quoi nous avait habitué, dans un souci d'exactitude des calculs, le langage Prolog III.

La différence majeure ainsi que l'une des innovations principales de Prolog IV réside dans le fait que le langage permet de traiter des contraintes numériques linéaires sur les rationnels , non-linéaires (y compris non-polynomiales) sur les réels, des contraintes entières (domaines finis) et enfin des contraintes booléennes ainsi que des contraintes «mixtes» sur plusieurs sous-domaines.

Pour résoudre ces contraintes, deux algorithmes de résolution (on dit parfois résolveurs) principaux sont utilisés. Tout d'abord, un algorithme spécifique est dédié principalement à la résolution des contraintes *linéaires*, c'est à dire les contraintes représentables sous la forme $a_0 + a_1x_1 + a_2x_2 + \ldots + a_nx_n \diamond 0$ où $\diamond$ peut être le signe $=, \neq, >, \geq, <$ ou $\leq$. Cet algorithme, comme en Prolog III est une combinaison d'une procédure d'élimination de Gauss et d'un Simplex incrémental dont la particularité est de détecter les variables *figées* (variables dont l'ensemble des valeurs est réduit à un seul élément). Cet algorithme gère également le traitement retardé de certaines multiplications non-linéaires.

Le second algorithme, qui traite principalement tous les autres types de contraintes numériques est un algorithme de *propagation de domaines*. Essentiellement, cet algorithme calcule localement des domaines de valeurs autorisés pour les variables apparaissant dans une contrainte et propage ces domaines aux autres contraintes qui font intervenir ces mêmes variables. Cette propagation s'opère jusqu'à ce qu'un état stable soit atteint. La séparation des solutions du système initial s'opère alors par une technique de «diviser et conquérir» qui revient principalement à effectuer une dichotomie sur les domaines de valeurs possibles et à explorer l'arbre de recherche binaire correspondant à ces deux choix. Il est extrêmement intéressant de noter que cet algorithme généralisant le traitement des domaines finis et donc des contraintes booléennes permet de ramener plusieurs classes de contraintes à un même niveau et donc de combiner étroitement des domaines généralement considérés comme disjoints. On peut ainsi partager des variables entre des contraintes entières et réelles et même des contraintes booléennes (considérées ici comme des contraintes en 0/1).

Le choix de l'algorithme, particulièrement lorsque les contraintes peuvent être traitées par les deux méthodes est laissé au programmeur. Par exemple, pour ajouter la contrainte $2x - \frac{1}{3}y = 7$ au système de contraintes courant traité par Gauss-Simplex on écrira :

```
2*X - 1/3*Y = 7
```

Pour ajouter la même contrainte au système de propagation d'intervalles, on écrira :

```
2.*.X.-.1/3.*.Y = 7
```

Notons au passage que rien n'interdit d'ajouter les deux contraintes à la fois. A ce propos, se pose la question de savoir comment les deux résolveurs communiquent. La réponse est encore une fois très simple. Les résolveurs communiquent uniquement par le biais des variables figées. Ainsi, si le domaine des valeurs possibles d'une variable est réduit à un seul élément par l'un des deux algorithmes et que cette variable apparaît dans une contrainte traitée par l'autre algorithme, ce dernier tiendra compte de cette information. Toute infor-

mation concernant une autre modification de l'ensemble des valeurs possibles restera locale à l'algorithme qui a effectué cette modification.

Environnement de programmation

Prolog IV n'est pas seulement le compilateur d'un concept de programmation sophistiqué. Le système est doté d'un environnement de programmation multi-fenêtré complet et convivial qui comprend son propre éditeur graphique muni de fonctions multiples, un vérificateur de syntaxe et un débogueur symbolique très soigné spécialement adapté à ce type de programmation. Toutes les facilités de communication avec l'extérieur sont disponibles, ainsi que la possibilité de développer des applications graphiques.

Domaines d'applications

Prolog IV est un langage généraliste muni de puissantes capacités de résolution de contraintes complexes. A ce titre, il s'impose comme un choix privilégié dans un très grand nombre d'applications qui peuvent tirer à la fois profit de ces deux caractéristiques. Parmi celle-ci, on peut citer toutes les applications d'ordonnancement et de planification, le développement de systèmes d'aide à la décision, la simulation de processus physiques complexes, l'optimisation sous contraintes et de manière générale la résolution de problèmes. Les techniques de résolution de contraintes mises à la disposition du programmeur ont prouvé leur efficacité, bien souvent supérieure à celle de systèmes spécialisés, particulièrement lorsque le problème à résoudre ne rentre pas exactement dans un cadre mathématique prédéfini (job-shop pur, voyageur de commerce pur, etc), ce qui s'avère être le cas de nombre d'applications réelles où de multiples contraintes, annexes au problème théorique mais cruciales dans la mise en œuvre des solutions, sont à prendre en compte. Enfin, la souplesse d'utilisation et le caractère concis, précis et déclaratif du langage permettent un développement rapide ainsi qu'une évolution et une maintenance simplifiées des systèmes développés.

Présentation des manuels

Le présent manuel est organisé en cinq parties principales. La première partie se compose d'une prise de contact rapide et d'un tutoriel qui aborde graduellement au travers d'exemples très simples les principaux concepts à la base du langage. La seconde partie définit plus en profondeur ces divers concepts et insiste tour à tour sur les différentes structures algébriques supportant les contraintes de Prolog IV, au travers d'exemples de programmes. La troisième partie compose le manuel de référence, dans lequel tous les prédicats prédéfinis sont détaillés. Une attention toute particulière à été portée à l'explicitation des différences avec Prolog III, ainsi qu'à la mise en valeur des fonctionnalités relatives aux contraintes qui ne sont, par essence, pas documentées dans la norme ISO. Les chapitres suivants décrivent précisément la syntaxe dans les deux modes et l'environnement de programmation. Le manuel se termine par un index général.

Pour terminer cette introduction, nous espérons que vous prendrez autant de plaisir à utiliser ce langage que nous en avons eu à le concevoir, l'implémenter et le rendre à la fois le plus précis, le plus efficace et le plus agréable possible à programmer et à utiliser. Le monde de la programmation par contraintes est vaste, excitant et parfois mystérieux mais les réussites sont le plus souvent

à la hauteur de l'énergie investie. Nous sommes fiers que vous ayez choisi Prolog IV comme outil de développement pour la résolution de vos problèmes et nous serons bien entendu très heureux de répondre à toutes vos remarques, critiques et suggestions.

1

Tutoriel et Survol de Prolog IV

1.1 Une session prolog sous Prolog IV

PROLOG IV est un langage de programmation. En tant que tel, il est doté des différents attributs d'un environnement de programmation : un compilateur et une bibliothèque. Le compilateur est utilisé pour transformer des programmes en une forme directement utilisable par la machine, et la bibliothèque contient tout le code prédéfini utilisé dans Prolog IV. Contrairement aux autres langages de programmation comme C ou C++, Prolog IV propose un mode interactif sous la forme d'un dialogue homme/machine de questions/réponses.

Pour démarrer une session Prolog IV vous devrez taper la commande `prolog4` sur votre ligne de commande, ou bien cliquer sur l'icone appropriée, ceci dépendant de la plate-forme que vous utilisez. Si vous avez des problèmes pour démarrer Prolog IV, faites-vous aider par votre administrateur système ou référez-vous au guide d'installation, fourni avec votre kit Prolog IV.

Prolog IV, c'est aussi prolog, c'est le but de cette mini-session

Si Prolog IV démarre normalement, vous verrez apparaître ce qui suit (aux date et numéro de version près) :

```
machine% prolog4

Prolog IV (beta III (204), cp4.8), Juin 1996 (C)PrologIA 1995,1996
[ Mon Jun 17 10:48:57 MET DST 1996 ]

>>
```

Le symbole `>>` est appelé l'*invite* (ou encore *prompt*). Elle indique que Prolog IV attend une requête. Tapons par exemple la requête «write(hello),nl.».

```
>> write(hello), nl.

hello
true.

>>
```

Vous ne devez bien sûr taper que la requête, c.à.d. ce qui est en gras, le reste est affiché en réponse par Prolog IV) . Dans notre exemple, notre requête

est constituée de deux littéraux séparés par une virgule. Le premier littéral, `write`, est une commande prédéfinie de Prolog IV qui peut donc être utilisée à tout moment. C'est une commande générale d'impression. Le littéral `nl` est aussi une commande prédéfinie, et imprime tout simplement un retour-chariot. On met toujours un point en fin de requête. Le mot `true` imprimé ensuite par Prolog IV indique que la commande a réussi, du moins d'un point de vue logique.

Maintenant, après avoir vu une requête, voici un programme. Il est composé de trois règles qui expriment que Jacques et Edouard vivent à Paris, et que Bernard habite Marseille.

```
>> consult.

Consulting ...
   habite(jacques,paris).
   habite(edouard,paris).
   habite(bernard,marseille).
   end_of_file.
true.

>>
```

La commande `consult` est une des commandes possibles permettant de rentrer un programme dans le système (on verra plus tard qu'il existe une commande `compile`, qui accepte en argument un nom de fichier). La ligne «`end_of_file.`», qu'il faut taper [1], indique au système la fin du programme et la terminaison du mode de consultation ; on revient donc au prompt Prolog IV.

Prolog IV enregistre les règles dans une *base de règles* et ce, dans l'ordre où elles sont tapées.

Essayons notre programme : Est-ce que Bernard habite Marseille ?

```
>> habite(bernard, marseille).

true.

>>
```

Dans cette requête, nous n'avons fait que vérifier ce que nous supposions déjà ; essayons maintenant d'interroger notre base de règle avec des questions du genre «Où habite Jacques ?».

Il est temps d'introduire les notions d'*identificateur* et de *variable*. Un identificateur est un nom ; il commence par une lettre minuscule[2]. Une variable est représentée par une suite de lettres qui commence par une lettre majuscule[3]. A toute variable est associé un domaine de valeurs possibles à un instant donné. Parmi ces valeurs possibles, il y a bien sûr les nombres, les identificateurs mais aussi les constructions arborescentes utilisant ces nombres et ces identificateurs ; on appelle arbres ces constructions.

1. Dans un shell unix, on peut se contenter de taper les touches `Control-D` .
2. Vous comprenez maintenant pourquoi nous avons mis des noms propres en minuscules dans notre programme ; ceci peut etre contourné, mais nous n'en parlerons que dans le chapitre décrivant la syntaxe.
3. Le chapitre syntaxe décrit précisement la syntaxe d'une variable.

Lors de la création d'une variable pendant l'exécution du programme, celle-ci peut prendre pour valeur chacun des arbres possibles du domaine sur lequel travaille Prolog IV. Le déroulement d'un programme Prolog IV réduit le domaine des variables qui participent à son exécution.

Posons maintenant nos questions qui comportent une ou plusieurs inconnues :

Où habite Jacques ?

```
>> habite(jacques, X).

X = paris.

>>
```

Qui habite Paris ?

```
>> habite(X,paris).

X = jacques;

X = edouard.

>>
```

Nous avons obtenu deux réponses à cette question, c.à.d. deux valeurs possibles pour la variable `X`. En Prolog IV, les solutions sont séparées par un point-virgule (`;`) et la dernière solution est suivie d'un point, tout comme les requêtes.

Qui habite où ?

```
>> habite(X,Y).

X = jacques,
Y = paris;

X = edouard,
Y = paris;

X = bernard,
Y = marseille.

>>
```

A chaque fois, le système répond par un ensemble de valeurs données aux variables de la requête (qui sont `X` et `Y`), valeurs qui satisfont à la relation `habite`.

Pour terminer une session Prolog IV, tapez la commande suivante :

```
>> halt.

machine%
```

et vous retournerez à l'interpréteur de commande de votre système d'exploitation. Si Prolog IV était utilisé depuis son environnement graphique, il faudrait actionner le bouton «Quit» de la palette principale.

1.2 D'autres exemples, avec des contraintes !

Jusqu'ici, nous n'avons vu qu'un petit programme Prolog IV (et même prolog) très simple, ne comportant d'autres contraintes que les égalites (l'unification). Prolog IV peut utiliser des programmes beaucoup plus complexes, et nous allons donner maintenant quelques exemples qui montrent comment manipuler les différents objets Prolog IV.

1.2.1 Les contraintes numériques

La partie la plus novatrice de Prolog IV est son traitement des nombres. Ses traitements de nombres devrait-on dire, car il y en a deux.

Quand le problème numérique posé est composé d'équations ou d'inéquations linéaires comme $\frac{3}{4}x - \frac{1}{5}y \geq 7z - 1$, c'est le solveur linéaire qui doit être utilisé et Prolog IV trouve les solutions exactes au problème.

Quand le problème numérique posé est composé d'éléments plus complexes, comme des mélanges d'opérations algébriques, de fonctions trigonométriques ou transcendentales (par exemple $x = 2sin^2x$), on utilisera le solveur dit «approché».

Commençons par examiner le solveur linéaire.

1.2.2 Contraintes numériques linéaires

Débutons par un petit exemple :

```
>> X+Y = 3 , X-Y = 1.

Y = 1,
X = 2.

>>
```

Ici, Prolog IV a été utilisé pour résoudre le système d'équations linéaires :

$$\begin{cases} x + y = 3 \\ x - y = 1 \end{cases}$$

Examinons maintenant un autre petit problème : nous avons un certain nombre de chats et d'oiseaux, ainsi que le nombre de leurs têtes et de leurs pattes, et nous nous demandons d'exprimer les relations qui lient ces nombres. Si nous appelons c le nombre de chats, v le nombre d'oiseaux (v pour volatile, o ressemblant trop à un zéro), t le nombre de têtes et p le nombre de pattes, nous avons les équations suivantes :

$$\begin{cases} t = c + v \\ p = 4c + 2v \end{cases}$$

qui sont bien linéaires (pas de produit ou de rapport de variables) et qui s'expriment naturellement en Prolog IV (en mettant bien les variables en majuscule !) :

```
>> T = C+V , P = 4*C+2*V .

P ~ real,
V ~ real,
C ~ real,
T ~ real.

>>
```

Prolog IV répond par une suite de contraintes qui indiquent que chacune des variables appartient à l'ensemble des nombres réels, (ce dont on se doutait). L'existence de cette réponse, dans le cadre du solveur linéaire, indique que le système est soluble. Allons un peu plus loin en donnant des valeurs à quelques unes des variables. Puisque le système est composé de deux équations à quatre inconnues, fixons la valeur de deux d'entre elles pour obtenir les valeurs des deux autres. Fixons le nombre de pattes à 14 et le nombre de têtes à 5 :

```
>> T = C+V , P = 4*C+2*V , P=14 , T=5 .

P = 14,
V = 3,
C = 2,
T = 5.

>>
```

Cette fois, Prolog IV nous dit que la solution est unique en exhibant les valeurs des variables pour lesquelles l'ensemble de contraintes est vérifié. Avant de poursuivre avec cet exemple, nous allons déclarer une règle qui lie les quatre variables et les équations, de sorte que nous n'ayons plus à écrire les équations à chaque fois :

```
>> consult.

Consulting ...
   chatsoiseaux(C,V,P,T) :- T = C+V , P = 4*C+2*V .
   end_of_file.
true.

>>
```

Nous venons d'introduire une nouvelle règle dont la signification est : *sous reserve que les deux contraintes (deux équations dans ce cas) soient vérifiées, la relation chatsoiseaux(c,v,p,t) est vraie.* Nous pouvons maintenant taper les requêtes suivantes :

```
>> chatsoiseaux(C,V,14,5).

C = 2,
V = 3.

>> chatsoiseaux(1,V,P,5).

V = 4,
P = 12.

>> chatsoiseaux(1,1,2,4).

false.

>> chatsoiseaux(C,V,6,4).

C = -1,
V = 5.

>>
```

Prolog IV nous repond `false` à la troisième requête, car l'ensemble des contraintes n'est pas soluble. Pour ce qui est de la quatrième, une des variables de la solution est négative. Il est vrai que lors de la formalisation de notre problème, nous n'avons jamais spécifié que les solutions devaient être positives. Nous devons donc réécrire notre relation `chatsoiseaux` pour contraindre les variables de notre relation à être positives. Ecrivons la nouvelle relation :

```
>> reconsult.

Reconsulting ...
   chatsoiseaux(C,V,P,T) :-
       T=C+V , P=4*C+2*V ,
       gelin(C,0), gelin(V,0), gelin(P,0), gelin(T,0).
   end_of_file.
true.

>>
```

La commande `reconsult` est utilisée pour remplacer un programme déjà défini avec une nouvelle version de celui-ci. Utiliser `consult` avec un programme déjà défini aurait amené à un message d'erreur. La contrainte `gelin(X,Y)` pose tout simplement la contrainte[4] $X \geq Y$. On a donc posé pour chacune des variables la contrainte $X \geq 0$.

4. Le nom $gelin$ vient de la juxtaposition de ge, qui est un raccourci de «greater or equal» (plus grand ou égal), et du suffixe lin qui rappelle que cette contrainte ne travaille que dans le linéaire. On peut trouver cette notation un peu lourde alors qu'existe le symbole conventionnel $\geq$ pour désigner cette relation d'ordre, mais d'une part la norme prolog ISO a reservé ce symbole à un usage précis (comparaison de nombres), ensuite il faut distinguer cette contrainte d'une autre contrainte de comparaison travaillant dans l'autre solveur numérique et enfin, c'est un nom officiel de Prolog IV, nom auquel on peut toujours se référer en toutes circonstances sans dépendre en aucune façon du contexte. Il sera fourni plus tard des «raccourcis» agréables qui altèrent la syntaxe..., pour le meilleur ou pour le pire de la lisibilité. Notons au passage l'existence des autres relations d'ordre fonctionnant dans le linéaire, qui sont $lelin$ «less or equal» (plus petit ou égal), $ltlin$ «less than» (plus petit strict) et $gtlin$ «greater than» (plus grand strict).

```
>> chatsoiseaux(C,V,6,4).

false.

>>
```

Prolog IV nous dit qu'il n'y a pas de solution à cette requête, ce que nous voulions. Mais maintenant, qu'arrive-t-il si nous donnons un nombre de pattes et un nombre de têtes tels qu'il n'y ait pas de solution entière, en mettant par exemple un nombre impair[5] de pattes ?

```
>> chatsoiseaux(C,V,7,3).

V = 5/2,
C = 1/2.

>>
```

Il nous est retourné un demi-chat et cinq-moitiés d'oiseaux... (une boucherie !) Il est vrai que nous n'avions pas la non plus specifié que les solutions devaient être entières. Nous avons utilisé de Prolog IV le solveur d'équations linéaires, qui raisonne sur l'ensemble des nombres rationnels (et donc pas seulement sur les entiers). Bien qu'il soit possible de contraindre une variable à être entière et d'affiner notre petit programme, nous nous arrêterons là avec cet exemple.

1.2.3 Contraintes sur les réels (solveur approché)

Prolog IV fournit un jeu de contraintes qui nous permet de raisonner sur les nombres réels. Nous prendrons bien soin de ne pas confondre les contraintes linéaires expliquées dans la section précédente avec les contraintes sur les réels que nous allons rapidement introduire ici. Il est difficile de raisonner précisement sur les nombres réels. Un nombre aussi simple que $\sqrt{2}$ n'est pas représenté exactement en machine avec Prolog IV.

Bien sûr, on peut calculer autant de chiffres que l'on veut et aller bien au delà de la précision de 1.414, mais nous ne pourrons jamais trouver (et donc à plus forte raison faire des opérations sur) toutes les décimales qui composent ce nombre.

Une alternative pourrait être d'implanter un solveur qui fait du calcul formel, mais il faut bien reconnaître que ces systêmes sont lourds et d'une certaine lenteur.

Ce qui nous reste donc est un solveur implantant un compromis entre performances (vitesse et consommation mémoire) et utilité de ce qui est déduit pour la résolution de problèmes.

Pour cette raison, ces contraintes sur les réels sont prises en compte par un solveur tout à fait différent du solveur complet des équations et inéquations linéaires.

Ceci nous amène à formuler les deux propriétés les plus importantes sur les contraintes sur les réels en Prolog IV :

- *La résolution d'*une contrainte de base *est* complète.
- *La résolution d'*un ensemble de contraintes *est* incomplète.

5. Les vertébrés ont normalement un nombre pair de pattes, et cette parité se transmet à toute collection de vertébrés. Mais c'est de la méta-connaissance et ceci déborde largement du sujet.

Par incomplet, nous voulons dire que Prolog IV peut ne pas détecter l'insolubilité de certains systèmes de contraintes qui n'ont pas de solution. Nous montrerons plus loin le pourquoi du passage de la complétude à l'incomplétude lorsqu'on passe de la résolution d'une contrainte à la résolution de plusieurs contraintes.

Bien sûr, l'incomplétude est *a priori* ennuyeuse, puisque décider si un système de contraintes est soluble ou pas est plus difficile. En fait, la façon dont ces contraintes sont traitées par Prolog IV nous garantit certains résultats.

Dans ce qui suit, on différencie le problème dit *exact*, qui est le problème originel (mathématique, physique, ...) du problème dit *approché*, qui est sa traduction naturelle en Prolog IV.

- Il est garanti que si un problème (système de contrainte) approché est déclaré insoluble par Prolog IV, alors le problème exact est tout aussi insoluble.
- Il est garanti que si le problème exact a une solution, alors elle se trouve parmi les solutions du problème approché (elle est donc trouvée par Prolog IV).
- Si Prolog IV trouve une solution au problème approché dans laquelle toutes les variables prennent une valeur unique, alors cette solution *est* aussi solution du problème exact.

Voyons quelques exemples. Essayons de calculer la valeur de $\sqrt{2}$. Il suffit de poser assez naturellement sous Prolog IV la contrainte $X = sqrt(2)$:

```
>> X = sqrt(2).

X ~ cc('>1.4142135','>1.4142136')

>>
```

L'expression $sqrt(2)$ est un *terme ensembliste* dit aussi *pseudo-terme*. Un pseudo-terme ne se distingue d'un terme que par le fait qu'il fait intervenir, à quelque niveau que ce soit, au moins un symbole de relation dont le nom et l'arité+1 sont réservés[6]. La définition est donc récursive : en effet soit le nœud a un nom de relation réservée, soit l'un des fils au moins du nœud est lui-même un pseudo-terme. Bien sûr tout ceci n'est pas exclusif et on peut fabriquer des imbrications assez diaboliques, avec plusieurs relations. Il faut noter qu'il n'y a pas de différences syntaxiques entre termes et pseudo-termes.

Les pseudo-termes numériques représentent des ensembles quelconques de réels. Parfois cet ensemble se réduit à un seul élément, nombre rationnel ou pas.

Pour en revenir à notre exemple, voyons qu'on peut tout aussi bien poser la contrainte à l'aide de la notation relationnelle plutôt que fonctionnelle, on obtiendra la même réponse (on voit mieux maintenant pourquoi la relation $sqrt$ est d'arité 2) :

6. Ils figurent dans une liste spéciale qu'on donnera plus loin dans le manuel. La relation $sqrt_2$ en fait partie.

```
>> sqrt(X, 2).

X ~ cc(`>1.4142135`,`>1.4142136`)

>>
```

Prolog IV répond en nous donnant le domaine de valeurs possibles pour la variable X, c'est à dire un sous-ensemble de **R**. Il ne s'agit toutefois pas de n'importe quel sous-ensemble de réels. Seulement une partie d'entre eux est utilisée par Prolog IV :

- les singletons formés d'un nombre rationnel (on parle alors dans ce cas-là de valeur plutôt que de domaine),
- les intervalles construits à l'aide des seuls nombres flottants simples de la norme IEEE (auxquels on se réfèrera plus tard sous le vocable de nombres flottants)[7] . Ces ensembles-là sont par ailleurs en nombre fini puisque formés à l'aide des seuls nombres flottants, qui sont en quantité finie[8] .

D'après nos principes, la solution (nous savons indépendamment de Prolog IV qu'elle existe) est à l'intérieur. Prolog IV nous a donné une réponse à l'aide des relations $\sim$ et cc. La contrainte $X \sim E$, avec X une variable signifie «X est élément de l'ensemble décrit par E». Par exemple $X \sim cc(1,2)$ signifie que X est élément de l'intervalle fermé[9] $[1,2]$.

Les valeurs des bornes sont des nombres rationnels[10] et la borne «`>1.4142135`» avec ces étranges décorations[11] vaut très précisément «le premier nombre flottant arrivant après le nombre décimal 1.4142135 .»

Comme on l'a vu précédemment, il est normal d'avoir un encadrement au lieu d'une simple valeur puisqu'il n'est pas possible de calculer (et à plus forte raison d'afficher) la valeur de $\sqrt{2}$. Prolog IV ne peut que retourner l'intervalle

$$[1.41421353816986083984375, 1.41421365737915039062 5]$$

qui est le plus petit intervalle possible qui contienne $\sqrt{2}$ et qui est tel que les deux bornes soient des nombres flottants en simple précision[12] . En fait, lors du traitement des contraintes sur les réels, Prolog IV effectue tous ses calculs de sous-domaines avec de tels intervalles.

Toutefois, Prolog IV peut calculer des valeurs exactes (qui sont donc des nombres rationnels), même quand il traite de telles contraintes :

```
>> X = sqrt(4).

X = 2.

>>
```

7. On peut travailler avec des unions d'intervalles en se plaçant dans un mode spécial.
8. L'ensemble des parties d'un ensemble fini est fini. Comme il y a environ 2^{32} nombres flottants, il y a donc environ $2^{2^{32}}$ sous-ensembles de nombres flottants.
9. *cc* est le mnémonique raccourci de *closed-closed* (fermé-fermé), désignant le type des bornes gauche et droite de l'intervalle. Il existe aussi bien sûr *co*, *oc* et *oo*, le *o* signifiant *open* (ouvert).
10. Les seuls nombres qui soient exprimables par Prolog IV ou le programmeur.
11. On aurait de même «`<1.4142135`» qui dénote le flottant précédant le nombre décimal 1.4142135 .
12. Prolog IV utilise une notation qui allège la suite de chiffres qu'il faudrait imprimer, sans perdre l'exactitude du nombre ainsi raccourci.

Dans ce cas, la valeur est précise, et nous sommes *sûrs* que c'est la solution de notre contrainte. Bien entendu, même dans l'exemple plus haut, la réponse était juste, quoique moins précisément exprimée car irrationnelle.

Regardons maintenant l'exemple suivant :

```
>> X = sqrt(2) , X = sqrt(2.0000001).

X ~ oo(‘>1.4142135‘,‘>1.4142136‘).

>>
```

La réponse est (presque) la même qu'avant et le résultat montre un succès bien qu'il soit évident pour nous que $\sqrt{2} \neq \sqrt{2.0000001}$. La raison en est qu'avec la précision donnée par les intervalles bornés par les nombres flottants, il n'est pas possible de faire la différence entre ces deux nombres, le pouvoir de résolution des nombres flottants [13] ne permettant pas de les distinguer. Toutefois, ce qui semble être une faiblesse respecte parfaitement un de nos principes «*s'il y a une solution du problème réel, elle se trouve dans les sous-domaines retournés par la résolution du problème approché.*»

Par acquis de conscience, donnons-nous le même exemple avec des nombres flottants séparables (il nous suffit d'enlever un 0) :

```
>> X = sqrt(2) , X = sqrt(2.000001).

false.

>>
```

Ouf !

Il faut toujours garder à l'esprit que le solveur des nombres réels ne travaille que sur le problème approché, pas sur le problème exact.

Voici maintenant la relation $square$. Elle ressemble assez à $sqrt$, mais ce n'est pas tout à fait la même chose puisque $sqrt(X)$ est le nombre positif dont le carré vaut X.

Jouons un peu avec toutes deux, en les utilisant de façon imbriquée pour changer :

```
>> square(sqrt(X)) = 2.

X ~ cc(‘>1.9999998‘,‘>2.0000004‘).

>> sqrt(square(X)) = 2, ge(X,0).

X = 2.

>>
```

La réponse à la première question donne une bonne approximation (bien lisible) de l'équation $(\sqrt{X})^2 = 2$ dont la solution est mathématiquement $X = 2$. La contrainte dans cette question n'est pas atomique, puisque formée de relations imbriquées.

13. Ce sont des nombres flottants IEEE en simple précision, celle-ci étant de l'ordre de 10^{-6} (relativement au nombre sur lequel elle s'applique, c.à.d. que la précision est de l'ordre de 10^{10} pour un nombre comme 10^{16}). L'exemple utilise des nombres séparés d'environ 10^{-7}.

La seconde question contient une nouvelle contrainte basée sur la relation ge [14] qui nous permet d'imposer à X d'être positif ($X \geq 0$). Bien que cette question comporte plusieurs contraintes approchées, sa réponse ne comporte que des valeurs (pas de domaine) ; elle est donc exacte, d'après l'un de nos principes (*Si Prolog IV trouve une solution au problème approché dans laquelle toutes les variables prennent une valeur, alors cette solution* est *aussi solution du problème exact*).

1.2.4 Le pourquoi de l'incomplétude

Voyons informellement comment se passe la résolution d'*une* contrainte :

1. A partir de la contrainte, on considère les domaines des arguments de sa relation (qu'on note par exemple $\mathcal{R}$) ; supposons la relation d'arité n, et appelons les domaines $D_1, \ldots, D_n$.
2. La contrainte est résolue complètement, c.à.d. on cherche le sous-ensemble E de $\mathbb{R}^n$ tel que $E = \mathcal{R} \cap D_1 \times \cdots \times D_n$. On supposera que E est non-vide (il y a sinon un échec.) Pour fixer les idées, imaginons que l'on est dans $\mathbb{R}^3$ et que notre ensemble ressemble à une patate.
3. On effectue une approximation de cet ensemble par le plus petit pavé de $\mathbb{R}^n$ qui l'englobe. Avec notre exemple, on obtient un parallèlépipède contenant la patate au plus près, c.à.d. la touchant sur chaque face [15].
4. Le pavé en question est bien sûr un produit cartésien d'intervalles de $\mathbb{R}$; on va approcher ceux-ci par des sous-domaines réels de Prolog IV. Chacune de ses dimensions fait l'objet d'une approximation par un ensemble :

   Si l'intervalle est réduit à un nombre, on le transforme en le plus petit intervalle flottant qui le contient.

   Sinon c'est un véritable intervalle et on arrondi chacune des bornes par le flottant le plus proche [16] par défaut ou par excès selon qu'il s'agit de la borne inférieure ou supérieure.

   Bref, quand les deux bornes d'un intervalle ne sont pas des nombres flottants, on approche celui-ci par un intervalle de nombres flottants, pris un peu plus large (juste ce qu'il faut) afin ne pas perdre de valeurs.
5. Ces domaines sont attribués aux variables qui apparaissaient en tant qu'argument. Le domaine de chacune des variables n'a pu que diminuer.

En plus de cette résolution existe un mécanisme prenant en compte exclusivement les nombres rationnels. Quand suffisamment d'arguments de la contrainte sont des nombres rationnels, il peut être déduit les valeurs (elles aussi rationnelles) des autres arguments.

La résolution d'*un ensemble* de contraintes se passe très simplement en traitant chaque contrainte (c'est une relation et des variables ou constantes en

14. ge est le raccourci de *greater or equal*. Il ne faut pas la confondre avec $gelin$!!! La relation ge pose une contrainte sur les réels, alors que $gelin$ permet de poser une contrainte linéaire qui n'est prise en compte que dans un autre solveur.

15. Donc le plus petit parallèlépipède possible.

16. Il peut s'agir du même borne si celle-ci est déjà un nombre flottant.

argument) comme indiqué ci-dessus. Les variables servent seules d'interface entre les contraintes, aussi un sous-domaine « surévalué » (qui approche trop grossièrement l'ensemble exact) peut éventuellement cacher une absence de solution mathématique. En effet, l'intersection de l'approximation de deux ensembles disjoints peut être non-vide.

Prenons par exemple la requête avec les contraintes suivantes :

```
>> X = oo(-1,1), Y = oo(-1,1), gt(X,Y).

Y ~ oo(-1,1),
X ~ oo(-1,1).

>> X = oo(-1,1), Y = oo(-1,1), lt(X,Y).

Y ~ oo(-1,1),
X ~ oo(-1,1).

>>
```

On obtient en réponse aux deux requêtes que les sous-domaines pour X et Y sont inchangés [17]. Ceci signifie que l'ensemble approximant la conjonction des trois contraintes de chaque requête est le pavé ouvert $(-1,1) \times (-1,1)$.

Et quand on met ensemble les quatres contraintes, on obtient...

```
>> X = oo(-1,1), Y = oo(-1,1), gt(X,Y), lt(X,Y).

Y ~ oo(-1,1),
X ~ oo(-1,1).
```

...un succès ! Alors que cet ensemble de contraintes est mathématiquement insoluble.

Nos ennuis viennent de la conjonction des deux contraintes $gt(X,Y)$ et $lt(X,Y)$. Chacune est traitée parfaitement, mais le traitement de l'ensemble des deux passe par l'utilisation de domaines grossiers (parce que se sont des pavés) qui servent d'interface (par le biais de variables communes). C'est cette faiblesse de représentation des domaines (voulue pour une implantation efficace) qui rend notre solveur incomplet.

Notes :

- Les contraintes de domaines (utilisant des relations comme *cc*, *oo*,...) ne posent pas de problèmes dans la mesure où ce qui est passé en argument est représentable par des nombres flottants.
- Il y a une autre cause d'incomplétude car il y a deux moments où il est fait une approximation pendant le traitement d'une contrainte (les points 3 et 4). Celle décrite au point 4 se présente moins fréquemment que celle que l'on vient de montrer : il s'agit de l'exemple avec $\sqrt{2}$ et $\sqrt{2.0000001}$, vu dans une section antérieure.
- Une contrainte de la forme $rel(X,X)$ n'est pas autre chose que la conjonction de contraintes $(\exists A)\ rel(X,A), eq(A,X)$ qui comporte donc *deux* contraintes et est donc soumise aux problèmes d'incomplétude précités.

17. On ne s'expliquera pas sur le choix d'intervalles ouverts dans ces exemples. Il permet de simplifier l'explication et de ne pas cumuler plusieurs effets.

1.2.5 Union d'intervalles

Oublions temporairement l'incomplétude en examinant d'autres requêtes et leurs réponses dans deux modes de fonctionnement du solveur de contraintes sur les réels.

```
>> X = sqrt(2).

X ~ cc(‘>1.4142135‘,‘>1.4142136‘).

>> square(X) = 2.

X ~ cc(-‘>1.4142136‘,‘>1.4142136‘)

>>
```

La première requête (qu'on connait déjà) rend un petit domaine alors que la dernière requête rend pour X un domaine de taille environ 2.8 d'amplitude (le premier nombre est négatif !). Nous savons que l'équation $X^2 = 2$ a deux racines $-\sqrt{2}$ et $\sqrt{2}$, et le plus petit intervalle qui les contient toutes deux est mathématiquement $[-\sqrt{2}, \sqrt{2}]$, intervalle dont Prolog IV donne une bonne approximation à travers le sous-domaine de la variable X.

Passons dans le mode Prolog IV *union d'intervalles* pour voir comment les choses se passent[18] :

```
>> set_prolog_flag(interval_mode, union).

true.

>> sqrt(square(X)) = 2.

X ~ -2 u 2.

>>
```

Nous rend X appartenant au sous-domaine formé des deux nombres -2 et 2, qui est une forme allégée de $\{-2\} \cup \{2\}$. Au passage, nous découvrons l'existence du pseudo-terme t_1 `u` t_2 (avec `u` pour union) qui est l'ensemble des arbres qui appartiennent à t_1 ou à t_2 (de façon non-exclusive).

En mode « intervalles simples » cette requête aurait pour réponse `X ~ cc(-2,2)`, et donc l'intervalle $[-2, 2]$.

La question suivante avait déjà été posée plus haut, en mode « intervalles simples ». Il y est maintenant répondu, en mode «union d'intervalles» :

```
>> square(X) = 2.

X ~ cc(-‘>1.4142136‘,-‘>1.4142135‘)u cc(‘>1.4142135‘,‘>1.4142136‘).

>>
```

Cette fois-ci, on obtient une union de deux petits intervalles respectivement autour de $-\sqrt{2}$ et de $\sqrt{2}$, au lieu de l'intervalle $[-\sqrt{2}, \sqrt{2}]$, obtenu en mode «intervalles simples». Dans certains cas, on a donc une meilleure séparation des solutions à l'aide du mode «union d'intervalles».

18. Pour en sortir, il suffira de taper le but « `set_prolog_flag(interval_mode, simple).` ».

1.2.6 Le quantificateur existentiel

Le quantificateur existentiel sert à construire une contrainte de la forme $X\ ex\ P$ dans laquelle X est une variable muette (ou existentielle) et P une contrainte (ou une conjonction de contraintes comme une requête). Elle se lit «*il existe* X *tel que* P» et a même valeur de vérité que P. Cette construction est une extension qui ne fait pas partie de la norme ISO. Pragmatiquement, cette construction signale à Prolog IV que la variable X n'a pas d'intérêt aux yeux du programmeur et que l'on n'est pas intéressé par sa valeur ou son sous-domaine, qui ne seront donc pas affichés en réponse quand on utilise cette construction dans une requête. Par exemple :

```
>> X = Y.

X = Y,
Y ~ tree.

>> X ex X = Y.

Y ~ tree.

>> X ex X = f(g(2),g(2)).

true.

>>
```

En réponse à la seconde requête, rien n'est affiché pour X. Dans la troisième, Prolog IV se contente de répondre que la requête est soluble.

Les variables existentielles permettent d'avoir à sa disposition des sortes de « variables locales », complètement déconnectées du contexte, elles ne sont donc pas liées avec l'extérieur de la construction ex.

Dans la requête suivante, la variable X joue deux rôles, dont un rôle tout à fait local dans la contrainte parenthésée. Il ne faut pas être étonné que X puisse valoir à la fois 1 et 3, puisqu'en fait il ne s'agit pas du tout du même X.

```
>> X = Y, (X ex X = 1), Y = 3.

Y = 3,
X = 3.

>>
```

Voici encore trois exemples utilisant la quantification existentielle.

```
>> X ex Y = f(X,X)

A ex
Y = f(A,A),
A ~ tree.

>> X ex Y = f(X).

Y ~ f(tree).

>> X ex Y ex Z = f(X,Y).

Z ~ f(tree,tree).

>>
```

Dans la première requête, on crée une contrainte comportant deux occurrences d'une même variable existentielle. Dans la réponse, une variable existentielle est créée (avec pour domaine l'ensemble des arbres) puisqu'elle est nécessaire pour refléter le fait que deux sous-arbres identiques inconnus figurent dans la valeur de Y.

Dans la seconde requête au contraire, il n'est pas nécessaire de créer une variable existentielle puisque l'arbre complètement inconnu n'apparait qu'une fois, Prolog IV s'est contenté de mettre à cette place le pseudo-terme $tree$, qui représente l'ensemble de tous les arbres.

La troisième requête ne sert ici qu'à montrer une imbrication de quantificateurs, et se lit : *il existe* X *tel que : il existe* Y *tel que :* Z *égale* ..., l'associativité se faisant de la droite vers la gauche.

1.2.7 Utiliser *compatible* ou *egal* ?

Quand doit-on mettre « $\sim$» ou «$=$» dans les contraintes ?

Ces deux relations $\sim$ (compatible) et $=$ (égal) sont très similaires et il faut bien avouer qu'elles semblent faire la même chose, dans le sens où le programmeur peut se tromper et écrire l'un pour l'autre, Prolog IV rectifiant de lui-même[19]. Par contre, Prolog IV ne se permet pas en sortie cet abus de notation et utilise toujours la bonne relation pour les réponses, en utilisant $=$ le plus souvent possible, et $\sim$ quand c'est nécessaire. La relation $=$ ne s'emploie qu'avec des termes, alors que $\sim$ seule accepte les pseudo-termes en membre gauche ou droit. Bref $=$ est l'égalité (unification dans les prologs standard) ordinaire, alors que $\sim$ se comporte selon le contexte de l'une des façons suivantes :

- $terme \sim terme$ est l'égalité (c'est exactement «$=$»).
- $terme \sim pseudoterme$ est vraie si l'individu représenté par $terme$ appartient à l'ensemble représenté par $pseudoterme$, fausse sinon.
- $pseudoterme \sim terme$ est vraie si l'individu représenté par $terme$ appartient à l'ensemble représenté par $pseudoterme$, fausse sinon.

19. La raison pour laquelle Prolog IV arrive à « rectifier » les $=$ abusifs en $\sim$ est qu'il n'y a pas d'unification sur les ensembles en Prolog IV, mais seulement des calculs d'intersection, internes et *a priori* inaccessibles. Il n'y a donc pas d'ambiguïté pour lui.

– $pseudoterme \sim pseudoterme$ est vraie si l'intersection des deux ensembles est non-vide, fausse sinon.

1.2.8 Les variables muettes *underscore*

Les variables *underscore* sont des variables muettes qui s'écrivent à l'aide du seul caractère «_». Chacune des occurrences représente une variable différente des autres. Ces variables sont tout simplement des variables existentielles qu'il n'est pas nécessaire de nommer et de quantifier.

```
>> X = f(_,_).

X ~ f(tree,tree).

>>
```

1.2.9 Relations et pseudo-termes

On a parfois montré à travers les exemples précédents des formes fonctionnelles (pseudo-terme) ou relationnelles (littéral) d'une même contrainte. Il faut savoir que toute contrainte peut être notée sous forme de relation ou de pseudo-terme (ensemble), la seule différence étant la position syntaxique et, bien sûr, l'arité.

Voici le rapport entre les deux concepts (et notations) que sont la relation et l'opération :

$$nom(X_0, X_1, \ldots, X_n) \equiv X_0 \sim nom(X_1, \ldots, X_n)$$

Comme on le voit, l'argument qui sert de «résultat» à l'opération est le premier de la relation. La formule précédente conserve l'ordre des arguments quand on lit chaque membre de l'équivalence de gauche à droite. Par exemple, les contraintes :

$ge(X, 0) \equiv X \sim ge(0)$ avec $ge(0)$ l'ensemble des nombres supérieurs ou égaux à zéro.

$u(X, 1, 2) \equiv X \sim u(1, 2)$ avec $u(1, 2)$[20] l'ensemble des deux nombres $\{1, 2\}$.

La présence de l'argument «résultat» en première position peut surprendre les habitués de prolog pour qui l'usage veut que les arguments de sortie viennent après ceux d'entrées, donc vers la fin du littéral, mais il faut se rappeler les points suivants :

– *ce qu'on appelle* «résultat» *n'est pas plus un argument de sortie qu'un autre, c'est tout au plus un argument privilegié quant à son extraction.*

– *Les relations sont rarement utilisées telles quelles, il faut leur préférer les notations fonctionnelles (pseudo-termes) qui sont la plupart du temps plus lisibles*[21].

– *L'argument* «résultat» *se trouve toujours au même endroit*[22].

20. u existant en tant qu'opérateur Prolog IV infixé, on peut faire aussi $1\ u\ 2$.

21. C'est particulèrement vrai lorsque les relations ont une forte connotation fonctionnelle, comme $plus$ ou cos.

22. Cette propriété n'est intéressante que pour la méta-programmation, où on manipule des règles et littéraux.

1.2.10 Des requêtes aussi puissantes qu'étonnantes

Dans tout ce qui suit, le programme consiste à chaque fois en une requête, sans avoir à définir de règles. On supposera sauf indication contraire qu'on est dans le mode «union d'intervalles».

L'ensemble des X tels que leur carré est strictement plus grand que 1 : (Ce qui suit un caractère `%` jusqu'à la fin de ligne est un commentaire prolog.)

```
>> gt(square(X),1).  % ou encore » square(X) ~ gt(1).

X ~ lt(-1)u gt(1).
```

X a pour domaine l'ensemble des nombres strictement plus petits[23] que -1 ou strictement plus grands que 1.

Une déduction élégante

Si Y est un carré ...

```
>> X ex square(X) = Y.

Y ~ ge(0).
```

... alors il est positif ou nul !

Il a été utilisé ici un quantificateur existentiel afin de ne pas voir apparaître inutilement X dans la réponse.

Quelques requêtes sur les nombres entiers

Voici le pseudo-terme XnY dont le n rappele le symbole $\cap$ de l'intersection. Il construit l'ensemble dont les éléments appartiennent à chacun des deux ensembles donnés en arguments[24]. Le pseudo-terme int représente l'ensemble des entiers relatifs. Quels sont les réels entre 1 et 9 qui sont des entiers ? (autrement dit quel est l'ensemble : $\{X | X \in [1,9] \cap \mathbb{N}\}$)

```
>> X ~ cc(1,9) n int.

X ~ 1 u 2 u 3 u 4 u 5 u 6 u 7 u 8 u 9.
```

Comme on est en mode «union», on obtient la liste en extension[25, 26].

23. *lt* provient de *less than* (plus petit strict). *gt* provient de *greater than* (plus grand strict). Il existe de même *le* pour *less or equal* (plus petit ou égal) et *ge* pour *greater or equal* (plus grand ou égal).

24. Quand l'un des arguments de n n'est pas un pseudo-terme (et donc pas un ensemble), alors l'argument est transformé en singleton contenant le terme (*idem* pour u (union))

25. en mode «simple», on obtient rien de plus comme domaine que `X ~ cc(1,9)`. En effet, int n'est pas un domaine, mais une contrainte, et Prolog IV n'affiche que les domaines et constructions dans ses réponses.

26. Si vous voulez savoir ce qui se passe quand on demande la liste en extension des entiers, allez directement à la section «**Problèmes**».

```
>> X  ~  cc(1,9) n times(2,int).      % les entiers pairs entre 1 et 9

X ~ 2 u 4 u 6 u 8.

>> X  ~  cc(1,9) n int n times(2,nint).  % les entiers impairs de 1..9

X ~ 1 u 3 u 5 u 7 u 9.
```

Le pseudo-terme $nint$[27] représente l'ensemble des réels qui ne sont pas des entiers. Le pseudo-terme $times(X, Y)$ construit l'ensemble des produits possibles de ses deux arguments. Pour exprimer que P est un nombre pair, il suffit de contraindre P a être le double d'un entier. Pour avoir un nombre impair, il faut le contraindre a être un entier qui est le double d'un nombre non-entier.

Les nombres premiers entre 10 et 100 (les diviseurs possibles à tester ne peuvent être que les nombres premiers de 2 à 7) :

```
>> X  ~  cc(10,100) n int n times(2,nint) n times(3,nint)
            n times(5,nint) n times(7,nint).

X ~ 11 u 13 u 17 u 19 u 23 u 29 u 31 u 37 u 41 u 43 u 47 u 53 u 59 u 61
u 67 u 71 u 73 u 79 u 83 u 89 u 97
```

Nombres rationnels et irrationnels

Le pseudo-terme pi représente l'ensemble des nombres égaux à π. Dans le monde réel, il n'y a bien sûr qu'une valeur possible, laquelle n'est pas représentable en Prolog IV, puisque irrationnelle. La seule déduction possible pour Prolog IV est le domaine de la relation, un petit intervalle ouvert.

```
>> X ~ pi.

X ~ oo(‘>3.1415925‘,‘>3.1415927‘).
```

Il ne faut pas croire pour autant que Prolog IV se laisse abuser facilement en lui donnant un nombre rationnel «humainement» proche de π[28] :

```
>> pi ~ 3.14159265358979323846264338327950288419716939937510582097494 4.

false.
```

Bien que l'on ait une réponse affirmative à la question :

```
>> oo(‘>3.1415925‘,‘>3.1415927‘) ~
        3.14159265358979323846264338327950288419716939937510582097494 4.
```

27. Le nom $nint$ provient de *not integer* (non-entier).

28. Faites-nous confiance, il s'agit bien des véritables soixante premières décimales de π.

En effet, une relation numérique dont tous les arguments sont des constantes est complètement vérifiée par Prolog IV. Dans le cas très simple de pi, les seules constantes numériques de Prolog IV étant les nombres rationnels et π étant irrationnel, la relation `pi(`$nombre$`)` est toujours fausse, puisque $nombre$ ne peut être qu'un rationnel.

Un peu de trigonométrie avec cosinus.

Quels sont les cosinus entiers d'angles compris entre 1 et 100 ? On peut formaliser cet ensemble par $\{y \mid \exists x, x \in [1,100],\ y = \cos x,\ y \in \mathbb{N}\}$, et le traduire immédiatement en la requête :

```
>> X ex X ~ cc(1,100), Y ~ cos(X) n int.

Y ~ -1 u 0 u 1
```

On pouvait tout aussi bien taper la requête suivante :

```
«Y ~ cos(cc(1,100)) n int».
```

Quels sont les x tels que $\cos x = 1$ *avec* $-10 \leq x \leq 10$ *?*

```
>> X ~ cc(-10,10), cos(X) ~ 1.

X ~ oo(-'>6.2831854',-'>6.283185')u 0 u oo('>6.283185','>6.2831854')
```

qui se traduirait mathématiquement par l'ensemble :

$$]-(2\pi+\varepsilon),\ -(2\pi-\varepsilon)[\ \cup\ \{0\}\ \cup\ ]2\pi-\varepsilon,\ 2\pi+\varepsilon[$$

On y reconnait l'approximation de l'ensemble $\{-2\pi, 0, 2\pi\}$.

Il y a bien entendu d'autres fonctions trigonométriques. Citons en vrac, outre cos, les fonctions sin, tan, cot, leurs réciproques $arcsin$, $arccos$, $arctan$ et les fonctions hyperboliques $cosh$, $sinh$, $tanh$ et $coth$.

Pour clôturer la liste des fonctions transcendentales, on citera aussi exp, ln et log (logarithme décimal).

1.2.11 Fonctions algébriques et autres

Sans grandes explications, voici quelques fonctions et opérations sur les réels. Les quatre opérations classiques sur les réels ($+$, $-$, $*$, $/$) deviennent quatre relations $plus$, $minus$, $times$, div) sur les arbres. Le plus et moins unaires deviennent quant à eux respectivement les relations $uplus$ et $uminus$.

Les notations avec pseudo-termes permettent de voir ces relations comme étant des opérations partielles sur les réels.

Pour des raisons pratiques, certaines relations ont un raccourci synonyme. En voici quelques-uns :

– La relation de symbole `.+.` est un synonyme[29] de la relation $plus$, l'ensemble des sommes de ses deux arguments ; c'est aussi un raccourci pour la relation $uplus$[30].

29. L'ajout de points permet de le distinguer du symbole +, qui lui travaille dans le solveur linéaire. Cette remarque vaut pour les quatres opérations.

– La relation `.-.` est synonyme de la relation $minus$ ou $uminus$.

– La relation `.*.` est synonyme de $times$.

– La relation `./.` est synonyme de div.

De plus, ces raccourcis peuvent être utilisés syntaxiquement comme opérateurs, par exemple `X.+.Y`.

Posons une question en utilisant ces raccourcis :

```
>> X ~ cc(-2,2), Y ~ cc(-1,1), Z ~ .-.sqrt(X.+.Z)./.2 .+. 3.*.Y .

Z ~ cc(-2,3),
Y ~ cc(-`<0.6666667`,1),
X ~ cc(-2,2).
```

Il existe d'autres pseudo-termes sur les réels, $floor$ et $ceil$, qui «rendent» respectivement le plus grand entier inférieur et le plus petit entier supérieur, à leur argument. On trouve aussi abs qui est bien sûr la valeur absolue de son argument, min et max qui «rendent» respectivement le plus petit et le plus grand de leurs deux arguments.

1.2.12 Deux mots sur la réduction des sous-domaines numériques

Voyons comment se propage la réduction des intervalles au travers de ce petit exemple mettant en œuvre l'addition. Soit l'équation $z = x+y$, avec $x \in [1,5]$ et $y \in [-1,8]$.

Quel est le domaine de z?

```
>> Z ~ X.+.Y, X ~ cc(1,5), Y ~ cc(-1,8).

Y ~ cc(-1,8),
X ~ cc(1,5),
Z ~ cc(0,13).
```

On obtient comme réponse que $z \in [0,13]$, le domaine des variables x et y étant inchangé .

Même question, mais en imposant à z d'appartenir à l'intervalle $[-1,1]$:

```
>> Z ~ X.+.Y, X ~ cc(1,2), Y ~ cc(-1,1), Z ~ cc(0,1).

Y ~ cc(-1,0),
X ~ cc(1,2),
Z ~ cc(0,1).
```

On obtient une réduction de domaine des variables x et y qui appartiennent respectivement à $[1,2]$ et $[-1,0]$, ainsi que z, qui ne peut plus qu'appartenir qu'à $[0,1]$.

Si on impose plutôt à z d'être négatif ou nul,

```
>> Z ~ X.+.Y, X ~ cc(1,5), Y ~ cc(-1,8), Z = le(0).

Y = -1,
X = 1,
Z = 0.
```

30. En réalité, le nom de relation est un couple ($symbole$,$arité$) qu'on note aussi $symbole/arité$, ou encore $symbole_{arité}$; on devrait donc parler des relations `.+.`$_3$ et `.+.`$_2$, ce qui lève toute ambiguïté.

on obtient ici des domaines restreints à une seule valeur pour chacune des variables. On sait alors en vertu d'un de nos principes que ce n'est plus une approximation du problème réel, mais que c'est sa solution exacte.

Remarques :

- L'ordre des contraintes n'a aucune importance.
- Le fait que les nombres soient des entiers ou non-entiers est sans importance.

1.2.13 Raccourcis et opérateurs

Voici une table donnant les raccourcis disponibles en Prolog IV, tout au moins en ce qui concerne les pseudo-termes. Ces raccourcis sont tous des opérateurs infixes ou préfixes.

Pseudo-termes	Raccourci
$uplus(t)$	`.+.` t
$uminus(t)$	`.-.` t
$plus(t_1, t_2)$	t_1 `.+.` t_2
$minus(t_1, t_2)$	t_1 `.-.` t_2
$times(t_1, t_2)$	t_1 `.*.` t_2
$div(t_1, t_2)$	t_1 `./.` t_2
$upluslin(t)$	`+` t
$uminuslin(t)$	`-` t
$pluslin(t_1, t_2)$	t_1 `+` t_2
$minuslin(t_1, t_2)$	t_1 `-` t_2
$timeslin(t_1, t_2)$	t_1 `*` t_2
$divlin(t_1, t_2)$	t_1 `/` t_2
$conc(t_1, t_2)$	t_1 `o` t_2
$n(t_1, t_2)$	t_1 `n` t_2
$u(t_1, t_2)$	t_1 `u` t_2
$index(t_1, t_2)$	t_1 `:` t_2

A savoir sur les priorités de ces opérateurs :

- Les opérateurs unaires (comme `.-.`) sont prioritaires sur les binaires.
- Les opérateurs multiplicatifs (comme `.*.` ou `./.`) sont prioritaires sur les opérateurs additifs (comme `+` ou `.+.`).
- L'associativité de ces opérateurs binaires *op* est gauche-droite, c.à.d. qu'une expression $t_1\ op\ t_2\ op\ t_3$ est interprétée $(t_1\ op\ t_2)\ op\ t_3$.
- L'intersection (n) est prioritaire sur l'union (u).

1.2.14 Enumération

A travers cet exemple sur les triangles pythagoriciens[31], nous allons effleurer l'énumération des entiers. Afin de limiter le nombre de solutions, on limitera volontairement les côtés des triangles à des valeurs comprises entre 1 et 10. On dit que X, Y et Z appartiennent à $[1, 10] \cap \mathbb{N}$ et on pose $Z^2 = X^2 + Y^2$.

31. Triangles rectangles de côtés entiers.

(le symbole `.+.` est un raccourci de la relation $plus$, qui est l'ensemble des sommes de ses deux arguments.)

```
% en mode "intervalles simples"
>> set_prolog_flag(interval_mode, simple).

true.

>> X ~ cc(1,10) n int, Y ~ cc(1,10) n int, Z ~ cc(1,10) n int,
      square(Z) ~ square(X).+.square(Y).

Z ~ cc(2,10),
Y ~ cc(1,9),
X ~ cc(1,9).

% en mode "union d'intervalles"
>> set_prolog_flag(interval_mode, union).

true.

>> X ~ cc(1,10) n int, Y ~ cc(1,10) n int, Z ~ cc(1,10) n int,
      square(Z) ~ square(X).+.square(Y).

Z ~ 5 u 10,
Y ~ 3 u 4 u 6 u 8,
X ~ 3 u 4 u 6 u 8.
```

On obtient dans les deux cas une solution unique qui contient les triplets qui nous intéressent. Ce qui nous est rendu est un pavé de $\mathbb{R}^3$ dans le premier cas, et un produit cartésien discret dans le cas des unions d'intervalles. On remarquera au passage que le sous-domaine de X est aussi celui de Y, ce qui n'est pas étonnant puisque ces variables jouent un rôle symétrique.

Ecrivons une petite relation comportant les contraintes relatives à un côté, afin de pouvoir taper des requêtes plus courtes :

```
>> consult.

Consulting ...
   cote(X,B) :- X ~ cc(1,B) n int.
   end_of_file.
true.

>> cote(X,10), cote(Y,10), cote(Z,10),
      square(Z) ~ square(X).+.square(Y).

Z ~ 5 u 10,
Y ~ 3 u 4 u 6 u 8,
X ~ 3 u 4 u 6 u 8

>>
```

Effectuons une énumération sur les variable X, Y et Z avec la primitive prédéfinie `intsplit` : on donne en argument à cette primitive une liste de variables dont on cherche des valeurs entières (qui respectent bien sûr le domaine de ces variables). On obtient alors par backtracking les quatre solutions :

```
>> X = cote(X,10), cote(Y,10), cote(Z,10),
      square(Z) ~ square(X).+.square(Y),
      intsplit([X,Y,Z]).

Z = 5,
Y = 4,
X = 3;

Z = 5,
Y = 3,
X = 4;

Z = 10,
Y = 8,
X = 6;

Z = 10,
Y = 6,
X = 8.

>>
```

1.2.15 Partitionnement (ou comment énumérer sur $\mathbb{R}$)

Soit à chercher les solutions (numériques) de l'équation $x = 2\sin x$, on écrit simplement en Prolog IV :

```
>> X = 2.*.sin(X).

X ~ cc(-2,2).
```

On obtient que l'ensemble des racines de l'équation, si elles existent, se situent dans l'intervalle $[-2, 2]$. C'est intéressant, mais comment obtenir quelque chose de plus précis ? Tous simplement en partitionnant le domaine de la variable X, en deux sous-intervalles par exemple, et pour chacun, voir si les contraintes sont toujours valides ou pas. Quand un intervalle est valide, on le coupe en deux, etc.... Bref, on espère par un partitionnement (dichotomique dans notre cas, mais ce n'est pas requis), élaguer des portions de domaines pour lesquelles il n'y a pas de solution ; on sait alors, par l'un de nos principes[32], qu'il n'y a pas de solution au problème réel, pour ces portions de domaine de X.

Le parallèle avec l'énumération est tout à fait justifié par le fait que le partitionnement de $\mathbb{R}$ s'effectue en suivant tout au plus la grille des nombres flottants, base de calcul des contraintes sur les réels. Ces nombres flottants étant en quantité finie, les intervalles constitués de deux nombres flottants sont tout aussi finis. Le partitionnement se fait donc en temps limité (même s'il peut être très long.)

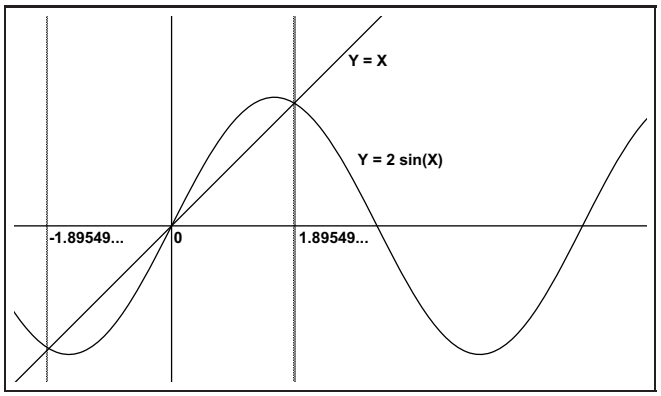
La primitive $realsplit$ se charge d'exhiber par backtracking les domaines où les contraintes restent satisfaites. Cette primitive accepte en argument une liste de variables v_i pour lesquelles on souhaite partitionner le domaine $D_{v_1} \times \cdots \times D_{v_i} \times \cdots \times D_{v_n}$.

32. «*Si un problème (système de contraintes) approché est déclaré insoluble par Prolog IV, alors le problème exact est tout aussi insoluble.* »

Améliorons nos résultats avec cette primitive : dans notre cas nous n'avons qu'une liste d'une seule variable :

```
>> X = 2.*.sin(X), realsplit([X]).

X ~ oo(-`>1.8954951`,-`>1.8954933`);

X ~ oc(-`>3.5762786e-7`,0);

X ~ oo(0,`<2.384186e-7`);

X ~ oo(`<1.895494`,`>1.8954945`).

>>
```

Il nous est rendu pour x quatre domaines disjoints assez fins, où peuvent se trouver les solutions du problème réel. Il n'est pas du ressort du solveur sur les réels d'en dire plus, nous avons obtenu là le maximum.[33]



Regardons les solutions obtenues et la figure ci-dessus, fruit d'une construction analytique sans rapport avec le raisonnement du solveur Prolog IV sur les réels ; repérons les principaux acteurs : les fonctions $y = x$, $y = 2\sin x$ et l'intersection de leurs courbes, en trois points. On remarque sur notre dessin que les solutions sont symétriques [34] par rapport à 0 .

La première et la quatrième réponse correspondent aux approximations des solutions opposés [35] $-1.89549\ldots$ et $1.89549\ldots$.

Quand aux deux autres solutions (la seconde et la troisième), elles forment en fait une seule solution séparée par les hasards du partitionnement dichotomique en deux bouts connexes [36]. C'est en fait un seul (petit) intervalle contenant 0.

33. Tout ce que nous pourrions essayer de faire est de substituer à x, dans l'equation, chacun (?!) des rationnels r qui appartiennent aux domaines, c.à.d. de faire $x = r, x = 2\sin x$ et voir s'il y a un succès ou un échec. Comme ce n'est guère praticable (il y a une infinité de nombres rationnels pour Prolog IV), on se contente tout au plus de tester les bornes fermées de ces domaines. On trouverait dans notre exemple la valeur 0 comme solution *certaine* de notre problème exact, en vertu d'un de nos principes sur les constantes.

34. Ne soyons pas étonnés, toutes les fonctions de notre problème sont impaires.

35. Nous seuls savons qu'elles sont opposées (pas Prolog IV). Les remarques sur les symétries du problème sont de la méta-connaissance.

36. Il n'y a strictement aucun réel entre ces intervalles.

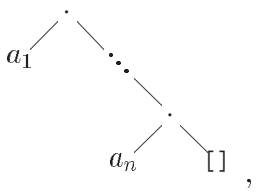
1.2.16 Arbres et listes

Le domaine de travail de Prolog IV est l'ensemble des arbres dont les nœuds sont étiquetés :

- soit par un nombre ; il n'y a alors pas de fils sous ce nœud.
- soit par un identificateur ; il peut y avoir alors zéro ou plusieurs fils.

On appelle *feuille* un arbre sans fils. Ces arbres se caractérisent également par le fait qu'ils ne comportent qu'un seul nœud.

Une *liste* est un arbre a de la forme



où les a_i sont des arbres quelconques. Dans notre texte, une telle liste est notée $[a_1, \ldots, a_n]$ et on désigne par $|a|$ la *longueur* n éventuellement nulle de la liste a. La concaténation de deux listes éventuellement vides est définie et notée par $[a_1, \ldots, a_m] \circ [a_{m+1}, \ldots, a_n] = [a_1, \ldots, a_n]$.

La liste vide est notée [ ] et c'est aussi une feuille et un identificateur. Elle a bien sûr pour longueur 0.

Le pseudo-terme $list$ représente l'ensemble de ces listes. Le pseudo-terme $X\,oY$[37] (ou encore $conc(X, Y)$) est la concaténation de deux listes.

Voici quelques exemples sur les listes et concaténations.

```
>> X ~ [1,2] o [3,4].

X = [1,2,3,4].

>> [1,2,3,4] ~ X o [3,4].

X = [1,2].

>> X ~ X o [].

X ~ list.

>> X ~ Y o [].

Y ~ list,
X ~ list.
```

La troisième requête ne donne pas de réponse pour X sinon qu'il appartient à l'ensemble des listes. La quatrième requête ne déduit rien sur X et Y[38] sinon leur domaine, car l'approximation de la concaténation utilisée dans le solveur des arbres, suffisante dans les cas courants, n'a pas assez d'informations pour conclure. En effet, dans le cas de la concaténation, il faut en général que deux des trois arguments soient des listes de longueurs connues pour déduire le

37. C'est la lettre o minuscule.

38. On pouvait s'attendre à obtenir $X = Y$, mais ce serait une déduction formelle, pas une déduction sur les domaines.

troisième argument, quel qu'il soit ; sinon, la contrainte $conc$ est retardée et seuls sont attribués les sous-domaines $list$ sur chacun des arguments.

Le pseudo-terme $size(L)$ apparaissant dans $N \sim size(L)$ est l'ensemble des tailles de la liste L. On peut aussi bien s'en servir pour determiner la taille d'une liste que pour la contraindre.

```
>> 10 ~ size(L).

L ~ [tree,tree,tree,tree,tree,tree,tree,tree,tree,tree].

>> N ~ size([1,2,3,4] o [5,6]).

N = 6.

>> N = size(L).

L ~ list,
N ~ ge(0).

>> lt(0) = size(L).

false.

>> N = le(0), N =size(L).

L = [],
N = 0.
```

La première requête a construit une liste de taille 10, garnie de dix éléments appartenant à l'ensemble des arbres. La seconde réponse nous donne la taille d'une liste de taille connue, ici 6. La troisième requête nous apprend que les listes ont une taille positive ou nulle. La quatrième nous confirme le fait précédent, la taille d'une liste ne peut être un nombre inférieur à 0. La cinquième nous indique qu'il n'y a qu'une seule liste de taille 0, c'est la liste vide [ ].

Encore plus loin avec les listes :

```
>> [a] o X  ~  X o [a] ,  10 ~ size(X).

X = [a,a,a,a,a,a,a,a,a,a].

>> [tree] o X  ~  X o [tree] ,  10 ~ size(X).

A ex
X = [A,A,A,A,A,A,A,A,A,A],
A ~ tree.
```

Les deux requêtes construisent des listes de tailles 10. La première est formée de dix occurrences de l'identificateur « a », et la seconde crée une liste de dix variables identiques (une variable existentielle de type $tree$ est donc créée pour pouvoir afficher la liste X.)

Apres avoir vu les deux contraintes de base sur les listes, tout au moins en ce qui concerne leur construction, nous allons maintenant introduire les relations $index$ et $inlist$.

La relation $index$, comme son nom le suggère, permet d'indexer[39] une liste comme on le ferait d'un tableau . Elle permet de poser la contrainte qu'un

39. Pour un i donné, prendre la $i$ème composante de ...

arbre A est le Nième élément de la liste L. Nous ne considérerons dans les exemples qui suivent que des listes de nombres, c'est là que les déductions de Prolog IV sont les plus fécondes.

Première requête : Où est 5 (à quel rang dans la liste) ?

```
>> 5 ~ index([1,3,2,4,6,5,8,10], N).

N = 6.
```

Où est 5 (à quel(s) rang(s) dans la liste) ?

```
>> 5 ~ index([1,3,2,4,6,5,5,8,10,5,13], N).

N ~ cc(6,10).
```

Il nous est repondu que l'indice N est entre 6 et 10 (en effet, 5 apparait plusieurs fois entre ces indices. Essayons d'obtenir mieux en mode «union».

```
>> set_prolog_flag(interval_mode, union).

true.

>> 5 ~ index([1,3,2,4,6,5,5,8,10,5,13], N).

N ~ 6 u 7 u 10.
```

On obtient cette fois-ci les valeurs exactes des rangs où se trouvent nos 5.

Un petit essai d'indexation avec le rang connu.

Quel est le neuvième élément de la liste ?

```
>> X ~ index([1,3,2,4,6,5,5,8,10,5,13], 9).

X = 10.
```

Un petit essai d'indexation avec le rang partiellement connu (entre 1 et 5).

```
>> X ~ index([1,3,2,4,6,5,5,8,10,5,13], N), N ~ cc(1,5).

N ~ cc(1,5),
X ~ cc(1,6).
```

Il est déduit que X est un réel entre 1 et 6.

Voyons maintenant le petit problème suivant : étant donnée une liste de nombres, nous voulons déterminer quels sont les éléments d'une liste L dont la valeur est leur rang dans cette liste, c.à.d. les x tels que $x = L_x$. Allons-y :

```
>> X ~ index([1,3,2,4,6,5,5,8,10,5,13], X).

X ~ cc(1,10).
```

Il nous est repondu que X peut être l'un des dix premiers éléments de la liste, qu'il vaut donc un entier entre 1 et 10 (13 a été éliminé). On aurait préféré en savoir plus sur X. Plaçons nous en mode «union» pour voir :

```
>> set_prolog_flag(interval_mode, union).

true.

>> X ~ index([1,3,2,4,6,5,5,8,10,5,13], X).

X ~ 1 u 2 u 3 u 4 u 5 u 6 u 8.
```

On obtient un meilleur résultat, puisque on n'a plus que sept valeurs possibles pour X au lieu de dix précédemment. Pour en savoir plus (en fait la réponse complète et définitive), effectuons une énumération sur X[40] :

```
>> X ~ index([1,3,2,4,6,5,5,8,10,5,13], X), intsplit([X]).

X = 1;

X = 4;

X = 8.
```

On a cette fois-ci les valeurs exactes (et non plus un sous-domaine) de X, le problème est donc complètement résolu puisqu'on a obtenu des constantes.

Pour voir ce qui se passe quand peu de choses sont connues :

X est le 3ème élément de la liste L :

```
>> X ~ index(L, 3).

L ~ [tree,tree,X|list],
X ~ tree.
```

Prolog IV crée le début de liste L, place X en troisième position puis termine avec quelque chose qui doit être une liste (il s'agit du pseudo-terme $list$.)

La relation $inlist$ ressemble beaucoup à $index$, mais en plus simple, car elle ne se préoccupe pas d'indexation, juste d'appartenance à la liste. Essayons $inlist$ avec, pour changer, une liste d'arbres, contraignons X à appartenir à la liste $[a, b, c]$ puis imposons à X une valeur toute autre :

```
>> X ~ inlist([a,b,c]), X = d.

false.
```

La relation if permet d'avoir un *si-alors-sinon* numérique piloté par la valeurs d'un argument (le premier). Voici sans autres commentaires quelques requêtes utilisant if :

40. Repasser en mode «intervalles simples» n'influe en rien sur la réponse.

```
>> X ~ if(B, 3,4).

B ~ cc(0,1),
X ~ cc(3,4).

>> X ~ if(B, 3,4), X = 5.

false.

>> X ~ if(B, 3,4), X ~ cc(0,3.5).

B = 1,
X = 3.
```

1.2.17 Booléens

Les booléens ne sont rien d'autre en Prolog IV que des entiers contraints à ne valoir que 0 (pour « faux ») ou 1 (pour « vrai »). Le premier avantage de ce choix est que ce sont des nombres réels comme les autres et sont donc utilisables dans n'importe quelle contrainte numérique (ou mixte, comme $index$). Il existe bien entendu toute une série de relations travaillant avec ces booléens, comme $impl$, and, or, xor et d'autres encore.

Traduisons en Prolog IV le fait que $a \Rightarrow b$, $b \Rightarrow c$ et $c \Rightarrow a$[41] :

```
>> impl(A,B), impl(B,C), impl(C,A).

C ~ cc(0,1),
B ~ cc(0,1),
A ~ cc(0,1).
```

On n'obtient pas d'autres déductions sur nos variables sinon qu'elles sont à valeurs dans $[0, 1]$. Fixons l'une d'entre elles à « vrai » (1) :

```
>> impl(A,B), impl(B,C), impl(C,A), C = 1.

C = 1,
B = 1,
A = 1.
```

On a cette fois-ci une déduction maximale sur nos variables.

Insistons sur le fait que les booléens sont astreints à ne valoir que 0 ou 1, et non pas seulement un réel entre 0 et 1. Ceci se voit immédiatement en passant simplement en mode «union», mais se voit aussi en mode «intervalles simples» de la façon suivante :

```
>> impl(B,_).

B ~ cc(0,1).

>> impl(B,_), B = co(-1,1).

B = 0.
```

La première requête montre un moyen correct de fabriquer un booléen. Dans la seconde requête, il est imposé à B d'appartenir à un l'intervalle contenant

41. Ceci traduit le fait que les variables a, b et c sont équivalentes, c.à.d. vraies en même temps, ou fausses en même temps.

0 mais *ouvert en 1*. Le booléen B prend alors instantanément la valeur 0 (la seule valeur booléenne qui appartienne au domaine donné).

Il existe toute une famille de contraintes faisant intervenir les booléens sous la forme d'une valeur de vérité. Leur nom commence par la lettre b et le pseudo-terme associé retourne une valeur booléenne qui représente la valeur de vérité de la contrainte obtenue en enlevant la première lettre (le b).

Par exemple, il existe la relation ble, qui rend le booléen associé à la valeur de vérité de la contrainte $le(X, Y)$, si on la posait. Un exemple sera plus clair.

B vaut «X est plus petit ou égal à 0 ». Que vaut B ?

```
>> B = ble(X, 0), X = 3.

X = 3,
B = 0.
```

La réponse est «faux» ($B = 0$).

Quels sont les x tels que $x \leq -1$ ET $1 \leq x$:

```
>> and(ble(X,-1), ble(1,X)).

false.
```

Il n'y en a bien sûr aucun.

Toutes ces relations permettent

- de retarder la pose d'une contrainte : la contrainte $le(X, -1)$ est posée dès que B est fixé à «vrai» (1)

```
>> B = ble(X,-1), B = 1.

B = 1,
X ~ le(-1).
```

- de contrôler la pose d'une contrainte (ou de son contraire) ; ici, selon la valeur de B, on pose la contrainte $le(X, -1)$ ou son contraire $lt(-1, X)$

```
>> B = ble(X,-1), B = 0.

B = 1,
X ~ gt(-1).
```

- d'élaborer des expressions booléennes complexes

```
>> and(binlist(X, [3,4,5]), bcc(X,4.5, 10)).

X = 5.
```

Citons pêle-mêle quelques unes de ces relations : $bimpl$, bor, $band$,..., bcc, bco,..., ble, blt, bge, bgt, beq, $bdif$[42] , $binlist$, $bint$, $bnint$, ...

42. $beq(X, Y)$ et $bdif(X, Y)$ sont respectivement les relations rendant la valeur de vérité des contraintes $eq(X, Y)$ ($X = Y$) et $dif(X, Y)$, si elles étaient posées.

1.2.18 Contraintes sur les arbres

Pour poser les contraintes portant sur les arbres (pour les tester ou les contraindre), on a à sa disposition un certain nombre de relations, liées aux sous-domaines de Prolog IV.

On trouvera la plupart du temps une relation d'appartenance à un sous-domaine nommé de Prolog IV, ainsi qu'une relation de non-appartenance à ce sous-domaine. Le pseudo-terme $nxxx(X)$ contraint X à ne pas vérifier xxx, donc à ne pas appartenir au sous-domaine xxx.

On trouve aussi les relations (qui commence par b) dont le pseudo-terme associé retourne une valeur booléenne qui represente la valeur de vérité de la contrainte obtenue en enlevant la première lettre (le b).

Appartenance aux arbres finis : $finite$, $infinite$, $bfinite$, $binfinite$.

Appartenance aux listes : $list$, $nlist$, $blist$, $bnlist$.

Appartenance aux feuilles : $leaf$, $nleaf$, $bleaf$, $bnleaf$.

Appartenance aux identificateurs : $identifier$, $nidentifier$, $bidentifier$, $bnidentifier$.

Appartenance aux nombres réels : $real$, $nreal$, $breal$, $bnreal$.

Quelques requêtes

```
>> X = f(f(X)).

X = f(X).

>> X ~ finite, X = f(f(X)).

false.

>> X = [A] n finite.

X = [A],
A ~ finite.

>> B = breal(B).

B = 1.

>> B = leaf n list.

B = [].
```

1.2.19 Contraintes et sous-domaines

Certaines relations permettent de contraindre leur argument à appartenir à un sous-domaine privilégié de Prolog IV, de même nom. Ceci nous amène à citer ces sous-domaines :

$tree$, $finite$, $list$, $nlist$, $leaf$, $nleaf$, $identifier$, $nidentifier$, $real$, $nreal$, $cc(r_1, r_2)$, $oo(r_1, r_2)$, ..., ou les r_i sont des nombres représentables par des nombres flottants.

Ce sont aussi bien des noms de relations que des noms de sous-domaines, elles peuvent donc apparaître dans les réponses de Prolog IV, avec la relation *compatible* ($X \sim sousDomaine$). Certaines d'entres-elles sont des négations d'autres ($nxxx(X)$ indique que X n'appartient pas au sous-domaine xxx .)

Une intersection de sous-domaine étant un sous-domaine, la relation n peut intervenir dans les sorties.

Dans le mode «unions d'intervalles», Prolog IV est susceptible d'afficher des unions d'ensembles (la relation u). C'est notamment le cas pour les résultats provenant de contraintes sur les réels.

NOTES :

- int et $nint$ ne sont pas des sous-domaines, seulement des contraintes.
- $infinite$ n'est pas un sous-domaine, seulement une contrainte.

1.3 L'environnement de programmation

On décrit dans cette section diverses possibilités du logiciel qui ne font pas partie du langage proprement dit. Il peut s'agir aussi bien de primitives que de modes de fonctionnement et de paramétrage.

1.3.1 Quelques options de la ligne de commande

La plupart des options de la ligne de commande concerne le paramétrage des piles et espace de travail utilisés par Prolog IV. D'autres options positionnent des modes de fonctionnement de Prolog IV.

Ces options peuvent tout aussi bien être fournies à Prolog IV lorsqu'il démarre d'un terminal que lancé sous son interface graphique (où ces options peuvent être données au travers d'un dialogue de préférences).

Dans ce qui suit, on appelle cellule un double mot-machine[43].

Voici quelques unes de ces options, avec entre crochet la valeur par défaut donnée par PrologIA, lorsque cette information est appropriée. Nb est un entier positif à fournir.

`-help`	Affiche la liste des options disponibles.
`-heap` Nb	Attribue à la pile $heap$ la place pour nb cellules [700000].
`-trail` Nb	Nb double-cellules pour la pile $trail$ [400000].
`-env` Nb	Nb cellules pour la pile env [50000].
`-choice` Nb	Nb cell. pour la pile $choice$ [50000].
`-global` Nb	Nb cell. pour la zone statique (consult,record,...) [500000].
`-local` Nb	Nb cell. pour la pile locale (unification, solveur de contraintes) [5000].
`-work` Nb	Nb cell. pour la zone temporaire (calculs rationnels, Gauss) [50000].
`-nbpcindex` Nb	Nb maximum de littéraux pour les programmes compilés (Pcode) [10000].
`-union`	Démarre Prolog IV en mode «unions d'intervalle» [intervalles «simples»].
`-iso`	Démarre Prolog IV en mode «iso étendu» [mode «prolog4»].

43. Une cellule occupe donc huit octets sur une machine 32 bits.

1.3.2 Prolog IV et la norme ISO

Prolog IV respecte la norme ISO, tout en l'étendant, les primitives de la norme sont implantées[44] et les mécanisme de base sont conformes à ce qu'édicte le standard.

Toutefois, afin de ne pas pénaliser lourdement par une syntaxe rigide l'usage des innovations qui caractérisent davantage Prolog IV que sa seule adhérence à la norme, deux modes de fonctionnement de Prolog IV sont disponibles, et on choisira son mode en fonction de ses propres goûts, selon que l'on souhaite suivre le standard de près ou au contraire choisir le mode «naturel» de Prolog IV.

Depuis le début de ce tutorial, le mode de fonctionnement est le mode « `prolog4` »[45] que nous ne saurons jamais assez vous conseiller d'adopter (c'est ce mode qui est positionné par défaut au démarrage de Prolog IV). Passer de l'un à l'autre est toutefois facile :

```
>> iso.

true.

?-
```

On remarque que l'invite est maintenant devenue «`?-`».

Pour revenir dans le mode naturel de Prolog IV dit mode `prolog4` :

```
?- prolog4.

true.

>>
```

Quelles sont les différences entre ces deux modes ? Les voici, en désordre :

– Les pseudo-termes ne sont pas compris par `consult` (ou `compile`) lorsque l'on est en mode iso. Ceci impose la notation relationnelle comme la seule permettant d'écrire des contraintes sous ce mode. La notation fonctionnelle ne construit que des termes. Les pseudo-termes restent donc de vrais termes, non évalués. Par exemple :

```
?- X = 3 + X, functor(X, N,A).

A = 2,
N = (+),
X = 3+X.
```

Outre la surprise que procurerait le succès d'une telle équation numérique si c'en était une, on trouve à l'aide de la primitive `functor`[46] que X est un arbre dont le nœud principal est étiqueté par + et d'arité deux, et pas du tout un nombre ni une variable numérique.

– Les nombres ayant la syntaxe d'un nombre flottant sont codés comme des nombres flottants IEEE lorsque l'on est en mode « iso », et non

44. Celles qui ne le sont pas encore dans une première version le seront rapidement.

45. Par opposition au mode «`iso`».

46. functor est une primitive de la norme qui dans notre cas retourne le nom et l'arité du nœud principal du terme, donné en premier argument.

pas comme des nombres rationnels en précision parfaite du mode `prolog4`.

```
?- X  = 1.00000000000000000000000000001.

X = 1.0.
```

– La forme $nom/arite$, fréquente lors de la gestion de règles dans la norme ISO, n'est pas disponible dans le mode `prolog4` (cette forme est interprétée comme une division retardée dans le solveur linéaire). Dans cet exemple, la contrainte $divlin(X, toto, 4)$ (de raccourci /) est fausse car $toto$ n'est pas un nombre.

```
>> X = toto/4.

false.
```

1.3.3 Les entrées/sorties de Prolog IV

Un ensemble de primitives qui gèrent les entrées/sorties est disponible en Prolog IV. Ces primitives permettent :

– l'ouverture de fichiers, que ce soit en lecture ou en écriture (`open`),

– la lecture de terme dans un fichier ouvert en lecture (`read`),

– l'écriture de caractères ou de termes dans un fichier ouvert en écriture (`write` et `nl`),

– la fermeture de fichiers (`close`).

– d'obtenir des informations sur les fichiers ouverts.

Introduisons approximativement la notion de descripteur de flot[47]. Un descripteur de flot est une entitée prolog qui permet de référencer, dans un programme, tout fichier ouvert. C'est bien sûr un terme prolog[48].

Montrons ceci sur un exemple. Supposons qu'il existe un fichier nommé `monfichier.p4` dans le répertoire courant, contenant le texte suivant :

```
toto(X,h(u),X).
[a,b,3333, 44444444444444,
555555555555, X].
```

Essayons de lire les deux termes (se terminant chacun par un point) de ce fichier ; on s'y prend de la manière suivante :

```
>> open('monfichier.p4', read, S), read(S,T1), read(S, T2), close(S).

A ex
T2 ~ [a,b,3333,44444444444444,555555555555,tree],
T1 = toto(A,h(u),A),
S = 4443940,
A ~ tree.

>>
```

47. Un flot d'entrée est un «endroit» (fichier par exemple) ou sont disponibles un certain nombre de caractères. Un flot de sortie est quant à lui un «endroit» ou on peut déposer des caractères.

48. C'est un nombre dans l'implémentation actuelle de Prolog IV.

la requête contient un appel à `open`, qui est la primitive générale d'ouverture de flot à qui on précise qu'on veut lire (par le mode *read*) dans le fichier `monfichier.p4`[49], suivie de deux appels à la primitive `read` qui lit un terme[50], on ferme enfin le descripteur de flot à l'aide de la primitive `close` de fermeture de flot. Nous avons récupéré nos deux termes dans les variables `T1` et `T2`. La variable `S` contient le descripteur de flot, et la valeur qu'elle possède après l'appel à la primitive `open` n'est guère destinée qu'à être transmise aux autres primitives d'entrée/sortie. Elle est sans doute différente dans votre session.

Essayons maintenant d'écrire des caractères ou des termes dans un fichier `ecr.p4`. On peut faire :

```
>> open('ecr.p4', write, S),
      write(S,'Salut tout le monde, voici un terme :'), nl(S),
      write(S, toto(h,10/20)), nl(S),
      close(S).

S = 4443940.

>>
```

On reconnait la primitive `open`, et on lui précise cette fois-ci que le fichier doit être ouvert pour y écrire[51] (par le mode *write*), ensuite nous retrouvons de vieilles connaissances en les primitives `write` et `nl` rencontrées au tout début de ce chapitre. Elles sont utilisées de façon légèrement différentes, avec un argument supplémentaire au début qui est le descripteur de flot retourné par `open`. On termine encore une fois par un appel à `close`.

Le fichier `ecr.p4` visualisé par ailleurs, contient maintenant les deux lignes :

```
Salut tout le monde, voici un terme :
toto(h,1/2)
```

1.3.4 L'entrée des règles

Nous avons tout au long de ce tutorial abordé de façon quelque peu légère l'entrée des règles sous Prolog IV, puisque l'accent a davantage été porté sur les possibilités du langage par le biais des contraintes prédéfinies. Nous avons rapidement vu les primitives `consult` et `reconsult` permettant de «donner» des règles à Prolog IV. Nous les rappelons ici, avec leurs différentes formes.

Auparavant, quelques notions élémentaires sur les règles prolog.

Paquets de règles

On appelle nom et arité d'une règle le nom et l'arité de la tête de cette règle.

On appelle paquet de règles un ensemble de règles de même nom et arité. Un paquet est donc parfaitement repéré par ce couple (nom, $arité$). L'usage veut

49. Le nom de fichier doit être un atome dans l'implantation actuelle de Prolog IV. Comme le nom `monfichier.p4` ne contient pas que des lettres ou chiffres, il faut le mettre entre apostrophes (`'...'`).

50. `read` ne peut lire qu'un terme suivi d'un point, suivi d'un espace ou retour-chariot. Ceci est du à la richesse (complexité ?) syntaxique des termes prolog, qui impose une convention pour arrêter l'analyse d'un terme prolog en cours de lecture.

51. Le fichier est créé s'il n'existe pas déjà, préalablement effacé sinon.

que les règles de même nom et arité soit regroupées dans un paquet, et non pas éparpillées dans un ou plusieurs fichiers [52].

Un nom de paquet sera souvent noté $nom/arité$.

Compilation, consultation

Il existe en Prolog IV trois façons d'entrer des règles :

- en compilant un fichier,
- en consultant un fichier (ou la console, qui n'est autre chose qu'un fichier spécial toujours ouvert),
- en assertant des règles (à partir de termes) pendant l'exécution d'un programme Prolog IV.

Il existe deux formes de codage de règle en Prolog IV, dues à la présence d'impératifs techniques contradictoires :

- la forme compilée (les primitives de la famille `compile`). Elle a pour avantage la vitesse d'exécution de ces règles, une fois appelées. Son inconvénient est l'impossibilité de traduire ces règles en termes, et de modifier dynamiquement les paquets de règles ainsi compilés (autrement qu'en les détruisant).
- la forme interprétée (ou assertée) (les primitives des familles `consult` ou `assert`). Son avantage est la possibilité de traduire ces règles en termes, et de modifier dynamiquement les paquets de règles (en ajoutant ou supprimant une règle donnée du paquet). Son inconvénient est une plus grande lenteur d'exécution, ainsi qu'une plus grande consommation mémoire.

Malgré ce qui a été montré tout au long de ce chapitre, c'est le mode d'entrée de règles par compilation qui est fortement préconisé. Il est en effet beaucoup plus rare d'avoir besoin d'une gestion dynamique de règles.

Les primitives de lecture de règles

`consult` : lit une suite de règles dans l'entrée courante. Si un paquet de règles lu avait préalablement été entré dans Prolog IV, une erreur indiquant une tentative de redéfinition surviendrait. Si on souhaite vraiment redéfinir ce paquet, il faut utiliser la primitive `reconsult`.

`consult(`*fichier*`)` : se comporte de même que `consult` sans argument, mais les paquets de règles sont lus dans un fichier de nom *fichier*.

`reconsult` : lit une suite de paquets de règles dans l'entrée courante. Ceux qui existaient déjà avec ce même nom et arité sont détruits si ce sont des règles utilisateurs [53] seulement.

`reconsult(`*fichier*`)` : se comporte de même que `reconsult` sans argument, mais les paquets de règles sont lus dans un fichier de nom *fichier*.

`compile(`*fichier*`)` : se comporte un peu comme consult, mais utilise la compilation des règles comme forme de codage interne.

52. Il est possible pour les rares cas ou c'est indispensable d'éparpiller des règles au moyen de déclarations par des directives. Ce n'est peut-être pas implanté dans la version actuelle. . . .

53. Ce sont toutes les règles qui ne font pas partie du système Prolog IV.

`recompile(`*`fichier`*`)` : se comporte un peu comme reconsult, mais utilise la compilation des règles comme forme de codage interne.

Dans tous les cas :

- La fin de fichier ou le terme «`end_of_file.`» arrêtent le lecteur de règles.
- Les règles sont codées en mémoire.
- Tenter de redéfinir des règles prédéfinies amène à une erreur.

Note : On ne peut pas compiler de règle dans la console, il faut passer par un fichier[54].

1.3.5 Factorisation des sorties

En positionnant cette option, l'affichage des sorties est davantage condensé en factorisant les sous-termes, quitte à introduire des variables existentielles (qui n'étaient donc pas dans la requête).

```
>> record(q__factorize, 1).

true.

>> X = f(Y), Y = f(X).

X = Y,
Y = f(Y).

>> X = f(g(2),g(2)).

A ex
X = f(A,A),
A = g(2).

>>
```

Au lieu des réponses «non factorisées», ce qu'on obtient par défaut :

```
>> record(q__factorize, 0).

true.

>> X = f(Y), Y = f(X).

X = f(X),
Y = f(Y).

>> X = f(g(2),g(2)).

X = f(g(2),g(2)).

>>
```

54. On peut cependant, lorsqu'on est sous unix et dans un terminal `tty` (comme une fenêtre `xterm` ou `cmdtool`) utiliser le pseudo-fichier `/dev/tty`, comme dans `compile('/dev/tty')`, qui attendra des buts dans le terminal, et ce jusqu'à la fin de fichier (généralement obtenue en tapant `CTRL-D`)

1.3.6 Problèmes

Il est décrit dans cette section quelques pièges et problèmes se présentant de temps à autres, le plus souvent par inattention.

Mélange de solveurs numériques

On cherche à calculer les valeurs possibles pour X à partir d'une formule numérique.

```
>> X = cc(1,2).*.3 - exp(1).

X ~ real.
```

La réponse n'est pas du tout satisfaisante. On pouvait s'attendre à trouver pour X un domaine numérique plus fin, plutôt que l'ensemble des réels. En fait, en regardant plus attentivement la requête, on peut se rendre compte que le signe `-`, qui est le raccourci de la relation $minuslin$ fonctionnant dans le solveur linéaire, a été utilisé au sein d'une contrainte sur les réels, ce qui est fortement déconseillé. Bref, **il ne faut pas mélanger des contraintes linéaires et des contraintes sur les réels.**

La correction consiste simplement à utiliser le signe `.-.` qui est le raccourci de la relation $minus$, la soustraction dans le solveur sur les réels.

```
>> X = cc(1,2).*.3.-.exp(1).

X ~ oo(`>0.281718`,`>3.2817182`).
```

La même faute courante :

```
>> X = -pi.

X ~ real.
```

donne un résultat vraiment peu informatif. Il a été utilisé le moins unaire du solveur linéaire au lieu du moins unaire du solveur sur les réels. Il fallait faire :

```
>> X = .-.pi.

X ~ oo(-`>3.1415927`,-`>3.1415925`).
```

Qui est bien l'approximation de $-\pi$.

Débordement en mode «union»

Le but de cette partie est essentiellement de montrer que les contraintes en rapport avec les entiers ainsi que les fonctions périodiques peuvent faire déborder Prolog IV en mode «union».

Tapons les requêtes suivantes :

```
>> set_prolog_flag(interval_mode, union).

true

>> X ~ int, X ~ cc(0,10).

error: heap_overflow
>>
```

On a demandé l'ensemble des entiers compris entre 1 et 10, et on obtient en fin de compte une erreur de débordement de la pile principale. Que c'est-il passé ?

Et bien, Prolog IV a tenté de construire l'union constituée par l'ensemble des entiers, tout au moins ceux qui sont dans l'approximation de l'ensemble des réels, c.à.d. les entiers qui sont représentables par des nombres flottants. Ces entiers sont en nombre fini, mais il n'est pas possible de représenter cet ensemble en mémoire, tout au moins avec les configurations matérielles qu'on trouve fréquemment sur le marché (de quelques dizaines de mégaoctets de mémoire vive).

Il faut donc, pour éviter de tomber sur ce débordement de mémoire, penser à réduire *avant* le domaine de la variable que l'on souhaite contraindre à être entière. Dans la plupart des cas, il suffira de réordonner des contraintes afin que les contraintes « gloutonnes » soient posées après celles qui effectuent de grosses coupes dans le domaine des variables à contraindre. On a maintenant ce qu'on veut :

```
>> X ~ cc(0,10), X ~ int.

X ~ 0 u 1 u 2 u 3 u 4 u 5 u 6 u 7 u 8 u 9 u 10.

>>
```

Quelles sont les contraintes «*gloutonnes* »*?*

Ce sont les contraintes faisant intervenir les entiers, ainsi que celles en rapport avec les fonctions périodiques[55] . La liste explicite est celle-ci : int, $nint$, $bint$, $bnint$, $floor$, $ceil$, sin, cos, tan et cot.

Il faut tout de même préciser que le compilateur réordonne tout seul les contraintes au sein d'un pseudo-terme. On peut donc sans crainte tout aussi bien écrire les deux requêtes :

```
>> X ~ cc(0,10) n int.

X ~ 0 u 1 u 2 u 3 u 4 u 5 u 6 u 7 u 8 u 9 u 10.

>> X ~ int n cc(0,10).

X ~ 0 u 1 u 2 u 3 u 4 u 5 u 6 u 7 u 8 u 9 u 10.

>>
```

Les requêtes sont transformées avec le compilateur. Les programmes le sont si les primitives de la famille `compile` sont utilisées. Les primitives de la famille `consult` ne réordonnent pas les contraintes.

Insistons sur le fait que ces débordements ne peuvent se produire avec ces contraintes qu'en mode « union d'intervalles ».

1.3.7 La mise au point de programme (Déboguage)

Prolog IV dispose d'un débogueur facilitant la mise au point des programmes. Celui-ci implémente le modèle des boîtes que l'on trouve dans certains

55. C'est de façon générale les contraintes dont la fonction associée ou sa réciproque est non-continue.

systèmes prolog, avec quelques améliorations en rapport avec les nouveautés de Prolog IV. Bien qu'il puisse fonctionner dans un environnement textuel pauvre comme les fenêtres `tty` (du genre `xterm` ou `cmdtool`), c'est sous l'environnement graphique de Prolog IV que le débogueur acquiert sa pleine puissance et sa simplicité d'utilisation. On a alors sous cet environnement la possibilité de suivre le déroulement de l'exécution dans les fichiers sources.

Tout ceci est décrit dans les chapitres « Environnement » et « Environnement Graphique ».

1.4 De Prolog III à Prolog IV

La présente section est une rapide introduction à Prolog IV, plus particulièrement destinée aux utilisateurs de Prolog III. Nous y décrivons donc les différences majeures entre Prolog III et Prolog IV.

1.4.1 Prolog IV est Prolog

La première affirmation importante concernant Prolog IV est :

"Prolog IV est Prolog"

Bien entendu, cela est vrai d'un point de vue historique puisque les concepts à la base de Prolog IV découlent directement des principes de la première version de Prolog, au début des années 1970. Toutefois, l'importance de notre affirmation initiale provient principalement du fait que Prolog IV est une extension de Prolog qui est conforme à la norme ISO pour Prolog de 1995. Pour les utilisateurs de Prolog, cela signifie qu'un programme développé en Prolog standard peut être compilé et exécuté en Prolog IV sans changement. Par exemple, le programme suivant est un programme Prolog IV et son exécution donnera le résultat défini par la norme ISO :

```
foo([X|L]) :-
    X =.. [f,1,Y] ,
    ( Y = 1 -> bar1(Y) ; bar2(Y) ) ,
    foo(L).
```

Notre affirmation a un sens complètement différent pour les utilisateurs de Prolog III. Puisque Prolog III n'est pas une extension de la norme ISO, le programme Prolog III suivant n'est donc *PAS* un programme Prolog IV :

```
Sumlist(<>,x) -> { x = 0 } ;
Sumlist(<x>.l,x+s) ->
  Sumlist(l,s) ;
```

En fait, il y a bien des différences entre cet échantillon de code et le standard ISO. Les quatre différences principales sont :

- Les symboles `->` et `;` doivent être remplacés par les symboles `:-` et `.` et la syntaxe des identificateurs et des variables a changé (nous retrouvons donc une syntaxe très proche de la syntaxe Edimbourg de Prolog III).
- Il n'est plus possible dans le standard d'utiliser les accolades pour entourer le système de contraintes, qui devra donc être intégré dans le programme.

– Les listes Prolog III (tuples) ne faisant pas partie du standard, les listes Edimbourg (utilisant `[` et `]`) doivent être utilisées systématiquement.

– Le symbole d'addition + n'est pas interprété en Prolog standard, ce qui signifie que le terme `x+s` est *a priori* interprété comme l'arbre `+(x,s)`.

En Prolog IV, toutes ces caractéristiques ont changé, de manière à être conforme à la norme ISO. Le programme qui calcule la somme des éléments d'une liste peut donc s'écrire de la manière suivante en Prolog IV(en mode ISO) :

```
sumlist( [] , 0 ).
  sumlist( [X|L] , S1 ) :-
    pluslin( S1 , X , S ) ,
    sumlist( L , S ).
```

Ici, nous avons utilisé le *prédicat prédéfini* `pluslin`, qui ajoute au système courant de contraintes la contrainte $S1 = X + S$. La principale raison de ce changement est que le standard ISO-Prolog indique clairement que les termes présents dans les littéraux doivent être interprétés comme des arbres. Avec cette restriction, la seule solution pour étendre le langage est donc d'ajouter un ensemble de prédicats prédéfinis comme `pluslin`, qui ont une sémantique particulière définie dans le langage. Ainsi, les contraintes de Prolog IV sont définies comme des prédicats prédéfinis dans le système (il y en a plus de 120).

Les utilisateurs de Prolog III sont en droit de se plaindre du fait qu'un tel programme est beaucoup moins lisible que la version Prolog III du même programme. Nous sommes d'accord avec eux, et c'est pourquoi nous avons ajouté au langage une syntaxe alternative, qui nous permet de calculer la somme des éléments d'une liste de la manière suivante :

```
sumlist( [] , 0 ).
  sumlist( [X|L] , X + S ) :-
    sumlist( L , S ).
```

La différence majeure avec le programme précédent réside dans le traitement de l'addition, qui ici ressemble fort à l'addition de Prolog III. En fait, l'argument `X + S` est un *pseudo-terme* Prolog IV ; pour arriver à cette écriture élégante (au sens Prolog III), il y a en fait une version intermédiaire qui est la suivante :

```
sumlist( [] , 0 ).
  sumlist( [X|L] , pluslin(X,S) ) :-
    sumlist( L , S ).
```

Dans cette version, nous voyons que l'addition est écrite de manière fonctionnelle, mais en conservant l'identificateur initial. En mode Prolog IV, cet identificateur (`pluslin`) est interprété de manière spéciale quand il apparaît en tant que foncteur à deux arguments : une transformation de programme est appliquée automatiquement, dans laquelle l'expression `pluslin(X,S)` est remplacée par une nouvelle variable[56] `New`, et le littéral `pluslin(New,X,S)` est ajouté en début de la queue de règle, ce qui nous donne le programme initial.

Ces pesudo-termes peuvent être utilisés pour toutes les contraintes, ce qui permet de simplifier de nombreuses expressions. Enfin, le fait de remplacer le

56. Soit une variable qui n'apparaît pas dans la règle courante.

foncteur `pluslin` par l'opérateur + n'est que du "sucre syntaxique" permettant de rendre l'expression plus simple à lire. De nombreux exemples d'utilisation des pseudo-termes apparaissent tout au long du présent manuel.

Il existe donc deux modes distincts d'interprétation des programmes, un qui est compatible avec la syntaxe Prolog définie dans la norme ISO, et un autre qui autorise l'utilisation de pseudo-termes, facilitant l'écriture de programmes avec contraintes. Il existe donc des moyens de passer d'un mode à l'autre. Sur la ligne de commande, pour passer en mode ISO, il faut taper :

```
>> iso.

   true.

  ?-
```

On note ici que l'invite de commande change de `>>` en mode Prolog IV à `?-` en mode ISO. Pour revenir en mode Prolog IV, il faut taper :

```
?- prolog4.

   true.

  >>
```

Pour conclure, nous pouvons résumer les points essentiels de cette section :

- En Prolog IV, les contraintes sont ajoutées par le biais de prédicats prédéfinis.
- En plus du mode ISO, il existe un mode spécifique Prolog IV, qui permet d'utiliser des pseudo-termes, et donc d'écrire les contraintes de manière plus élégante.

1.4.2 Prolog IV hérite de Prolog III

Une deuxième affirmation importante à propos de Prolog IV est :

"Prolog IV hérite de Prolog III"

De nombreux programmes Prolog III peuvent être portés en Prolog IV en effectuant un effort minimal, souvent d'ordre syntaxique. Par exemple, un solveur linéaire est inclus dans Prolog IV, qui peut manipuler les mêmes contraintes que le solveur linéaire de Prolog III. La requête suivante permet donc de résoudre le système d'équations linéaires $\{X + Y = 3, X - Y = 1\}$:

```
>> X + Y = 3 , X - Y = 1 .

   Y = 1,
   X = 2.

  >>
```

Il existe bien d'autres caractéristiques de Prolog III qui existent toujours en Prolog IV. Par exemple, les prédicats de gestion des contraintes linéaires, comme `enum` et `lower_bound` existent toujours. A l'inverse, certaines caractéristiques ont dû être modifiées, généralement pour assurer la compatibilité avec la norme ISO. Par exemple, les opérateurs $>$ et $<$ sont prédéfinis dans la norme ISO, comme étant des comparaisons de nombres après évaluation des arguments par la primitive `is` ; on peut donc avoir l'échange suivant :

```
>> X > 1 .

   error: error(instantiation_error,(is)/2)
  >>
```

A la place de $>$, il faut donc utiliser la contrainte `gtlin`, et les autres contraintes de la famille (`gelin`, `ltlin` et `lelin`). L'échange normal serait donc le suivant :

```
>> gtlin(X,1).

   X ~ real.

  >>
```

On note ici une autre différence entre Prolog III et Prolog IV, qui est que Prolog IV n'affiche en sortie que les sous-domaines associés aux variables, et pas les contraintes du système [57].

Le solveur linéaire n'est pas le seul solveur de Prolog III que l'on retrouve en Prolog IV. Une opération de concaténation de listes est également disponible en Prolog IV, et elle permet d'écrire un programme élégant pour renverser le contenu d'une liste de la manière suivante :

```
reverse([] , []).
  reverse([X] o L1 , L2 o [X]) :-
    reverse(L1,L2).
```

L'opération de concaténation est maintenant noté o (o minuscule), puisque le point (.) est réservé dans la norme ISO. Le programme ci-dessus se comporte exactement comme le programme Prolog III équivalent tant que la longueur d'un de ses arguments est connue. Par contre, il peut avoir un comportement différent dans les autres cas.

En fait, comme dans tout héritage, certaines des fonctions héritées ont été modifiées, dégradées, voire perdues. Le solveur de listes en est un parfait exemple. Il existe toujours des contraintes permettant de contraindre la longueur d'une liste ou de concaténer deux listes, mais leur sémantique a été modifiée. Ici, le but était d'unifier les listes Prolog III avec les listes classiques de la norme ISO, et de rendre les opérations sur les listes symétriques. Cela nous a amené à diminuer de manière significative la puissance du raisonnement sur les listes [58]. Veuillez consulter les exemples ou le reste du présent manuel pour de plus amples informations sur les contraintes sur les listes.

Enfin, certains changements sont encore plus importants. Le plus important concerne le solveur booléen, qui n'existe plus en tant que tel dans Prolog IV. En effet, il perd son aspect complet et formel en étant plongé dans le solveur sur les réels, et gagne en puissance par les possibilités de mélange de contraintes numériques, ce qui n'était pas possible en Prolog III.

Il demeure donc possible de traiter des contraintes booléennes, un peu affaiblies, d'un côté et largement ouvertes d'un autre, en utilisant les contraintes sur les réels présentées dans la section suivante.

57. Ces différences sont discutées en détail à la fin du présent chapitre.

58. Par exemple, en Prolog III, les équations liant les longueurs des listes apparaissant dans des concaténations étaient traitées par le solveur linéaire ; ce n'est plus le cas en Prolog IV.

Pour conclure cette section, nous pouvons retenir les faits suivants :

– Il existe toujours en Prolog IV un solveur linéaire et un solveur sur les listes, mais le solveur booléen (un peu affaibli) est immergé dans le solveur numérique.

– Dans les solveurs qui ont été repris de Prolog III, des adaptations ont été nécessaires, et le comportement de certains programmes peut avoir changé.

1.4.3 Prolog IV est nouveau

Une troisième affirmation à propos de Prolog IV, et probablement la plus importante, est :

"Prolog IV est nouveau"

Prolog IV a de nombreuses caractéristiques qui peuvent être utilisées pour résoudre une large gamme de problèmes que Prolog III ne pouvait pas résoudre. De plus, comme nous l'avons déjà vu brièvement, certaines des fonctionnalités de Prolog III ont été entièrement révisées, de manière à donner à nos utilisateurs un système plus efficace et plus simple à utiliser.

Le changement le plus visible est l'ajout d'un nouveau solveur, basé sur l'arithmétique des intervalles, qui permet de manipuler toutes sortes de contraintes portant sur des nombres réels, sur des entiers et sur des booléens. Ce solveur est fondamentalement différent du solveur rationnel (ou linéaire) hérité de Prolog III, pour de nombreuses raisons :

– Ce solveur peut manipuler toutes sortes de contraintes non-linéaires, de la simple multiplication aux fonctions transcendentales comme `cos`.

– Des contraintes sur les réels, les entiers et les booléens peuvent être librement mélangées.

– Ce solveur est basé sur des *approximations*, ce qui implique que le solveur n'est pas complet[59] ; ses résultats doivent donc être analysés avec de plus grandes précautions que ceux du solveur linéaire.

Commençons par introduire de manière très informelle la façon de fonctionner de ce solveur. Il est destiné à raisonner sur les nombres réels, mais ces nombres ne peuvent pas être manipulés directement ; le solveur effectue donc ses calculs sur des intervalles de nombres réels qui ont la propriété suivante : leurs bornes doivent être représentables exactement par des nombres flottants[60]. Par exemple, Prolog IV contient un prédicat prédéfini `sqrt`, qui peut être utilisé pour calculer la racine carrée d'un nombre. La requête suivante a donc le résultat attendu :

```
>> X = sqrt(4).

   X = 2.

>>
```

A l'opposé, la requête suivante a des résultats plus surprenants :

59. Un solveur est complet si pour tout système de contraintes, il est capable de déterminer si le système est soluble ou non.

60. On utilise ici les nombres flottants simples définis dans la norme IEEE 754.

```
>> X = sqrt(2).

   X ~ cc(`>1.4142135`,`>1.4142136`).

  >>
```

Ce résultat signifie que le système n'est pas capable de calculer un résultat exact, mais qu'il est par contre capable de déterminer que la solution est dans l'intervalle :

$$[1.41421353816986083984375, 1.41421365737915039062 5].$$

Les deux nombres ci-dessus sont les valeurs exactes (avec toutes les décimales) des nombres flottants simples qui encadrent $\sqrt{2}$. L'écriture renvoyée par Prolog IV est une écriture simplifiée mais non équivoque. L'objet `` `>1.4142135` `` désigne en fait le plus petit nombre flottant supérieur à 1.4142135 ce qui permet de désigner un nombre flottant précisément sans avoir à donner toutes ses décimales.

On voit donc que le résultat n'est pas précis. Si on complique un peu notre requête, nous pouvons avoir :

```
>> X = sqrt(2), X = sqrt(2.0000001).

   X ~ oo(`>1.4142135`,`>1.4142136`).

  >>
```

Ce résultat est bien plus surprenant. En effet, il est clair pour nous que les nombres $\sqrt{2}$ et $\sqrt{2.0000001}$ sont des nombres différents. Toutefois, la faible précision des nombres flottants simples ne permet pas de les différencier, puisqu'ils font partie du même intervalle. Cette requête est donc un exemple de l'incomplétude du langage, dans la mesure le système n'a pas pu détecter l'incohérence d'un système de contraintes. Bien entendu, ce problème disparait dès que la précision de calcul est suffisante :

```
>> X = sqrt(2), X = sqrt(2.000001).

   false.

  >>
```

Ici, nous avons simplement changé la constante, en lui enlevant un 0. Ce changement suffit au système pour se rendre compte de l'incohérence du système.

Comme indiqué précédemment, le nouveau solveur permet également de manipuler des contraintes sur les entiers, sans recourir systématiquement à l'énumération comme le faisait Prolog III. En fait, une seule contrainte nous sert pour cela, la contrainte `int`. Considérons par exemple la requête suivante :

```
>> X ~ cc(1.5,2), Y ~ cc(2.25,2.75), Z = X .+. Y, int(Z), ge(Y,2.5).

 Z = 4,
 Y = 5/2,
 X = 3/2.

>>
```

Décrivons en détail la résolution de ce problème pour montrer comment fonctionne le solveur :

- La première contrainte indique que $X \in [1.5, 2]$.
- La deuxième contrainte indique que $Y \in [2.25, 2.75]$.
- Ensuite, en considérant la contrainte d'addition, on infère très simplement que $Z \in [3.75, 4.75]$. Cette inférence provient du fait qu'en additionnant n'importe quel nombre du domaine de X à un nombre du domaine de Y, nous devons obtenir un nombre du domaine de Z, et que le domaine $[3.75, 4.75]$ est le plus petit domaine vérifiant cette propriété.
- Ensuite, en considérant la contrainte `int`, le domaine de Z est réduit à $[4, 4]$ puisque 4 est le seul entier dans le domaine de départ de Z. En reconsidérant les contraintes accumulées et en particulier l'addition, le domaine de Y peut être réduit à $[2.25, 2.5]$ et celui de X à $[1.5, 1.75]$. Pour effectuer la réduction du domaine de Y, le raisonnement est le suivant : nous savons que $X + Y = 4$; Y est maximal quand X est minimal, soit quand $X = 1.5$. La valeur maximale de Y est donc 2.5. Le même raisonnement est appliqué pour chaque borne.
- La dernière étape consiste à considérer la contrainte $Y \geq 2.5$. Cette contrainte réduit le domaine de Y au singleton $[2.5, 2.5]$. Par propagation, le domaine de X se trouve également réduit et devient $[1.5, 1.5]$.
- Enfin, puisque les intervalles sont réduits à un singleton, ils sont traduits en nombres rationnels. Un mécanisme permet de déterminer les valeurs de certains arguments d'une contrainte lorsque suffisamment d'arguments prennent une valeur rationnelle.

Ce court exemple nous a permis de mettre en évidence les mécanismes de base de l'algorithme de résolution :

- Il est basé sur un algorithme de réduction d'intervalles, qui essaie continuellement de réduire les domaines de variables.
- Il est basé sur des calculs de point fixe, dans lequel les contraintes sont reconsidérées pour essayer de réduire les domaines jusqu'à avoir atteint la stabilité.
- Quand un intervalle se réduit à un singleton, une interface avec les autres solveurs permet de vérifier l'exactitude des calculs.

En fait, l'incomplétude de cet algorithme est quelque peu similaire à la situation qui se produisait en Prolog III avec les multiplications retardées. Quand une multiplication restait retardée (et donc pas prise en compte par le solveur numérique), il était impossible d'être sûr que le système courant était soluble. Par contre, quand la solution d'un système de contraintes contenant des multiplications était entièrement figée, nous étions alors certains que la multiplication était vérifiée. En utilisant les contraintes sur les réels de Prolog IV, il faut continuellement faire attention à la même chose. Considérons l'exemple classique où on cherche à résoudre le système insoluble $X \times X = -1$:

```
>> X .*. X = -1 .

   X ~ real.

  >>
```

On voit ici que l'incohérence n'est pas détectée. Pour la détecter, nous pouvons, comme en Prolog III, utiliser une énumération. Cela nous permet de mettre en évidence deux avantages importants du raisonnement sur les réels en utilisant des intervalles :

- Le nombre de "valeurs" possibles (c'est-à-dire d'intervalles de réels à bornes flottantes) est fini, ce qui permet de faire une énumération totale.
- Il est possible d'énumérer sur les nombres réels, c'est-à-dire de considérer tous les intervalles, même ceux qui n'ont pas de bornes entières.

Voyons tout de suite un exemple, en utilisant le prédicat `realsplit`, qui cherche des intervalles de réels :

```
>> X .*. X = -1 , realsplit([X]).

   false.

  >>
```

Ici, l'incohérence est détectée, dans la mesure où le système n'est pas capable de trouver une valeur de X qui vérifie l'équation, ce qui est le résultat attendu.

L'algorithme d'énumération, qui est en fait ici un algorithme de "découpage" des intervalles, est également beaucoup plus puissant que l'énumération de Prolog III, dans la mesure où il offre de nombreuses options permettant de contrôler la manière dont s'effectue le découpage, et permet de travailler simultanément sur plusieurs variables.

Nous n'irons pas plus loin dans cette section, et vous êtes invités à lire le reste du manuel pour de plus amples informations concernant les nouveaux solveurs de Prolog IV. Les informations les plus importantes de cette section sont :

- Un nouveau solveur approximé est disponible pour traiter les contraintes numériques ; il utilise l'arithmétique des intervalles et n'est pas complet.
- Ce solveur permet de mélanger des contraintes sur les nombres réels, les entiers et les booléens.
- Des utilitaires puissants sont mis à votre disposition pour les opérations d'énumération.

1.4.4 Prolog IV ne ressemble pas à Prolog III

La quatrième et dernière affirmation concernant Prolog IV est donc :

"Prolog IV ne ressemble pas à Prolog III"

Par cela, nous voulons dire que l'environnement de Prolog IV ne ressemble pas du tout à l'environnement de Prolog III. Voici les principaux changements :

- Prolog IV est un langage compilé, et non un langage interprété comme Prolog III.
- Prolog IV est livré avec un environnement complet, incluant un éditeur dédié, une console spécifique et un débogueur source interactif.

Nous n'allons pas dans la présente section discuter de l'environnement graphique, mais nous allons insister sur les quelques commandes qui vous permettent de contrôler le compilateur, et donner quelques informations concernant la console.

Ecrivons par exemple un programme `animaux` qui établit un lien entre un nombre de chats, un nombre d'oiseaux, un nombre de pattes et un nombre de têtes. Nous devons faire l'échange suivant :

```
>> consult.

   Consulting ...
   animaux(Chats,Oiseaux,Pattes,Tetes) :-
     Pattes = 4 .*. Chats .+. 2 .*. Oiseaux,
     Tetes = Chats + Oiseaux ,
     int(Chats), int(Oiseaux),
     ge(Chats,0), ge(Oiseaux,0),
     ge(Pattes,0), ge(Tetes,0),
     intsplit([Chats,Oiseaux]).
   end_of_file.
   true.

  >>
```

Le prédicat `consult` est utilisé pour rentrer le programme. Nous avons ici utilisé sa forme interactive (sans argument). Si un argument est fourni, ce doit être un atome qui contient le nom d'un fichier contenant le programme source. ATTENTION, le code inséré par `consult` n'est pas compilé. Pour obtenir du code compilé, il faut placer le code dans un fichier, et utiliser la commande `compile`, qui prend nécessairement un argument (le nom du fichier).

Le programme `animaux` se contente ici de poser les contraintes liant les variables, d'indiquer que nous voulons toujours avoir un nombre entier de chats et d'oiseaux, et que toutes les variables sont positives, avant de faire une énumération[61]. Pour finir la saisie, il faut taper la séquence "`end_of_file.`"[62] . Il n'est pas permis comme en Prolog III de terminer la saisie en entrant un point sur une ligne isolée. Nous pouvons maintenant tester notre programme :

61. Vous pouvez noter ici l'utilisation de `intsplit` qui effectue un découpage en considérant que les variables passées en arguments sont des variables entières.

62. On peut taper Ctrl-D si on est dans un terminal unix, hors de la console Prolog IV.

```
>> Pattes = 10, animaux(Chats,Oiseaux,Pattes,Tetes).

   Tetes = 5,
   Oiseaux = 5,
   Chats = 0,
   Pattes = 10;

   Tetes = 4,
   Oiseaux = 3,
   Chats = 1,
   Pattes = 10;

   Tetes = 3,
   Oiseaux = 1,
   Chats = 2,
   Pattes = 10.

  >>
```

La requête demande toutes les solutions pour lesquelles le nombre de pattes est 10. On obtient donc trois solutions. On note ici une différence avec Prolog III au niveau de l'interface utilisateur. En Prolog III, chacune des solutions était exprimée sous la forme d'un système de contraintes qui pouvait en retour être donné en entrée à Prolog III. Ici, c'est la solution entière du système qui est exprimée sous la forme d'une requête. Les solutions sont donc séparées par des points-virgules, et la dernière est terminée par un point. En recopiant cette solution sur la console, on obtient de nouveau la même solution, à une permutation des variables près.

Prolog IV contient également un mécanisme de *quantification existentielle* qui permet de simplifier des résultats. Nous pouvons par exemple écrire la même requête que précédemment, en stipulant qu'on ne souhaite connaitre les résultats que par rapport à `Chats` et `Oiseaux` :

```
>> Pattes ex Tetes ex Pattes = 10, animaux(Chats,Oiseaux,Pattes,Tetes).

 Oiseaux = 5,
 Chats = 0;

 Oiseaux = 3,
 Chats = 1;

 Oiseaux = 1,
 Chats = 2.

>>
```

L'opérateur `ex` permet donc d'indiquer que son opérande de droite doit être évalué sans chercher une valeur particulière pour son opérande de gauche (qui doit être une variable). Ce mécanisme permet donc d'éliminer des variables de la sortie, mais également d'utiliser des variables "locales" dans des littéraux Prolog.

Enfin au niveau de l'interaction, il est à noter que Prolog IV n'affiche pas comme résultat le système de contraintes résultant, mais seulement les domaines des variables non existentielles apparaissant dans la requête. Ces domaines sont des sous-ensembles de l'ensemble des arbres, dont les types prin-

cipaux apparaissent dans l'exemple ci-dessous :

```
>> L = [X,Y,Z] o C, Y = f(Y), cc(X,1,pi), int(Z).

   Y = f(Y),
   C ~ list,
   Z ~ real,
   L ~ list,
   X ~ co(1,'>3.1415927').

  >>
```

Ici, les variables `L` et `C` sont typées comme des listes, puisqu'elles apparaissent dans une concaténation. La variable *Z* est typée comme étant numérique ; cela paraît plus faible que ce qu'implique la contrainte `int`, mais `int` n'est qu'une contrainte, et non un domaine, et n'apparaît donc pas dans les sorties. Le domaine de la variable `X` est correct (la valeur de π est bien entendu approximée). Pour `Y`, nous avons l'équation définissant un arbre infini, non pas parce que c'est une contrainte importante, mais parce que c'est une construction, et que de plus cet arbre est unique, et la valeur de `Y` est donc entièrement connue. Ces notations demandent un peu d'habitude, mais elles deviennent assez rapidement très informatives sur le déroulement du programme.

Pour terminer ce chapitre destiné aux utilisateurs de Prolog III, voici un potpourri de requêtes dont le résultats devrait surprendre, émerveiller ou inquiéter de nombreux utilisateurs Prolog III :

– Une contrainte vraiment pas linéaire :

```
>> X = cos(X) .

   X ~ oo('>0.739085','>0.7390852').

  >>
```

– Les contraintes sont traitées de manière approximée :

```
>> cc(X,0,pi), cos(X) = sin(X).

   X ~ cc(0,'<1.570797').

  >>
```

– Mais un petit "split" règle les problèmes :

```
>> cc(X,0,pi), cos(X) = sin(X), realsplit([X]).

   X ~ oc('>0.7853977','<0.7853985').

  >>
```

– Les variables existentielles servent à simplifier le résultat :

```
>> X ex Y ex cc(X,0,pi), Y = cos(X), Y = sin(X),
      Z = square(Y) .*. 2, int(Z), realsplit([X]).

   Z = 1.

  >>
```

2

Les Bases de Prolog IV

2.1 Introduction

LE PREMIER PRINCIPE d'un langage tel que Prolog IV est d'être bâti sur une structure mathématique expressive dans laquelle on résout des contraintes complexes portant sur des éléments inconnus.

Le deuxième principe est d'avoir la liberté d'enrichir cette structure de nouvelles relations et de continuer à résoudre des contraintes dans la nouvelle structure obtenue.

Ces principes ne pouvant être respectés qu'à des degrés divers, un troisième principe de ce langage est d'être suffisamment bien conçu pour que ses insuffisances soient clairement perçues et donc mieux maîtrisées par l'utilisateur.

Explicitons ces trois principes.

Premier principe : une structure de base expressive

A la base de notre langage se trouve la notion de contrainte et pour entrer dans le vif du sujet en voici une, peu commune, formulée avec des notations empruntées à la logique du premier ordre et plus généralement aux mathématiques :

$$\exists u \exists v \exists w \exists x \left(\begin{array}{l} \quad y \leq 5 \\ \wedge \;\; v_1 = \cos v_4 \\ \wedge \;\; \text{size}(u) = 3 \\ \wedge \;\; \text{size}(v) = 10 \\ \wedge \;\; u \bullet v = v \bullet w \\ \wedge \;\; y \geq 2 + (3 \times x) \\ \wedge \;\; x = (74 > \lfloor 100 \times v_1 \rfloor) \end{array} \right) \tag{2.1}$$

La seule variable qui a des occurrences libres dans cette formule est y et donc cette contrainte exprime une propriété de sa valeur. Trouver cette valeur, c'est le rôle de Prolog IV.

Cet y doit donc être plus grand ou égal 5 et de plus il faut qu'il existe des vecteurs u, v, w et un réel x tels que les 1er et 4ème éléments du vecteur v soient égaux, les longueurs de u et v soient 3 et 10 le vecteur obtenu par concaténation de u avec v soit le même que celui obtenu par concaténation de v avec w, le réel y soit plus grand ou égal à $2+3x$ avec x contraint à être égal

à 1 ou 0 suivant que 74 est ou n'est pas plus grand à $\lfloor 100 \times v_1 \rfloor$, c'est-à-dire la partie entière de 100 fois le premier élément du vecteur v. Ouf ! Il n'y a qu'un seul y qui satisfait à toute cette contrainte et c'est

$y = 5.$

Comment se présentent les choses en Prolog IV ? On lance le système qui signale qu'il attend une requête en affichant un double chevron fermant, on lui fournit la contrainte précédente dans une syntaxe adéquate, il la résout, fournit la réponse et affiche à nouveau le double chevron pour signaler qu'il est prêt à recommencer. Cela donne :

```
>> U ex V ex W ex X ex
   le(Y,5),
   V:1 = cos(V:4),
   size(U) = 3,
   size(V) = 10,
   U o V = V o W,
   ge(Y,2.+.(3.*.X)),
   X = bgt(74,floor(100.*.V:1)).

Y = 5.
```

Rapidement quelques mots sur la syntaxe utilisée. Prolog IV se voulant conforme syntaxiquement à la norme Prolog ISO [1], toute expression se doit d'être un «terme» au sens de cette norme [2]. Les termes au sens de la norme n'étant pas les termes comme on les connaît en logique, nous appellerons les premiers, termes-ISO. Ces termes-ISO sont essentiellement de l'une des trois formes

1 variable,
2 nombre ou identificateur,
3 identificateur(terme-ISO,...,terme-ISO),

avec des variations infixées, des notations spécifiques aux listes et la convention que les noms de variables commencent par des majuscules. Ces variations permettent d'écrire ici

`X:I`	pour	`index(X,I),`
`X o Y`	pour	`conc(X,Y),`
`X.+.Y`	pour	`plus(X,Y),`
`X.*.Y`	pour	`times(X,Y),`

les relations `index/3`, `conc/3`, `plus/3`, `times/3` étant utilisées avec des notations fonctionnelles.

Passons à la sémantique de la contrainte. Le connecteur logique $\wedge$, noté ici par une virgule et la quantification existentielle $\exists xp$ notée ici x `ex` p ont leurs sens habituels, nous y reviendrons plus tard. Les variables représentent des élément inconnus pris dans un certain domaine et les autres symboles des relations et des opérations bien précises dans ce même domaine.

Ces opérations et relations dans un domaine précis font partie d'une structure plus complète, choisie une bonne fois pour toute, la structure de base π_4. Ainsi que le laisse pressentir notre exemple de contrainte, il s'agit d'une structure

1. *ISO/IEC 13211-1*, Information Technology, Programming Languages, Prolog, Part 1: General Core, 1995.

2. A la page 5 du livre de Pierre Deransart, AbdelAli Ed-Dbali et Laurent Cervoni, *Prolog: The Standard*, Springer-Verlag, 1996, on peut lire : «In Standard Prolog a unique data structure is used: terms»

particulièrement riche. Nous avons voulu à la fois pouvoir manipuler (1) des symboles, (2) des nombres et (3) des objets complexes construits à partir de ceux-ci. En ce qui concerne la partie numérique, nous nous sommes montré particulièrement ambitieux en introduisant une centaine de relations portant sur l'ensemble **R** des nombres réels (au sens mathématique), sur l'ensemble **Q** des nombres rationnels (les fractions), sur l'ensemble **Z** des nombres entiers relatifs et sur l'ensemble **B** des valeurs booléennes 0 et 1, et ceci, tout en respectant les inclusions $\mathbf{B} \subset \mathbf{Z} \subset \mathbf{Q} \subset \mathbf{R}$.

Deuxième principe : Une structure enrichissable

En écrivant un programme Prolog IV on définit de nouvelles relations qui complètent la structure de base π_4. Cet enrichissement est plus d'ordre quantitatif que d'ordre qualitatif : essentiellement les nouvelles relations permettent d'exprimer en une seule contrainte la conjonction d'une multitude de contraintes atomiques exprimables dans π_4. Par exemple le programme

$$\forall l \forall x \left(\text{sigma}(x,l) \equiv \left(\begin{array}{l} l = [\,] \wedge x = 1 \\ \qquad \vee \\ \exists l' \exists y' \exists x' \left(\begin{array}{l} l = .(y', l') \\ \wedge\ \text{plus}(y', x', x) \\ \wedge\ \text{sigma}(x', l') \end{array} \right) \end{array} \right) \right) \tag{2.2}$$

permet de définir le lien qui existe entre une liste et la somme de ses éléments par une conjonction de contraintes portant sur le lien qui existe entre deux nombres et leur somme. Des conventions syntaxiques permettent d'écrire le programme sous forme d'un paquet de clauses plus habituel dans le monde Prologien :

```
sigma(0,[]).
sigma(X,.(Yp,Lp)) :- plus(X,Yp,Xp), sigma(Xp,Lp).
```

Troisième principe : les insuffisances sont explicites

Prolog IV ne peut être parfait, il est hors de question de pouvoir résoudre parfaitement des contraintes dans une structure aussi riche que π_4 et il est encore moins question de laisser un usager enrichir π_4 sans aucune discipline et s'attendre à ce que miraculeusement la machine résolve toutes les contraintes exprimables dans cette nouvelle structure.

Ce que nous avons voulu c'est que Prolog IV soit parfait dans son imperfection ou plus exactement que ses imperfections soient explicites. Ceci passe par un fondement logique bien précis que l'on peut résumer en trois points.

(1) Syntaxiquement le langage Prolog IV est vu comme un langage du premier ordre avec égalité et les contraintes comme des formules positives, quantifiées existentiellement (partiellement).

(2) Sémantiquement et en l'absence de tout programme, le langage Prolog IV est vu comme une théorie $\mathcal{T}(\pi_4)$ du premier ordre avec égalité c'est-à-dire un ensemble d'axiomes exprimant des propriétés de la structure π_4. C'est uniquement ces propriétés qui sont utilisées pour résoudre des contraintes, mais ces propriétés sont utilisées au mieux. Lorsque le programmeur donne une contrainte p à résoudre à Prolog IV il s'attend à ce que le système lui retourne une contrainte q équivalente à p dans la structure π_4 c'est-à-dire ayant les mêmes solutions que p dans la structure π_4 mais dont les solutions sont plus

«visibles». Notamment il aimerait que Prolog IV retourne la contrainte *false* si p n'a pas de solutions dans π_4. En fait la contrainte q retournée sera non seulement équivalente à p dans la structure π_4 mais dans toutes les structures ϕ ayant les propriétés $\mathcal{T}(\pi_4)$, ce qui s'écrit

$$\mathcal{T}(\pi_4) \quad \models \quad p \equiv q$$

et ne sera *false* que si, non seulement elle n'a pas de solutions dans la structure π_4, mais si en plus elle n'a de solutions dans aucune des autres structures ϕ qui ont les propriétés $\mathcal{T}(\pi_4)$, c'est-à-dire si

$$\mathcal{T}(\pi_4) \quad \models \quad p \equiv \mathit{false}$$

(3) Sémantiquement et en présence d'un programme $\mathcal{P}$ le langage Prolog IV est vu comme la théorie $\mathcal{T}(\pi_4) \cup \mathcal{P}$. Le programme $\mathcal{P}$ est donc un ensemble d'axiomes supplémentaires que l'on ajoute à $\mathcal{T}(\pi_4)$. Chacun de ces axiomes définit la propriété essentielle d'une nouvelle relation nommée r et est de la forme

$$\forall x_1 \ldots \forall x_n \ (r\, x_1 \ldots x_n \equiv p).$$

où p est une formule n'ayant pas d'autres occurrences libres de variables que celles $x_1, \ldots, x_n$. Bien entendu tout est limpide si p ne fait que contenir des relations et des opération de π_4, mais dans le cas ou p fait intervenir d'autres nouvelles relations et surtout la relation r elle-même, on a à faire à des définitions récursives. La résolution d'une contrainte p doit alors être vue comme le calcul d'une contrainte équivalent à q dans la théorie $\mathcal{T}(\pi_4) \cup \mathcal{P}$ c'est-à-dire telle que

$$\mathcal{T}(\pi_4) \cup \mathcal{P} \quad \models \quad p \equiv q$$

Plan de l'exposé

Cette introduction constitue la partie 1 de notre exposé. La partie 2 est consacrée à des définitions purement logiques et notamment à ce que nous entendons par une contrainte, une structure, etc. C'est un petit cours de logique taillé sur mesure. La partie 3 décrit en détail la structure choisie π_4. La partie 4 est consacrée à l'axiomatisation de cette structure. La partie 5 décrit la notion de programme et la machinerie Prolog IV. Enfin la partie 6 donne des exemples divers de programmes Prolog IV.

2.2 Syntaxe et sémantique

2.2.1 Objets sémantiques

Les seuls objets sémantiques dont nous aurons besoin sont les relations et les opérations. Quelques définitions précises et quelques commentaires s'imposent ici. Soit D un ensemble et n un entier non négatif.

Une *relation* ρ est un sous-ensemble quelconque de D^n. On dit que ρ est une relation d'*arité* n *dans* D. Du fait qu'une relation est vue comme un ensemble de n-uplets, tous de même longueur, l'ensemble vide $\emptyset$ est une relation qui a pour arité n'importe quelle entier non négatif.

Une *opération* φ est une application de D^n dans D. On dit que φ est une opération d'*arité* n *dans* D. Il est commode de considérer les opérations comme des cas particuliers de relations ; pour ceci on confond l'opération n-aire φ avec la relation $n+1$-aire

$$\{(a_0, a_1, \ldots, a_n) \in D^{n+1} \mid a_0 = \varphi(a_1, \ldots, a_n)\}.$$

Une opération n-aire est donc une relation $n+1$-aire ρ qui a la propriété :

> quel que soit $(a_1, \ldots, a_n) \in D^n$,
> il existe un et un seul $a_0 \in D$ tel que $(a_0, a_1, \ldots, a_n) \in \rho$.

Une relation $n+1$-aire ρ qui a la propriété affaiblie :

> quel que soit $(a_1, \ldots, a_n) \in D^n$,
> il existe au plus un $a_0 \in D$ tel que $(a_0, a_1, \ldots, a_n) \in \rho$.

sera appelée *opération partielle* ou *pseudo-opération*.

2.2.2 Objets syntaxiques

Les seuls objets syntaxiques dont nous aurons besoin sont les termes et les formules. Les termes servent à représenter des éléments d'un domaine et les formules servent à représenter des propriétés de ces éléments.

On se donne une bonne fois pour toute un ensemble universel infini V de *symboles de variables*[3] pour représenter des individus pris dans un domaine. On se donne en plus des ensembles F, R de symboles d'opérations et de relations. A chacun de ces symboles est associé un entier positif ou nul, son *arité*.

Les *termes* sont des expressions de l'une des deux formes :

1	x	x
2	$f\, t_1 \ldots t_n$	$f\,(t_1\,, \ldots, t_n)$

avec $x \in V$, $f \in F$ et d'arité n et les t_i des termes plus courts que ceux que l'on est train de définir. Les notations sont celles qui sont le plus couramment utilisées. Dans la colonne de droite nous donnons un aperçu de la notation Prolog IV de chaque forme.

3. Rappelons que la grande trouvaille de la norme ISO, à laquelle nous nous conformons dans la syntaxe de Prolog IV, est de noter ces variables par des identificateurs commençant par une lettre majuscule !

Les *formules* sont des expressions de l'une des 11 formes

1	*true*	`true`
2	*false*	`false`
3	$t_1 \doteq t_2$,	t_1 `=` t_2
4	$r\,t_1 \ldots t_n$	r`(`t_1`,...,`t_n`)`
5	$(p \wedge q)$	`(`p `,` q`)`
6	$(p \vee q)$	`(`p `;` q`)`
7	$\exists x p$	x `ex` p
8	$\neg p$	
9	$(p \Rightarrow q)$	
10	$(p \equiv q)$	
11	$\forall x p$	

(2.3)

où les t_i sont des termes, r un symbole de R d'arité n et p, q des formules plus courtes que celles que l'on est train de définir. Dans la colonne de droite nous donnons toujours, pour chaque forme, un aperçu de la notation Prolog IV, quand elle existe. Une occurrence de variable x dans une formule q est dite *liée* si elle se produit à l'intérieur d'une formule de la forme 7 ou 11 figurant dans q. Une occurrence qui n'est pas liée est dite *libre*. Une formule qui ne contient aucune occurrence libre de variable est une *proposition*.

Nous réservons le mot *contrainte* pour désigner des formules existentielles positives, c'est-à-dire ne faisant intervenir que les 7 première formes. Une contrainte ou formule *atomique* est une formule de la forme 1, 2, 3 ou 4.

2.2.3 Liens entre objets syntaxiques et objets sémantiques

Si l'on donne une signification à chaque symbole qui figure dans un terme ou dans une formule on pourra systématiquement étendre ces significations pour attribuer une signification à tout le terme où toute la formule. C'est cette extension de signification que nous allons préciser. On appelle *interprétation* I toute application qui fait correspondre

- à chaque variable $x \in V$, un élément $I(x)$ d'un ensemble D,
- à chaque symbole d'opération $f \in F$ d'arité n, une opération n-aire $I(f)$ dans le même ensemble D,
- à chaque symbole de relation $r \in F$ d'arité n, une relation n-aire $I(r)$ dans le même ensemble D.

L'ensemble commun D est appelé domaine de I et sera noté dom(I), l'ensemble F des symboles d'opérations interprétés par I est noté op(I) et l'ensemble R des symboles de relations interprété est noté rel(I)

L'interprétation I est donc l'objet mathématique qui attribue une signification à chaque symbole de $V \cup F \cup R$. Cette interprétation ce prolonge en une application I^* qui attribue à chaque terme t construit sur $V \cup F$ un élément $I^*(t)$ de dom(I) et à chaque formule p construit sur $V \cup F \cup R$ une valeur de vérité $I^*(p)$ prise dans $\{$vrai, faux$\}$ comme suit.

L'application I^* est définie récursivement sur les deux formes de termes par :

$$\begin{array}{lll} 1 & I^*(x) & = & I(x) \\ 2 & I^*(f\,t_1 \ldots t_n) & = & I(f)(I^*(t_1), \ldots, I^*(t_n)) \end{array}$$

et sur les 11 formes de formules par :

1	$I^*(\mathit{true})$	$=$	vrai
2	$I^*(\mathit{false})$	$=$	faux
3	$I^*(t_1 \doteq t_2)$	$=$	$\begin{cases} \text{vrai} & \text{si } I^*(t_1) = I^*(t_2), \\ \text{faux} & \text{sinon} \end{cases}$
4	$I^*(r\, t_1 \ldots t_n)$	$=$	$\begin{cases} \text{vrai} & \text{si } (I^*(t_1), \ldots, I^*(t_n)) \in I(r) \\ \text{faux} & \text{sinon} \end{cases}$
5	$I^*(p \wedge q)$	$=$	$\begin{cases} \text{vrai} & \text{si } I^*(p) = \text{vrai et } I^*(q) = \text{vrai} \\ \text{faux} & \text{sinon} \end{cases}$
6	$I^*(p \vee q)$	$=$	$\begin{cases} \text{vrai} & \text{si } I^*(p) = \text{vrai ou } I^*(q) = \text{vrai} \\ \text{faux} & \text{sinon} \end{cases}$
7	$I^*(\exists x p)$	$=$	$\begin{cases} \text{vrai} & \text{s'il existe } J \in \text{next}(I, x) \text{ avec } J^*(p) = \text{vrai}, \\ \text{faux} & \text{sinon} \end{cases}$
8	$I^*(\neg p)$	$=$	$\begin{cases} \text{vrai} & \text{si } I^*(p) = \text{faux} \\ \text{faux} & \text{sinon} \end{cases}$
9	$I^*(p \Rightarrow q)$	$=$	$\begin{cases} \text{faux} & \text{si } I^*(p) = \text{vrai et } I^*(q) = \text{faux} \\ \text{vrai} & \text{sinon} \end{cases}$
10	$I^*(p \equiv q)$	$=$	$\begin{cases} \text{vrai} & \text{si } I^*(p) = I^*(q) \\ \text{faux} & \text{sinon} \end{cases}$
11	$I^*(\forall x p)$	$=$	$\begin{cases} \text{vrai} & \text{si pour tout } J \in \text{next}(I, x) \text{ on a } J^*(p) = \text{vrai}, \\ \text{faux} & \text{sinon} \end{cases}$

en convenant que next(I, x) est l'ensemble d'interprétations composé de l'interprétation I et de toutes les interprétations J qui ne diffèrent de I que par le fait que $I(x) \neq J(x)$.

Une remarque importante s'impose : la valeur $I^\star(p)$ de la formule p ne dépend pas des valeurs $I(x)$ des variables x sans occurrences libres dans p. Donc, si p est une proposition, c'est-à-dire une contrainte sans occurrences libres de variables, $I^\star(p)$ ne dépend pas de la façon d'interpréter les variables.

Si $I^\star(p) =$ vrai on dit que p *est vraie dans* l'interprétation I mais aussi que I *satisfait* p. Si $\mathcal{T}$ est un ensemble de formules et p une formule on dira que p *est vrai dans* $\mathcal{T}$ et on écrira

$$\mathcal{T} \models p$$

si toute interprétation qui satisfait chaque formule de $\mathcal{T}$ satisfait aussi p.

Dans beaucoup de raisonnements on est amené à faire varier l'interprétation des symboles de variables tout en gardant la même interprétation pour les autres symboles. Il est alors intéressant de décomposer l'interprétation I en un couple (ϕ, σ) d'applications définies comme suit.

$$I(s) = \begin{cases} \phi(s), & \text{si } s \in F \cup R, \\ \sigma(s), & \text{si } s \in V. \end{cases}$$

L'application σ est une *affectation de variable* c'est-à-dire une application de V dans un ensemble que l'on notera dom(V). L'application ϕ est une *structure* c'est-à-dire une famille d'opérations $\phi(f)$ et de relations $\phi(r)$ sur un domaine dom(ϕ) indicés par des symboles d'opérations f pris dans un ensemble op(ϕ) et par des symboles de relations pris dans un ensemble de symboles rel(ϕ).

Lorsque l'interprétation I satisfait la formule p et que I est décomposé dans le couple (ϕ, σ) on peut utiliser un jargon plus mathématique que logique et dire que l'affectation σ *est solution* de la formule p *dans* la structure ϕ.

2.2.4 Notation fonctionnelle de relations

Afin d'utiliser des notations fonctionnelles pour représenter des individus qui sont dans certaines relations, on assimile la relation $(n+1)$-aire ρ sur l'ensemble D à l'application suivante de D^n dans l'ensemble $\mathcal{P}(D)$ des parties de D

$$(a_1, \ldots, a_n) \mapsto \{a_0 \,|\, (a_0, a_1, \ldots, a_n) \in \rho\}.$$

On construit alors des expressions qui, contrairement aux termes, ne représentent pas des éléments de D mais des ensembles d'éléments de D. Ces nouvelles expressions que nous appellerons *pseudo-termes* sont de l'une des deux formes :

1	$r\, s_1 \ldots s_n$	$r\,(s_1\,,\ldots,s_n)$
2	$f\, s_1 \ldots s_n$	$f\,(s_1\,,\ldots,s_n)$

avec $x \in V$, $f \in F$ et d'arité n, $r \in R$ et d'arité $n+1$, les s_i des termes ou pseudo-termes plus courts que ceux que l'on est train de définir et l'expression de la forme 2 contenant au moins un pseudo-terme. Ici aussi la colonne de droite donne un aperçu des notations Prolog IV. On donne alors une définition généralisée des formules logiques en complétant leurs 11 formes, de la page 66, par les deux formes

3bis	$s_1 \sim s_2$	$s_1 \sim s_2$
4bis	$r\, s_1 \ldots s_n$	$r\,(s_1\,,\ldots,s_n)$

où les s_i sont des termes ou des pseudo-termes et où r appartient à R et est d'arité n. Bien entendu les formules atomiques sont maintenant les formules de la forme 1, 2, 3, 3bis, 4 ou 4bis et les notions d'occurrences liées ou libres de variables restent inchangées.

Le signe $\sim$ se lit *est compatible avec* et, si a et b sont des éléments ou des sous-ensembles de D, a la signification suivante :

$$a \sim b \stackrel{def}{\equiv} \begin{cases} a = b, & \text{si } a \text{ et } b \text{ sont des éléments de } D, \\ a \in b & \text{si } a \text{ est un élément et } b \text{ un sous-ensemble de } D, \\ a \ni b & \text{si } a \text{ est un sous-ensemble et } b \text{ un élément de } D, \\ a \cap b \neq \emptyset & \text{si } a \text{ et } b \text{ sont des sous-ensembles de } D. \end{cases}$$

La contrainte atomique $r\, s_1 \ldots s_n$ exprime qu'il existe des individus $a_1, \ldots, a_n$ qui sont dans la relation r et qui sont compatibles avec $s_1, \ldots, s_n$.

Plus précisément si on se donne une interprétation I, la valeur de vérité $I^\star(p)$ d'une formule généralisée de la forme 3bis ou 4bis est définie par

$$3bis \quad I^*(s_1 \sim s_2) = \begin{cases} \mathit{true}, & \text{si } I^*(s_1) \sim I^*(s_2) \\ \mathit{false}, & \text{sinon} \end{cases}$$

$$4bis \quad I^*(r\, s_1 \ldots s_n) = \begin{cases} \mathit{true} & \text{s'il existe des } a_i \text{ avec} \\ & a_1 \sim I^*(s_1), \ldots, a_n \sim I^*(s_n) \\ & \text{et } (a_1, \ldots, a_n) \in I(r) \\ false, & \text{sinon} \end{cases}$$

où la valeur $I^\star(s)$ d'un pseudo-terme est un ensemble défini sur les deux formes du pseudo-terme par

$$
\begin{array}{lrcl}
1 & I^*(r\,s_1 \ldots s_n) & = & \left\{ \begin{array}{l} \text{l'ensemble des } b \text{ tels qu'il existe des } a_i \text{ avec} \\ a_1 \sim I^*(s_1), \ldots, a_n \sim I^*(s_n) \text{ et } (b, a_1, \ldots, a_n) \in I(r) \end{array} \right. \\
2 & I^*(f\,s_1 \ldots s_n) & = & \left\{ \begin{array}{l} \text{l'ensemble des } b \text{ tels qu'il existe des } a_i \text{ avec} \\ a_1 \sim I^*(s_1), \ldots, a_n \sim I^*(s_n) \text{ et } b = I(f)(a_1, \ldots, a_n). \end{array} \right.
\end{array}
$$

Toutes les autres notions liées aux interprétations restent inchangées.

Une formule généralisée peut toujours être transformée en une formule équivalente ne faisant intervenir ni pseudo-terme ni le signe $\sim$. Il suffit de répéter les transformations suivantes :

$$
\begin{array}{llll}
1 & t_1 \sim t_2 & \text{devient} & t_1 = t_2 \\
2 & p(r\,s_1 \ldots s_n) & \text{devient} & \exists x \;\; r\,x s_1 \ldots s_n \wedge p(x)
\end{array}
\tag{2.4}
$$

où t_1 et t_2 sont des termes, où $p(r\,s_1 \ldots s_n)$ est une formule atomique dans laquelle on a choisi une occurrence d'un pseudo-terme $r\,s_1 \ldots s_n$ de la forme 1 et où $p(x)$ est la même formule atomique dans laquelle on a substitué une nouvelle variable x à l'occurrence choisie du pseudo-terme.

La transformation 2 permet d'éliminer tous les symboles de relations imbriqués dans des contraintes atomiques et donc d'éliminer tous les pseudo-termes. La transformation 1 permet ensuite d'éliminer tous les signes $\sim$.

Si on applique ces transformations sur la contrainte

```
X ~ [cc(1,2) u co(4,8), gt(9), real]
```

du tableau 2.3 à la page 73 on obtient, en tenant compte de ce que `cc(1,2) u co(4,8)` n'est que la variante infixée de `u(cc(1,2),co(4,8))`,

```
A ex B ex C ex D ex E ex
X = [A,D,E], u(A,B,C), cc(B,1,2), co(C,4,8),
gt(D,9), real(E)
```

Terminons cette partie par une remarque d'ordre syntaxique. L'introduction du signe $\sim$ n'était pas absolument nécessaire, on aurait très bien pu utiliser le signe «égal» en étendant son sens à celui de «compatible». Il aurait cependant été choquant de lier par l'égalité des ensembles pouvant être distincts. Cependant, dans une contrainte de la forme $x \sim s$, où s est un pseudo-terme représentant un ensemble de un ou de zéro élément, le signe égal est plus esthétique. Pour toutes ces raisons, le compilateur Prolog IV acceptera que l'on remplace en entrée toute occurrence du signe compatible par celle du signe égal. C'est aussi ce que nous avions fait dans notre tout premier exemple de contrainte à la page 61.

2.3 La structure de base π_4

2.3.1 Domaine de la structure choisie π_4

Le domaine $\mathrm{dom}(\pi_4)$ de la structure choisie est l'ensemble des arbres dont les nœuds, qui forment un ensemble éventuellement infini, sont étiquetés

1. soit par des identificateurs Prolog-ISO (complétés d'une arité),
2. soit par des nombres réels (considérés comme des étiquettes d'arité nulle),

On ne distingue pas les arbres ayant un seul nœud de l'étiquette portée par ce nœud. Les identificateurs et les réels sont donc assimilés à des arbres d'un nœud étiquetés par des identificateurs et par de réels. Plusieurs sous-ensembles de notre domaine jouent un rôle particulier, passons les en revue.

Ensemble des nombres rationnels Ce sont les nombres réels de la forme $\pm\frac{p}{q}$ où q est un entier positif et p un entier positif ou nul. On les écrira tout simplement sous la forme `p/q` avec éventuellement un signe devant.

Ensemble des nombres rationnels décimaux Ce sont des nombres rationnels de la forme $\pm p \times 10^{\pm q}$, où p et q sont des entiers positifs ou nuls. On les note d'une façon classique avec éventuellement un point décimal et une partie exposant introduite par la lettre `E`. Remarquons que les nombres entiers sont un cas particulier des nombres décimaux.

Ensemble des nombres rationnels IEEE Ce sont des nombres binaires, à virgule flottante, conformes à la norme IEEE simple précision[4]. Ils sont de la forme

$$\pm p \times 2^q \quad \text{avec} \quad 0 \leq p \leq 2^{24}-1 \text{ et } -149 \leq q \leq 104,$$

où p et q sont des entiers. Il y en a $2n+1$ avec $n = 2^{31}-2^{23}-1$, c'est-à-dire en tout et pour tout $2\,139\,095\,039$. Si on les numérote de $-n$ à n et que l'on désigne par q_i le nombre numéro i alors l'entier $|i|$ écrit en binaire sur 31 bits et précédé d'un bit pour indiquer le signe de i constitue le codage IEEE de q_i. Ce codage est sans redondances sauf le nombre 0 qui est codé avec le signe plus ou le signe moins.

Les rationnels IEEE sont des cas particuliers des rationnels décimaux[5] et peuvent donc être notés sous forme décimale avec beaucoup de chiffres. Cependant afin de disposer d'une notation plus compacte on écrit `'<`x`'` et `'>`x`'` pour désigner les rationnels IEEE qui précèdent et suivent immédiatement le nombre décimal sans signe x.

4. IEEE Standard for Binary Floating-Point Arithmetic, IEEE Std 754-1985, Approved March 21, 1985, Reaffirmed December 6,1990, IEEE Standards Board, approved July 26, 1985 American National Standards Institute

5. En effet si q est non négatif ce sont des entiers et donc des rationnels décimaux et si q est négatif ils sont de la forme $\pm(5^{|q|} \times p) \times 10^q$.

Intervalle C'est un ensemble de nombre réels de l'une des 10 formes :

1	$(-\infty,+\infty)$:	$\mathbf{R}$	`real`
2	$[a,+\infty)$:	$\{x \in \mathbf{R} \mid a \leq x\}$,	`ge(`a`)`
3	$(a,+\infty)$:	$\{x \in \mathbf{R} \mid a < x\}$,	`gt(`a`)`
4	$(-\infty,b]$:	$\{x \in \mathbf{R} \mid x \leq b\}$,	`le(`b`)`
5	$(-\infty,b)$:	$\{x \in \mathbf{R} \mid x < b\}$,	`lt(`b`)`
6	$[a,b]$:	$\{x \in \mathbf{R} \mid a \leq x \leq b\}$,	`cc(`a`,`b`)`
7	$(a,b]$:	$\{x \in \mathbf{R} \mid a < x \leq b\}$,	`oc(`a`,`b`)`
8	$[a,b)$:	$\{x \in \mathbf{R} \mid a \leq x < b\}$,	`co(`a`,`b`)`
9	(a,b) :	$\{x \in \mathbf{R} \mid a < x < b\}$	`oo(`a`,`b`)`
10	$\emptyset$:	l'ensemble vide.	`ntree`

où a et b, les bornes inférieure et supérieure, sont des nombres réels et où la dernière colonne donne les notations Prolog IV.

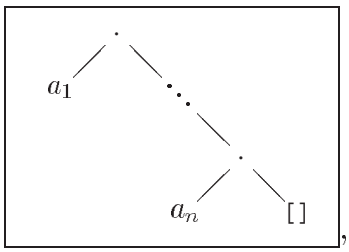
Intervalle IEEE C'est un intervalle dont les bornes inférieure a et supérieure b, si elles existent, sont des nombres rationnels IEEE. On remarquera que l'ensemble des intervalles IEEE a pour éléments l'ensemble vide et l'ensemble des réels et est fermé pour toute intersection finie ou infinie de ses éléments. Le tout dernier point découle du fait que l'ensemble des nombres rationnels IEEE est fini.

Union d'intervalles IEEE C'est un ensemble de réels de la forma $E_1 \cup \cdots \cup E_n$, où les E_i sont des intervalles IEEE.

Ensemble des arbres rationnels Ce sont les arbres qui ont un ensemble fini de sous-arbres. Rappelons qu'il existe des arbres ayant un ensemble infini de nœuds mais un nombre fini de sous-arbres (non isomorphes).

Ensemble des arbres doublement rationnels Ce sont les arbres rationnels qui ne font pas intervenir d'autres nombres que des nombres rationnels.

Ensemble des listes Ce sont des arbres a de la forme



où les a_i sont des arbres quelconques. Une telle liste a est notée $[a_1, \ldots, a_n]$ et on désigne par $|a|$ sa *taille* éventuellement nulle n. La concaténation de deux listes, de tailles éventuellement nulles, est définie et notée par $[a_1, \ldots, a_m] \circ [a_{m+1}, \ldots, a_n] = [a_1, \ldots, a_n]$.

2.3.2 Sous-domaines privilégiés

Tout le mécanisme de résolution approché de contraintes de Prolog IV repose sur le fait qu'à chaque variable est attribué un sous-ensemble de l'ensemble

dom(π_4) d'arbres et que constamment on essaye

1. de «réduire» ces sous-ensembles en tenant compte de leurs interactions avec chaque contrainte prise séparément et
2. de «globaliser» toute contrainte locale c'est-à-dire de la remplacer par une contrainte plus basique que l'on sait résoudre globalement.

Ces sous-ensembles ne sont pas quelconques, ce sont les *sous-domaines privilégiés*. Ils sont obtenus par intersections multiples des 15 formes de bases du tableau 2.1 qui représentent des sous-ensembles pas toujours disjoints. Une

TAB. 2.1 – *Sous-domaines privilégiés*

	Formes de base
1	l'ensemble des arbres
2	l'ensemble des arbres finis
3	l'ensemble des arbres infinis
4	l'ensemble des arbres qui sont des listes
5	l'ensemble des arbres qui ne sont pas des listes
6	pour chaque entier n positif ou nul, l'ensemble des arbres qui sont des listes de taille n
7	pour chaque n-uplet $A_1, \ldots, A_n$ d'intervalles IEEE, l'ensemble des listes de la forme $[a_1, \ldots, a_n]$, avec $a_i \in A_i$
8	l'ensemble des arbres ayant un seul nœud
9	l'ensemble des arbres ayant plus d'un seul nœud
10	l'ensemble des identificateurs
11	l'ensemble des arbres qui ne sont pas un simple identificateur
12	pour chaque intervalle IEEE A, l'ensemble A
13	l'ensemble des arbres qui ne sont pas un simple nombre réel
14	pour chaque arbre a doublement rationnel, l'ensemble $\{a\}$
15	l'ensemble vide

variante possible pour les formes 7 et 12 est donnée dans le tableau 2.2. Les modalités pour utiliser cette variante sont décrites dans la partie 9 de ce document : environnement général.

TAB. 2.2 – *Variante des sous-domaines privilégiés*

	Formes de base
7	pour chaque n-uplet $A_1, \ldots, A_n$ d'unions A_i d'intervalles IEEE, l'ensemble des listes de la forme $[a_1, \ldots, a_n]$
12	pour chaque union A d'intervalle IEEE, l'ensemble A

Chaque sous-domaine privilégié est de catégorie (a), de catégorie (b) ou à la fois de catégorie (a) et (b). Les sous-domaines de catégorie (a) interviennent dans le processus de réduction des sous-domaines et ceux de catégorie (b) dans le processus de globalisation des contraintes.

– Sont de catégorie (a) tous les sous-domaines privilégiés, à l'exception de ceux qui sont de la forme 14, lorsque a est un nombre, qui n'est pas un rationnel IEEE.

– Sont de catégorie (b) tous les sous-domaines privilégiés, à l'exception de ceux qui sont de la forme 7 ou 12, lorsque un A_i ou A est autre que

l'ensemble vide ou l'ensemble **R** de tous les réels.

On remarquera que l'ensemble des sous-domaines privilégiés de catégorie (a) est fermé par intersections finis ou infinies et a pour élément l'ensemble de tous les arbres. De ceci il s'ensuit que, pour tout sous-ensemble A du domaine $\mathrm{dom}(\pi_4)$, le plus petit (au sens de l'inclusion) sous-domaine privilégié de catégorie (a) qui contiennent A existe toujours.

En effet cet ensemble est égal à l'intersection de tous les sous-domaines privilégiés de catégorie (a) qui contiennent A, sous-domaines parmi lesquels figure au moins $\mathrm{dom}(\pi_4)$.

Dans le tableau 2.3 on trouvera des exemples de notation Prolog IV pour exprimer que X appartient à l'une des 15 formes possible de sous-domaines privilégiés avec les deux variantes. Tous ces exemples sont présentés sous forme d'une disjonction multiple formant une requête.

TAB. 2.3 – *Exemples d'appartenances à des sous-domaines privilégiés de base*

```
>> X ~ tree;                                  % 1
   X ~ finite;                                % 2
   X ~ infinite;                              % 3
   X ~ list;                                  % 4
   X ~ nlist;                                 % 5
   X ~ [tree,tree,tree];                      % 6
   X ~ [cc(1,2),co(4,8.1),gt(`>9.1`),real];   % 7
   X ~ [cc(1,2) u co(4,8), gt(9), real];      % 7 variante
   X ~ leaf;                                  % 8
   X ~ nleaf;                                 % 9
   X ~ ident;                                 % 10
   X ~ nident;                                % 11
   X ~ oc(`<2.1`,9);                          % 12
   X ~ oc(2.1,3) u cc(3,5) u gt(5);           % 12 variante
   X ~ nreal;                                 % 13
   Y ex X = trio(X,Y,6), Y = duo(X,Y);        % 14
   X ~ ntree.                                 % 15
```

2.3.3 Opérations de la structure π_4

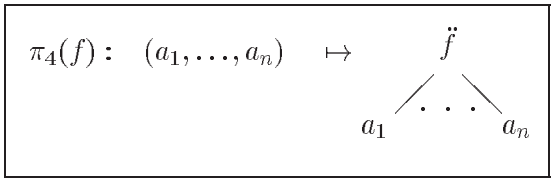
L'ensemble op(π_4) des symboles d'opérations f est constitué de

- tout identificateur avec arité, id/n, qui n'est pas un identificateur réservé du tableau 2.4,
- toute constante représentant un nombre rationnel.

TAB. 2.4 – *Identificateurs réservés pour les symboles de relation de π_4*

abs/2	bnleaf/1	div/3	ln/2	outoo/3
and/2	bnlist/1	divlin/3	log/2	pi/1
arccos/2	bnot/2	eq/2	lt/2	plus/3
arcsin/2	• bnprime/2	equiv/2	ltlin/2	pluslin/3
arctan/2	bnreal/2	exp/2	max/3	power/3
band/3	boc/4	finite/1	min/3	• prime/1
bcc/4	boo/4	floor/2	minus/3	real/1
bco/4	bor/3	• gcd/3	minuslin/3	root/3
bdif/3	boutcc/4	ge/2	• mod/3	sin/2
beq/3	boutco/4	gelin/2	n/3	sinh/2
bequiv/3	boutlist/3	gt/2	nidentifier/1	size/2
bfinite/2	boutoc/4	gtlin/2	nint/1	sqrt/2
bge/3	boutoo/4	identifier/1	nlist/1	square/2
bgt/3	• bprime/2	if/4	nleaf/1	tan/2
bidentifier/2	breal/2	impl/2	not/1	tanh/2
bimpl/3	bxor/3	index/3	• nprime/1	times/3
binfinite/2	cc/3	infinite/1	ntree/1	timeslin/3
binlist/3	ceil/2	inlist/2	nreal/2	tree/1
bint/2	co/3	• intdiv/3	oc/3	u/3
ble/3	conc/3	int/1	oo/3	uminus/2
bleaf/1	cos/2	• lcm/3	or/2	uminuslin/2
blist/1	cosh/2	le/2	outcc/3	uplus/2
blt/3	cot/2	leaf/1	outco/3	upluslin/2
bnidentifier/2	coth/2	lelin/2	outlist/2	xor/2
bnint/2	dif/2	list/1	outoc/3	

L'opération $\pi_4(f)$ que la structure π_4 associe au symbole n-aire f consiste à partir d'une suite $a_1, \ldots, a_n$ d'arbres à construire un arbre dont la racine est étiquetée, à peu de chose près par f, et dont la suite de fils immédiats est $a_1, \ldots, a_n$. Plus précisément l'opération $\pi_4(f)$ est l'application



où $\ddot{f}$ est, soit un identificateur égal[6] à f, soit le nombre rationnel représenté par la constante f. Bien entendu, les a_i sont des éléments du domaine dom(π_4) et l'entier n peut être nul.

6. Si l'identificateur f commence par le caractère ASCII accent circonflexe, on enlève ce caractère dans $\ddot{f}$. Ceci permet de créer des arbres étiquetés par des identificateurs réservés.

2.3.4 Relations de la structure π_4

L'ensemble rel(π_4) des symboles de relation r est constitué de

- tout identificateur avec arité, id/n, du tableau 2.4 à la page 74,
- le symbole dif$'/2$,
- tout symbole de la forme in$A/1$, où A est un sous-domaine privilégié.

Le symbole dif$'/2$ et les symboles de la forme in$A/1$ ne sont pas accessibles au programmeur. Ils ont été introduits pour énoncer les propriétés utilisés par les algorithmes de résolution des contraintes.

Les tableaux 2.5, 2.6 et 2.7 donnent l'association symbole-relation

$r \mapsto \pi_4(r)$

pour chaque symbole r de rel(π_4). Attention, les relations dont les noms sont précédés d'un point dans le tableau 2.4 ne sont pas implantées.

TAB. 2.5 – *Relations de la structure de base π_4, première partie*

Arbres et listes, relations

1 tree/1	$\mapsto$	$\{a \in \mathbf{A} \mid a$ n'est assujetti à aucune condition$\}$
2 ntree/1	$\mapsto$	$\{a \in \mathbf{A} \mid a$ ne peut exister$\}$
3 finite/1	$\mapsto$	$\{a \in \mathbf{A} \mid a$ est fini$\}$
4 infinite/1	$\mapsto$	$\{a \in \mathbf{A} \mid a$ est infini$\}$
5 list/1	$\mapsto$	$\{a \in \mathbf{A} \mid a$ est une liste$\}$
6 nlist/1	$\mapsto$	$\{a \in \mathbf{A} \mid a$ n'est pas une liste$\}$
7 leaf/1	$\mapsto$	$\{a \in \mathbf{A} \mid a$ est un arbre d'un nœud$\}$
8 nleaf/1	$\mapsto$	$\{a \in \mathbf{A} \mid a$ est un arbre de plus d'un nœud$\}$
9 real/1	$\mapsto$	$\{a \in \mathbf{A} \mid a$ est un nombre réel$\}$
10 nreal/1	$\mapsto$	$\{a \in \mathbf{A} \mid a$ n'est pas un nombre réel$\}$
11 identifier/1	$\mapsto$	$\{a \in \mathbf{A} \mid a$ est un identificateur$\}$
12 nidentifier/1	$\mapsto$	$\{a \in \mathbf{A} \mid a$ n'est pas un identificateur$\}$
13 eq/2	$\mapsto$	$\{(a,b) \in \mathbf{A}^2 \mid a = b\}$
14 dif/2	$\mapsto$	$\{(a,b) \in \mathbf{A}^2 \mid a \neq b\}$
15 inlist/2	$\mapsto$	$\{(a,b) \in \mathbf{A}\times\mathbf{L} \mid a$ figure dans $b\}$
16 outlist/2	$\mapsto$	$\{(a,b) \in \mathbf{A}\times\mathbf{L} \mid a$ ne figure pas dans $b\}$

Arbres et listes, pseudo-opérations produisant des booléens

17 bfinite/2	$\mapsto$	$\{(a,b) \in \mathbf{B}\times\mathbf{A} \mid a = 1$ ssi $b \in \pi_4(\text{finite}/1)\}$
18 binfinite/2	$\mapsto$	$\{(a,b) \in \mathbf{B}\times\mathbf{A} \mid a = 1$ ssi $b \in \pi_4(\text{infinite}/1)\}$
19 blist/2	$\mapsto$	$\{(a,b) \in \mathbf{B}\times\mathbf{A} \mid a = 1$ ssi $b \in \pi_4(\text{list}/1)\}$
20 bnlist/2	$\mapsto$	$\{(a,b) \in \mathbf{B}\times\mathbf{A} \mid a = 1$ ssi $b \in \pi_4(\text{nlist}/1)\}$
21 bleaf/2	$\mapsto$	$\{(a,b) \in \mathbf{B}\times\mathbf{A} \mid a = 1$ ssi $b \in \pi_4(\text{leaf}/1)\}$
22 bnleaf/2	$\mapsto$	$\{(a,b) \in \mathbf{B}\times\mathbf{A} \mid a = 1$ ssi $b \in \pi_4(\text{nleaf}/1)\}$
23 breal/2	$\mapsto$	$\{(a,b) \in \mathbf{B}\times\mathbf{A} \mid a = 1$ ssi $b \in \pi_4(\text{real}/1)\}$
24 bnreal/2	$\mapsto$	$\{(a,b) \in \mathbf{B}\times\mathbf{A} \mid a = 1$ ssi $b \in \pi_4(\text{nreal}/1)\}$
25 bidentifier/2	$\mapsto$	$\{(a,b) \in \mathbf{B}\times\mathbf{A} \mid a = 1$ ssi $b \in \pi_4(\text{identifier}/1)\}$
26 bnidentifier/2	$\mapsto$	$\{(a,b) \in \mathbf{B}\times\mathbf{A} \mid a = 1$ ssi $b \in \pi_4(\text{nidentifier}/1)\}$
27 beq/3	$\mapsto$	$\{(a,b,c) \in \mathbf{B}\times\mathbf{A}^2 \mid a = 1$ ssi $(b,c) \in \pi_4(\text{eq}/2)\}$
28 bdif/3	$\mapsto$	$\{(a,b,c) \in \mathbf{B}\times\mathbf{A}^2 \mid a = 1$ ssi $(b,c) \in \pi_4(\text{dif}/2)\}$
29 binlist/3	$\mapsto$	$\{(a,b,c) \in \mathbf{B}\times\mathbf{A}\times\mathbf{L} \mid a = 1$ ssi $(b,c) \in \pi_4(\text{inlist}/2)\}$
30 boutlist/3	$\mapsto$	$\{(a,b,c) \in \mathbf{B}\times\mathbf{A}\times\mathbf{L} \mid a = 1$ ssi $(b,c) \in \pi_4(\text{outlist}/2)\}$

Arbres et listes, pseudo-opérations courantes

31 size/2	$\mapsto$	$\{(a,b) \in \mathbf{Z}\times\mathbf{L} \mid a = \lvert b\rvert\}$
32 conc/3	$\mapsto$	$\{(a,b,c) \in \mathbf{L}^3 \mid a = b \bullet c\}$
33 index/3	$\mapsto$	$\{(a,b,c) \in \mathbf{A}\times\mathbf{L}\times\mathbf{Z} \mid 1 \leq c \leq \lvert b\rvert$ et a est le $c^{\text{ième}}$ élément de $b\}$
34 u/3	$\mapsto$	$\{(a,b,c) \in \mathbf{A}^3 \mid a = b$ ou $a = c\}$
35 n/3	$\mapsto$	$\{(a,b,c) \in \mathbf{A}^3 \mid a = b$ et $a = c\}$
36 if/4	$\mapsto$	$\{(a,b,c,d) \in \mathbf{A}\times\mathbf{B}\times\mathbf{A}^2 \mid$ si $b = 1$ alors $a = b$ sinon $a = c\}$

Booléens, relations

37 not/1	$\mapsto$	$\{a \in \mathbf{B} \mid a = 0)\}$
38 xor/2	$\mapsto$	$\{(a,b) \in \mathbf{B}^2 \mid a \neq b\}$
39 or/2	$\mapsto$	$\{(a,b) \in \mathbf{B}^2 \mid a = 1$ ou $b = 1\}$
40 and/2	$\mapsto$	$\{(a,b) \in \mathbf{B}^2 \mid a = 1$ et $b = 1\}$
41 impl/2	$\mapsto$	$\{(a,b) \in \mathbf{B}^2 \mid$ si $a = 1$ alors $b = 1\}$
42 equiv/2	$\mapsto$	$\{(a,b) \in \mathbf{B}^2 \mid a = b\}$

A : ensemble des arbres, **B** : ensemble $\{0,1\}$,
L : ensemble des listes, **Z** : ensemble des entiers, **R** : ensemble des nombres réels.

TAB. 2.6 – *Relations de la structure de base π_4, deuxième partie*

Booléens, pseudo-opérations

43 bnot/2	$\mapsto$	$\{(a,b) \in \mathbf{B}^2 \mid a = 1 \text{ ssi } b \in \pi_4(\text{not}/1)\}$
44 bxor/3	$\mapsto$	$\{(a,b,c) \in \mathbf{B}^3 \mid a = 1 \text{ ssi } (b,c) \in \pi_4(\text{xor}/2)\}$
45 bor/3	$\mapsto$	$\{(a,b,c) \in \mathbf{B}^3 \mid a = 1 \text{ ssi } (b,c) \in \pi_4(\text{or}/2)\}$
46 band/3	$\mapsto$	$\{(a,b,c) \in \mathbf{B}^3 \mid a = 1 \text{ ssi } (b,c) \in \pi_4(\text{and}/2)\}$
47 bimpl/3	$\mapsto$	$\{(a,b,c) \in \mathbf{B}^3 \mid a = 1 \text{ ssi } (b,c) \in \pi_4(\text{impl}/2)\}$
48 bequiv/3	$\mapsto$	$\{(a,b,c) \in \mathbf{B}^3 \mid a = 1 \text{ ssi } (b,c) \in \pi_4(\text{equiv}/2)\}$

Réels, relations

49 int/1	$\mapsto$	$\{a \in \mathbf{R} \mid a \text{ est entier}\}$
50 nint/1	$\mapsto$	$\{a \in \mathbf{R} \mid a \text{ n'est pas entier}\}$
51 prime/1	$\mapsto$	$\{a \in \mathbf{R} \mid a \text{ est un entier premier}\}$
52 nprime/1	$\mapsto$	$\{a \in \mathbf{R} \mid a \text{ est un entier non premier}\}$
53 gt/2	$\mapsto$	$\{(a,b) \in \mathbf{R}^2 \mid a > b\}$
54 gtlin/2	$\mapsto$	$\{(a,b) \in \mathbf{R}^2 \mid a > b\}$
55 lt/2	$\mapsto$	$\{(a,b) \in \mathbf{R}^2 \mid a < b\}$
56 ltlin/2	$\mapsto$	$\{(a,b) \in \mathbf{R}^2 \mid a < b\}$
57 ge/2	$\mapsto$	$\{(a,b) \in \mathbf{R}^2 \mid a \geq b\}$
58 gelin/2	$\mapsto$	$\{(a,b) \in \mathbf{R}^2 \mid a \geq b\}$
59 le/2	$\mapsto$	$\{(a,b) \in \mathbf{R}^2 \mid a \leq b\}$
60 lelin/2	$\mapsto$	$\{(a,b) \in \mathbf{R}^2 \mid a \leq b\}$
61 cc/3	$\mapsto$	$\{(a,b,c) \in \mathbf{R}^3 \mid a \in [b,c]\}$
62 co/3	$\mapsto$	$\{(a,b,c) \in \mathbf{R}^3 \mid a \in [b,c)\}$
63 oc/3	$\mapsto$	$\{(a,b,c) \in \mathbf{R}^3 \mid a \in (b,c]\}$
64 oo/3	$\mapsto$	$\{(a,b,c) \in \mathbf{R}^3 \mid a \in (b,c)\}$
65 outcc/3	$\mapsto$	$\{(a,b,c) \in \mathbf{R}^3 \mid a \notin [b,c]\}$
66 outco/3	$\mapsto$	$\{(a,b,c) \in \mathbf{R}^3 \mid a \notin [b,c)\}$
67 outoc/3	$\mapsto$	$\{(a,b,c) \in \mathbf{R}^3 \mid a \notin (b,c]\}$
68 outoo/3	$\mapsto$	$\{(a,b,c) \in \mathbf{R}^3 \mid a \notin (b,c)\}$

Réels, pseudo-opérations produisant des booléens

69 bint/2	$\mapsto$	$\{(a,b) \in \mathbf{B}\times\mathbf{R} \mid a = 1 \text{ ssi } b \in \pi_4(\text{int}/1)\}$
70 bnint/2	$\mapsto$	$\{(a,b) \in \mathbf{B}\times\mathbf{R} \mid a = 1 \text{ ssi } b \in \pi_4(\text{nint}/1)\}$
71 bprime/2	$\mapsto$	$\{(a,b) \in \mathbf{B}\times\mathbf{R} \mid a = 1 \text{ ssi } b \in \pi_4(\text{prime}/1)\}$
72 bnprime/2	$\mapsto$	$\{(a,b) \in \mathbf{B}\times\mathbf{R} \mid a = 1 \text{ ssi } b \in \pi_4(\text{nprime}/1)\}$
73 bgt/3	$\mapsto$	$\{(a,b,c) \in \mathbf{B}\times\mathbf{R}^2 \mid a = 1 \text{ ssi } (b,c) \in \pi_4(\text{gt}/2)\}$
74 blt/3	$\mapsto$	$\{(a,b,c) \in \mathbf{B}\times\mathbf{R}^2 \mid a = 1 \text{ ssi } (b,c) \in \pi_4(\text{lt}/2)\}$
75 bge/3	$\mapsto$	$\{(a,b,c) \in \mathbf{B}\times\mathbf{R}^2 \mid a = 1 \text{ ssi } (b,c) \in \pi_4(\text{ge}/2)\}$
76 ble/3	$\mapsto$	$\{(a,b,c) \in \mathbf{B}\times\mathbf{R}^2 \mid a = 1 \text{ ssi } (b,c) \in \pi_4(\text{le}/2)\}$
77 bcc/4	$\mapsto$	$\{(a,b,c,d) \in \mathbf{B}\times\mathbf{R}^3 \mid a = 1 \text{ ssi } (b,c,d) \in \pi_4(\text{cc}/3)\}$
78 bco/4	$\mapsto$	$\{(a,b,c,d) \in \mathbf{B}\times\mathbf{R}^3 \mid a = 1 \text{ ssi } (b,c,d) \in \pi_4(\text{co}/2)\}$
79 boc/4	$\mapsto$	$\{(a,b,c,d) \in \mathbf{B}\times\mathbf{R}^3 \mid a = 1 \text{ ssi } (b,c,d) \in \pi_4(\text{oc}/2)\}$
80 boo/4	$\mapsto$	$\{(a,b,c,d) \in \mathbf{B}\times\mathbf{R}^3 \mid a = 1 \text{ ssi } (b,c,d) \in \pi_4(\text{oo}/2)\}$
81 boutcc/4	$\mapsto$	$\{(a,b,c,d) \in \mathbf{B}\times\mathbf{R}^3 \mid a = 1 \text{ ssi } (b,c,d) \in \pi_4(\text{outcc}/2)\}$
82 boutco/4	$\mapsto$	$\{(a,b,c,d) \in \mathbf{B}\times\mathbf{R}^3 \mid a = 1 \text{ ssi } (b,c,d) \in \pi_4(\text{outco}/2)\}$
83 boutoc/4	$\mapsto$	$\{(a,b,c,d) \in \mathbf{B}\times\mathbf{R}^3 \mid a = 1 \text{ ssi } (b,c,d) \in \pi_4(\text{outoc}/2)\}$
84 boutoo/4	$\mapsto$	$\{(a,b,c,d) \in \mathbf{B}\times\mathbf{R}^3 \mid a = 1 \text{ ssi } (b,c,d) \in \pi_4(\text{outoo}/2)\}$

A : ensemble des arbres, **B** : ensemble $\{0,1\}$,
L : ensemble des listes, **Z** : ensemble des entiers, **R** : ensemble des nombres réels.

TAB. 2.7 – *Relations de la structure de base π_4, troisième partie*

Réels et entiers, pseudo-opérations courantes

85 floor/2 $\mapsto$ $\{(a,b) \in \mathbf{Z}\times\mathbf{R} \mid a = \lfloor b \rfloor\}$
86 ceil/2 $\mapsto$ $\{(a,b) \in \mathbf{Z}\times\mathbf{R} \mid a = \lceil b \rceil\}$
87 gcd/3 $\mapsto$ $\{(a,b,c) \in \mathbf{Z}^3 \mid a \geq 0, b \geq 0, c \geq 0 \text{ et } a = \text{pgcd}(b,c)\}$
88 lcm/3 $\mapsto$ $\{(a,b,c) \in \mathbf{Z}^3 \mid a \geq 0, b \geq 0, c \geq 0 \text{ et } a = \text{ppcm}(b,c)\}$
89 intdiv/3 $\mapsto$ $\{(a,b,c) \in \mathbf{Z}^3 \mid \text{il existe } d \in \mathbf{Z} \text{ tel que } b = ac + d \text{ et } c > d \geq 0\}$

Réels, pseudo-opérations courantes

90 uplus/2 $\mapsto$ $\{(a,b) \in \mathbf{R}^2 \mid a = +b\}$
91 upluslin/2 $\mapsto$ $\{(a,b) \in \mathbf{R}^2 \mid a = +b\}$
92 uminus/2 $\mapsto$ $\{(a,b) \in \mathbf{R}^2 \mid a = -b\}$
93 uminuslin/2 $\mapsto$ $\{(a,b) \in \mathbf{R}^2 \mid a = -b\}$
94 abs/2 $\mapsto$ $\{(a,b) \in \mathbf{R}^2 \mid a = |b|\}$
95 plus/3 $\mapsto$ $\{(a,b,c) \in \mathbf{R}^3 \mid a = b + c\}$
96 pluslin/3 $\mapsto$ $\{(a,b,c) \in \mathbf{R}^3 \mid a = b + c\}$
97 minus/3 $\mapsto$ $\{(a,b,c) \in \mathbf{R}^3 \mid a = b - c\}$
98 minuslin/3 $\mapsto$ $\{(a,b,c) \in \mathbf{R}^3 \mid a = b - c\}$
99 times/3 $\mapsto$ $\{(a,b,c) \in \mathbf{R}^3 \mid a = b \times c\}$
100 timeslin/3 $\mapsto$ $\{(a,b,c) \in \mathbf{R}^3 \mid a = b \times c\}$
101 div/3 $\mapsto$ $\{(a,b,c) \in \mathbf{R}^3 \mid c \neq 0 \text{ et } a = b/c\}$
102 divlin/3 $\mapsto$ $\{(a,b,c) \in \mathbf{R}^3 \mid c \neq 0 \text{ et } a = b/c\}$
103 mod/3 $\mapsto$ $\{(a,b,c) \in \mathbf{R}^3 \mid 0 \leq a < c \text{ et } a = b \bmod c\}$
104 min/3 $\mapsto$ $\{(a,b,c) \in \mathbf{R}^3 \mid a = \min(b,c)\}$
105 max/3 $\mapsto$ $\{(a,b,c) \in \mathbf{R}^3 \mid a = \max(b,c)\}$

Réels, pseudo-opérations correspondant aux fonctions spéciales

106 pi/1 $\mapsto$ $\{a \in \mathbf{R} \mid a = \pi\}$
107 square/2 $\mapsto$ $\{(a,b) \in \mathbf{R}^2 \mid a = b^2\}$
108 sqrt/2 $\mapsto$ $\{(a,b) \in \mathbf{R}^2 \mid b \geq 0 \text{ et } a = \sqrt{b}\}$
109 power/3 $\mapsto$ $\{(a,b,c) \in \mathbf{R}^2\times\mathbf{Z} \mid c \geq 0 \text{ et } a = b^c\}$
110 root/3 $\mapsto$ $\{(a,b,c) \in \mathbf{R}^2\times\mathbf{Z} \mid c \geq 0 \text{ et } a = \sqrt[c]{b}\}$
111 arccos/2 $\mapsto$ $\{(a,b) \in \mathbf{R}^2 \mid 0 \leq a \leq \pi \text{ et } b = \cos a\}$
112 arcsin/2 $\mapsto$ $\{(a,b) \in \mathbf{R}^2 \mid -\pi/2 \leq a \leq \pi/2 \text{ et } b = \sin a\}$
113 arctan/2 $\mapsto$ $\{(a,b) \in \mathbf{R}^2 \mid -\pi/2 < a < \pi/2 \text{ et } b = \tan a\}$
114 cos/2 $\mapsto$ $\{(a,b) \in \mathbf{R}^2 \mid a = \cos b\}$
115 cosh/2 $\mapsto$ $\{(a,b) \in \mathbf{R}^2 \mid a = \cosh b\}$
116 cot/2 $\mapsto$ $\{(a,b) \in \mathbf{R}^2 \mid b \bmod 2\pi \neq 0 \text{ et } a = \cot b\}$
117 coth/2 $\mapsto$ $\{(a,b) \in \mathbf{R}^2 \mid a = \coth b\}$
118 exp/2 $\mapsto$ $\{(a,b) \in \mathbf{R}^2 \mid a = \mathrm{e}^b\}$
119 ln/2 $\mapsto$ $\{(a,b) \in \mathbf{R}^2 \mid b > 0 \text{ et } a = \ln b\}$
120 log/2 $\mapsto$ $\{(a,b) \in \mathbf{R}^2 \mid b > 0 \text{ et } a = \log b\}$
121 sin/2 $\mapsto$ $\{(a,b) \in \mathbf{R}^2 \mid a = \sin b\}$
122 sinh/2 $\mapsto$ $\{(a,b) \in \mathbf{R}^2 \mid a = \sinh b\}$
123 tan/2 $\mapsto$ $\{(a,b) \in \mathbf{R}^2 \mid b \bmod 2\pi \neq \pi/2 \text{ et } a = \tan b\}$
124 tanh/2 $\mapsto$ $\{(a,b) \in \mathbf{R}^2 \mid a = \tanh b\}$

Relations internes, pour chaque sous-domaine privilégié A

125 dif′/2 $\mapsto$ $\{(a,b) \in \mathbf{A}^2 \mid a \neq b\}$
126 in$A$/1 $\mapsto$ $\{a \in \mathbf{A} \mid a \text{ est élément du sous-domaine privilégié } A\}$

A : ensemble des arbres, **B** : ensemble $\{0,1\}$,
L : ensemble des listes, **Z** : ensemble des entiers, **R** : ensemble des nombres réels.

2.4 Axiomatisation de π_4

2.4.1 Généralités

Soit p une contrainte dans laquelle $x_1, \ldots, x_n$ sont les seules variables qui ont des occurrences libres. Résoudre p consiste à trouver les jeux de valeurs possibles de ces variables qui respectent la condition exprimée par p dans la structure π_4. Il faut donc trouver l'ensemble des solutions σ de p dans la structure π_4, c'est-à-dire l'ensemble des affectations σ de variables qui sont telles que la combinaison de σ avec la structure π_4 forment une interprétation I pour laquelle

$I^*(p) =$ vrai.

Cet ensemble de solutions étant généralement infini, on s'attachera plutôt à simplifier p en une contrainte équivalente q qui rend l'ensemble recherché de solutions plus « visible ». En particulier si p n'a pas de solution dans π_4 on aimerait que q se réduise à la constante logique *false*.

Compte tenu de la richesse de la structure π_4, il est complètement illusoire de vouloir atteindre ce dernier objectif. Il faut donc se résigner à utiliser un «solveur» de contraintes incomplet. Mais comment, sans entrer dans tous les détails de son implantation faire connaître au programmeur toutes les capacités et incapacités du solveur ? C'est là qu'intervient l'axiomatisation de la structure π_4. L'idée est de retenir l'ensemble exact $\mathcal{T}(\pi_4)$ des propriétés de π_4 utilisées par l'algorithme de résolution et de garantir que ces propriété sont utilisées au mieux. Chaque propriété p retenue sera une proposition du premier ordre telle que

$\pi_4^*(p) =$ vrai

et l'ensemble $\mathcal{T}(\pi_4)$ de ces p sera une théorie du premier ordre qui constituera une axiomatisation partielle de π_4 c'est-à-dire qu'il existera des propositions p tels qu'à la fois

$$\mathcal{T}(\pi_4) \not\models p \quad \text{et} \quad \mathcal{T}(\pi_4) \not\models \neg p$$

Lorsque le programmeur donnera une contrainte p à résoudre à Prolog IV le système lui retournera une contrainte q non seulement équivalente à p dans la structure π_4 mais dans toutes les structures π_4' ayant les propriétés $\mathcal{T}(\pi_4)$, ce qui s'écrit

$$\mathcal{T}(\pi_4) \models p \equiv q$$

Cette contrainte q ne se réduira à la constante logique *false* que si elle n'a pas de solutions dans la structure π_4 et si en plus elle n'a de solutions dans aucune autre structure qui a les propriétés $\mathcal{T}(\pi_4)$, c'est-à-dire si

$$\mathcal{T}(\pi_4) \models p \equiv \mathit{false}$$

Comme le montre le tableau 2.8 la théorie $\mathcal{T}(\pi_4)$ est composé de 25 schémas possibles d'axiomes regroupés en 8 catégories. Il s'agit d'un premier essai d'axiomatisation et il est fort probable qu'il y ait encore des erreurs.

TAB. 2.8 – *Propriétés retenues de la structure π_4*

1	Sous-domaine vide	Sous-domaines privilégiés
2	Sous-domaine universel	’’ ’’
3	Intersection de sous-domaines	’’ ’’
4	Opérations distinctes, résultats distincts	Opérations sur les arbres
5	Au moins une solution	’’ ’’
6	Propagation des égalités	’’ ’’
7	Sous-domaines singletoniques	Interactions des sous-domaines avec les opérations
8	Réduction de sous-domaines	’’ ’’
9	Arbres infinis	’’ ’’
10	Taille finie des listes	’’ ’’
11	Réduction de sous-domaines	Relation générales
12	Construction de contrainte	’’ ’’
13	Multiplication par scalaire positif	Contraintes linéaires
14	Somme de contraintes	’’ ’’
15	Respect de l'ordre	’’ ’’
16	Constante cachée	’’ ’’
17	Linéarisation de l'égalité	’’ ’’
18	Typage numérique	Relations à linéariser
19	Linéarisation de contrainte	’’ ’’
20	Détection d'égalités	Relations dif, bdif et beq
21	Egalités, bdif et beq	’’ ’’
22	Détection de diségalités	’’ ’’
23	Diségalités, bdif et beq	’’ ’’
24	Propagation du dif	Propagation supplémentaire d'égalités
25	Propagation du dif′	’’ ’’

2.4.2 Axiomatisation des sous-domaines privilégiés

Les premières propriétés retenues concernent l'appartenance à un sous-domaine privilégié :

1 $\forall x$
$\text{in}\emptyset\, x \;\equiv\; \mathit{false}$

2 $\forall x$
$\text{in}\mathbf{A}\, x$

3 $\forall x$
$$\left(\begin{array}{c} \text{in}A\, x \\ \wedge\ \text{in}B\, x \end{array}\right) \;\equiv\; \text{in}A\cap B\, x$$

où **A** est l'ensemble de tous les arbres c'est-à-dire tout le domaine et A, B des sous-domaines privilégiés quelconques.

La propriété 1 exprime que l'ensemble vide ne contient aucun élément, la propriété 2 que tout individu est un arbre et la troisième qu'un individu qui appartient à deux sous-domaines privilégiés appartient aussi à leur intersection, qui comme nous l'avons vu est aussi un sous-domaine privilégié. Voici quelques requêtes pour illustrer ceci.

```
>> X ~ ntree.
false.

>> X = X.
X ~ tree.

>> X ~ cc(1,5), X ~ cc(2,6).
X ~ cc(2,5).
```

2.4.3 Axiomatisation des opérations sur les arbres

On retient 3 propriétés :

4 $\forall x_1 \forall y_1 \forall u_1 \ldots \forall u_m \forall v_1 \ldots \forall v_n$
$$\left(\begin{array}{rl} & x_1 = y_1 \\ \wedge & x_1 = f\, u_1 \ldots u_m \\ \wedge & y_1 = g\, v_1 \ldots v_n \end{array} \right) \equiv \mathit{false}$$

5 $\forall u_1 \ldots \forall u_m \exists x_1 \ldots \exists x_n$
$$\left(\begin{array}{rl} & x_1 = t_1(x_1, .., x_n, u_1, \ldots, u_m) \\ & \ldots \\ \wedge & x_n = t_n(x_1, .., x_n, u_1, \ldots, u_m) \end{array} \right)$$

6 $\forall u_1 \ldots \forall u_m \forall v_1 \ldots \forall v_m \forall x_1 \ldots \forall x_n \forall y_1 \ldots \forall y_n$
$$\left(\begin{array}{rl} & x_1 = t_1(x_1, .., x_n, u_1, .., u_m) \\ & \ldots \\ \wedge & x_n = t_n(x_1, .., x_n, u_1, .., u_m) \\ \wedge & y_1 = t_1(y_1, .., y_n, v_1, .., v_m) \\ & \ldots \\ \wedge & y_n = t_n(y_1, .., y_n, v_1, .., v_m) \end{array} \right) \Rightarrow \left(\begin{array}{c} \left(\begin{array}{rl} & u_1 = v_1 \\ & \ldots \\ \wedge & u_m = v_m \end{array} \right) \\ \equiv \\ \left(\begin{array}{rl} & x_1 = y_1 \\ & \ldots \\ \wedge & x_n = y_n \end{array} \right) \end{array} \right)$$

où f et g sont des symboles d'opérations m-aires et n-aires distincts, où les $t_i(x_1, .., x_n, u_1, .., u_m)$ sont des termes construits avec une occurrence de symbole d'opération et des variables prises dans $\{x_1, .., x_n, u_1, .., u_m\}$ et où les $t_i(y_1, .., y_n, v_1, .., v_m)$ sont les mêmes termes dans lequels on a remplacé les x_j par des y_j et les u_j par des v_j.

La propriété 4 exprime que deux arbres construits par des opérations nommées par des symboles distincts sont distincts. Les requêtes qui suivent illustrent ce phénomène.

```
>> f(X)=g(Y).
false.

>> a=b.
false.

>> 1/3=1/4.
false.

>> f(X)=f(Y,X).
false.
```

On notera que dans la dernière requête les deux occurrences de `f` sont considérées comme faisant référence à deux symboles distincts `f/1` et `f/2`. L'arité du symbole est en fait codée dans les virgules et les parenthèses.

La propriété 5 exprime qu'un type particulier de conjonction d'équations admet au moins une solution

```
>> X ex Y ex
   X=f(X,Y), Y=g(X,Y).
true.
```

La propriété 6 permet de décomposer une équation en plusieurs équations.

```
>> f(X,Y) = f(U,V).
Y = V,
X = U,
V ~ tree,
U ~ tree.
```

Elle affirme aussi que la conjonction d'équations de l'axiome 5 admet au plus une solution. Ceci permet des simplifications importantes.

```
>> X = f(f(f(X))).
X = f(X).

>> X = g(g(g(X,Y),Y),Y).
X = g(X,Y),
Y ~ tree.
```

Soit ϕ la structure obtenue en privant π_4 de ses symboles de relations. Il existe d'autres structures que cette structure d'arbre ϕ dans laquelle toutes ces propriétés sont vraies. Cependant Michael Maher[7] a montré qu'il existe un ensemble de propriétés, en fait équivalent à l'ensemble $\mathcal{T}$ des propriétés 4,5,6, qui forme une axiomatisation complète de ϕ. Autrement dit si p est une proposition alors, pour toute structure ϕ' dans laquelle les propositions $\mathcal{T}$ sont vraies, on a

$\phi(p) = \phi'(p)$.

2.4.4 Axiomatisations des interactions des sous-domaines avec les opérations

Les 4 propriétés retenues qui suivent concernent les interactions des contraintes de sous-domaines et des opérations :

7. Complete axiomatizations of the algebra of finite, rational and infinite trees. *Technical report, IBM T.J. Watson Research Center*, 1988

7 $\forall x_1 \ldots \forall x_n$
$$\begin{pmatrix} x_1 = t_1(x_1, \ldots, x_n) \\ \ldots \\ \wedge\ x_n = t_n(x_1, \ldots, x_n) \end{pmatrix} \equiv \begin{pmatrix} \text{in}\{a_1\}\, x_1 \\ \ldots \\ \wedge\ \text{in}\{a_n\}\, x_n \end{pmatrix}$$

8 $\forall x_1 \ldots \forall x_n$
$$\begin{pmatrix} x_0 = f x_1 \ldots x_n \\ \wedge\ \text{in}A_0\, x_0 \\ \wedge\ \text{in}A_1\, x_1 \\ \ldots \\ \wedge\ \text{in}A_n\, x_n \end{pmatrix} \equiv \begin{pmatrix} x_0 = f x_1 \ldots x_n \\ \wedge\ \text{in}A'_0\, x_0 \\ \wedge\ \text{in}A'_1\, x_1 \\ \ldots \\ \wedge\ \text{in}A'_n\, x_n \end{pmatrix}$$

9 $\forall x_1 \ldots \forall x_n$
$$\begin{pmatrix} x_1 = s_1(x_2) \\ \wedge\ x_2 = s_2(x_3) \\ \ldots \\ \wedge\ x_n = s_n(x_1) \end{pmatrix} \Rightarrow \text{in}\mathbf{I}\, x_1$$

10 $\forall x_1 \ldots \forall x_n \forall y_1 \ldots \forall y_n$
$$\begin{pmatrix} y_1 = .\, x_1 y_2 \\ \wedge\ y_2 = .\, x_2 y_3 \\ \ldots \\ \wedge\ y_n = .\, x_n y_1 \end{pmatrix} \Rightarrow \text{in}\mathbf{K}\, y_1$$

où les $t_i(x_1, \ldots, x_n)$ sont des termes construits avec un symbole d'opération et des variables prises dans $\{x_1, \ldots, x_n\}$, où les a_i sont des arbres doublement rationnels vérifiant l'équivalence 7, où les A_i et A'_i sont des sous-domaines privilégiés de catégorie (a) vérifiant l'équivalence 8, où les $s_i(x_j)$ sont des termes construits avec un symbole d'opération et au moins une occurrence de x_j, où **I** est l'ensemble des arbres infinis et où **K** est l'ensemble des arbres qui ne sont pas des listes. Rappelons que les listes sont construites à l'aide du constructeur point.

La propriété 7 établit l'équivalence entre deux propriétés : «être contraint par les opérations et les égalités à n'avoir qu'une seule valeur» et «appartenir à un sous-domaine privilégié singletonique».

```
>> X ~ cc(2,2).
X = 2.
```

La propriété 8 permet de réduire les sous-domaines des variables intervenant dans une opération et entre autres d'obtenir le sous-domaine vide, qui d'après la propriété 1 à la page 80 va produire *false*.

```
>> X = [Y|Z],
   X ~ list,
   Y ~ tree,
   Z ~ tree.
X = [Y|Z],
Y ~ tree,
Z ~ list.

>> X = [Y],
   X ~ real,
   Y ~ real.
false.
```

La propriété 9 correspond au fameux « occurs-check » qui détecte les arbres infinis.

```
>> X = f(g(X,U)), X ~ finite.
false.
```

Enfin la propriété 10 interdit qu'une liste ait une infinité d'éléments.

```
>> X = [1|X], X ~ list.
false.
```

2.4.5 Axiomatisation des relations générales

Les deux prochaines propriétés retenues concernent tous les symboles de relation r à l'exception de ceux qui se terminent par « lin » et de ceux qui sont inaccessibles au programmeur.

11 $$\forall x_1 \ldots \forall x_n \begin{pmatrix} & r\,x_1 \ldots x_n \\ \wedge & \text{in}A_1\,x_1 \\ & \ldots \\ \wedge & \text{in}A_n\,x_n \end{pmatrix} \equiv \begin{pmatrix} & r\,x_1 \ldots x_n \\ \wedge & \text{in}A'_1\,x_1 \\ & \ldots \\ \wedge & \text{in}A'_n\,x_n \end{pmatrix}$$

12 $$\forall x_1 \ldots \forall x_n \begin{pmatrix} & r\,x_1 \ldots x_n \\ \wedge & \text{in}B_1\,x_1 \\ & \ldots \\ \wedge & \text{in}B_n\,x_n \end{pmatrix} \equiv c(x_1, \ldots, x_n)$$

Ici les A_i et A'_i sont des sous-domaines privilégiés de catégorie (a) tels que l'équivalence 11 soit vérifiée. La formule $c(x_1, \ldots, x_n)$ est une *formule de base* de l'une des 7 formes :

1	*true*,	constante logique,
2	*false*,	constante logique,
3	$p \wedge q$,	conjonction,
4	$\exists x p$,	quantification,
5	$x \in C$,	sous-domaine privilégié,
6	$x = y$,	égalité,
7	$x_0 = f\,x_1 \ldots x_n$,	construction,

où p et q sont elles-mêmes des formules de base et C un sous-domaine privilégié quelconque. Les B_i sont des sous-domaines privilégiés de catégorie (b) tels que l'équivalence 12 soit respectée dans π_4.

La propriété 11 permet de réduire les sous-domaines des variables intervenant dans une relation générale et entre autres d'obtenir le sous-domaine vide, qui

d'après l'axiome 1 à la page 80 va produire *false*.

```
>> X = conc(Y,Z).
Z ~ list,
Y ~ list,
X ~ list.

>> X = index(Y,I).
X ~ tree,
Y ~ list,
I ~ ge(1).

>> B = beq(X,Y).
Y ~ tree,
X ~ tree,
B ~ cc(0,1).

>> X = cos(X).
X ~ oo(`>0.739085`,`>0.7390852`)

>> gt(square(X),1).
X ~ real

>> X = Y.*.Z,
   X ~ cc(1,3),
   Y ~ cc(-0.5,1.5),
   Z ~ cc(-0.5,1.5).
Z ~ cc(`>0.6666666`,1.5),
Y ~ cc(`>0.6666666`,1.5),
X ~ cc(1,2.25).

>> X = plus(Y,Z),
   X ~ cc(3,4),
   Y ~ cc(0,1),
   Z ~ cc(0,1).
false.
```

La propriété 12 exprime le fait que certaines formes de contraintes, équivalentes dans π_4 à des formules de base, doivent se comporter comme telles. Dans les parties numériques ceci a souvent pour effet de passer à un calcul en précision infinie.

```
>> X = conc(Y,Z),
   X ~ [tree,tree,tree,tree],
   Y ~ tree,
   Z ~ [tree,tree].
A ex B ex C ex D ex
Z = [B,A],
Y = [D,C],
X = [D,C,B,A],
A ~ tree,
B ~ tree,
C ~ tree,
D ~ tree.

>> X = index(Y,I),
   X ~ tree,
   Y ~ [tree,tree,tree,tree],
   I ~ cc(3,3).
I = 3,
Y ~ [tree,tree,X,tree],
X ~ tree.

>> B = beq(X,Y),
   B ~ cc(1,1).
X = Y,
B = 1,
Y ~ tree.

>> square(X,Y),
   X ~ tree,
   Y ~ cc(1/3,1/3).
Y = 1/3,
X = 1/9.

>> X = 1/3.*.6/7.
X = 2/7.

>> 1 = plus(1/3,X).
   X = 2/3.
false.
```

Restriction Dans les axiomes 11 et 12, si r est un des symboles binlist/3, boutlist/3, conc/3, index/3, inlist/2, outlist/2, alors les A_i, A'_i et B_i doivent être des sous-domaines privilégiés que l'on peut obtenir sans faire intervenir l'ensemble des arbres finis ou l'ensemble des arbres infinis.

2.4.6 Axiomatisation des contraintes linéaires

Pour exprimer les axiomes des contraintes linéaires nous introduirons la notation

$$a_0 + a_1x_1 + \cdots + a_nx_n \geq 0 \tag{2.5}$$

souvent abrégée en $a_0 + \sum a_ix_i \geq 0$ pour désigner l'inégalité linéaire écrite sous forme fonctionnelle où le produit correspond à la relation «timeslin», la somme à la relation «pluslin» et le signe $\geq$ à la relation «gelin», où les a_i sont des entiers relatifs et bien entendu les x_i des variables distinctes. On considère aussi que la contrainte 2.5 désigne toute contrainte obtenue en permutant ou associant différemment les monômes a_ix_i qui la composent ou en ajoutant

d'autres monômes dont les coefficients a_i seraient nuls. Nous retiendrons 5 propriétés :

13 $\forall x_1 \ldots \forall x_n$
$a_0 + \sum a_i x_i \geq 0 \;\equiv\; ka_0 + \sum ka_i x_i \geq 0$

14 $\forall x_1 \ldots \forall x_n$
$\left(\begin{array}{ll} & a_0 + \sum a_i x_i \geq 0 \\ \wedge & b_0 + \sum b_i x_i \geq 0 \end{array} \right) \;\Rightarrow\; a_0 + b_0 + \sum (a_i + b_i) x_i \geq 0$

15 $-a_0 \geq 0 \;\equiv\; \mathit{false}$

16 $\forall x_1$
$\left(\begin{array}{ll} & +a_0 + a_1 x_1 \geq 0 \\ \wedge & -a_0 - a_1 x_1 \geq 0 \end{array} \right) \;\equiv\; x_1 = -a_0 / a_1$

17 $\forall x_1 \forall x_2$
$\left(\begin{array}{ll} & x_1 = x_2 \\ \wedge & \mathbf{inR}\, x_1 \\ \wedge & \mathbf{inR}\, x_2 \end{array} \right) \;\Rightarrow\; \left(\begin{array}{ll} & +x_1 - x_2 \geq 0 \\ \wedge & -x_1 + x_2 \geq 0 \end{array} \right)$

où k est un entier strictement positif, où a_0 est strictement positif dans l'axiome 15 et où **R** est l'ensemble des réels.

Les propriétés 13 et 14 expriment que, si on se limite à des coefficients positifs, la conjonction de deux contraintes implique toute combinaison linéaire de celles-ci. La propriété 15 exprime qu'un entier ne peut être à la fois positif et négatif.

```
>> gelin(-5+3*X-2*Y,0),
   gelin(8-6*X+4*Y,0).
false.
```

La propriété 16 permet de détecter les variables qui n'ont qu'une seule valeur possible.

```
>> gelin(-5+3*X,0),
   gelin(5-3*X,0).
X = 5/3.
```

La propriété 17 permet de coder une égalité numérique sous forme d'inégalités linéaires.

```
>> X = Y, gelin(0,X), gelin(Y,1).
false.
```

2.4.7 Axiomatisation des relations à linéariser

Une *contrainte linéaire générale* est une contrainte de l'une des 6 formes

1	*true*,	constante logique,
2	*false*,	constante logique,
3	$p \wedge q$,	conjonction,
4	$\exists x p$,	quantification,
5	dif xy,	non-égalité,
6	$a_0 + a_1 x_1 + \cdots + a_n x_n \geq 0$,	contrainte linéaire de base.

où p et q sont elles-mêmes des contraintes linéaires générales.

Voici les deux propriétés retenues des contraintes construites avec des symboles r de relation se terminant en «lin».

18 $\forall x_1 \ldots \forall x_n$

$$r\,x_1 \ldots x_n \;\Rightarrow\; \left(\begin{array}{rl} & \text{in}\mathbf{R}\,x_1 \\ & \ldots \\ \wedge & \text{in}\mathbf{R}\,x_n \end{array}\right)$$

19 $\forall x_1 \ldots \forall x_n$

$$\left(\begin{array}{rl} & r\,x_1 \ldots x_n \\ \wedge & \text{in}B_1\,x_1 \\ & \ldots \\ \wedge & \text{in}B_n\,x_n \end{array}\right) \;\equiv\; c(x_1, \ldots, x_n)$$

Ici **R** est l'ensemble des réels, $c(x_1, \ldots, x_n)$ est une contrainte linéaire générale et les B_i des sous-domaines privilégiés de catégorie (b) qui vérifient l'équivalence 19 dans π_4.

La propriété 18 signale que les arguments d'une contrainte atomique en «lin» sont toujours des nombres réels.

```
>> ltlin(X,Y).
Y ~ real,
X ~ real.

>> X = timeslin(Y,Z).
Z ~ real,
Y ~ real,
X ~ real.
```

La propriété 19 exprime le fait que certaines formes de contraintes, équivalentes dans π_4 à des formules linéaire générales, doivent se comporter comme ces formules linéaires.

```
>> Y = timeslin(A,X),
   Y = timeslin(B,X),
   A = 2,
   B = 3.
B = 3,
X = 0,
A = 2,
Y = 0.

>> gelin(X,Y),
   lelin(X,Y),
   X = 4.
Y = 4,
X = 4.

>> gelin(X,Y),
   ltlin(X,Y).
false.
```

2.4.8 Axiomatisation des relations dif, bdif et beq

Le traitement des contraintes construites avec les symboles de relation dif, bdif et beq est très complet. Il respecte 6 axiomes dont voici les 4 premiers :

20 $\forall x_1 \forall x_2$

$$\left(\begin{array}{c} \left(\; x_1 = x_2 \; \right) \\ \vee \\ \left(\begin{array}{rl} & +x_1 - x_2 \geq 0 \\ \wedge & -x_1 + x_2 \geq 0 \end{array} \right) \end{array} \right) \Rightarrow \neg \mathrm{dif}\, x_1 x_2$$

21 $\forall x_0 \forall x_1 \forall x_2$

$$\neg \mathrm{dif}\, x_1 x_2 \Rightarrow \left(\begin{array}{rl} & (\mathrm{beq}\, x_0 x_1 x_2 \Rightarrow x_0 = 1) \\ \wedge & (\mathrm{bdif}\, x_0 x_1 x_2 \Rightarrow x_0 = 0) \end{array} \right)$$

22 $\forall x_1 \forall x_2 \forall u_1 \ldots \forall u_m \forall v_1 \ldots \forall v_n$

$$\left(\begin{array}{c} \left(\begin{array}{rl} & \mathrm{in} A_1\, x_1 \\ \wedge & \mathrm{in} A_2\, x_2 \end{array} \right) \\ \vee \\ \left(\begin{array}{rl} & x_1 = f\, u_1 \ldots u_m \\ \wedge & x_2 = g\, v_1 \ldots v_n \end{array} \right) \\ \vee \\ \left(\begin{array}{rl} & +a_0 + x_1 - x_2 \geq 0 \\ \wedge & -a_0 - x_1 + x_2 \geq 0 \end{array} \right) \end{array} \right) \Rightarrow \mathrm{dif}'\, x_1 x_2$$

23 $\forall x_0 \forall x_1 \forall x_2$

$$\mathrm{dif}'\, x_1 x_2 \Rightarrow \left(\begin{array}{rl} & (\mathrm{bdif}\, x_0 x_1 x_2 \Rightarrow x_0 = 1) \\ \wedge & (\mathrm{beq}\, x_0 x_1 x_2 \Rightarrow x_0 = 0) \end{array} \right)$$

où A_1 et A_2 sont des sous-domaines privilégiés disjoints de catégorie (a), où f et g sont des symboles d'opération distincts et où a_0 un entier non nul.

La propriété 20 permet de détecter un certain nombre d'égalités qui sont prises en compte dans la propriété 21.

```
>> X = Y, dif(X,Y).
false.

>> gelin(X,Y), gelin(Y,X), dif(X,Y).
false.

>> X = Y, Z = beq(X,Y).
Z = 1,
X = Y,
Y ~ tree.

>> gelin(X,Y), gelin(Y,X), Z = bdif(X,Y).
Z = 0,
Y ~ real,
X ~ real.
```

La propriété 22 permet de détecter un certain nombre de diségalités qui sont prises en compte dans la propriété 23.

```
>> X ~ finite, Y ~ infinite, Z = beq(X,Y).
Z = 0,
Y ~ infinite,
X ~ finite.

>> X = f(U), Y = g(V), Z = bdif(X,Y).
Z = 1,
Y = g(V),
X = f(U),
V ~ tree,
U ~ tree.

>> X = Y+1, Z = beq(X,Y).
Z = 0,
Y ~ real,
X ~ real.
```

2.4.9 Axiomatisation de la propagation supplémentaire d'égalités

Par négation des relations nommées dif et dif′ ont obtient des égalités. Ces égalités doivent respecter l'équivalent de l'axiome 6 à la page 81 :

24 $\forall u_1 \ldots \forall u_m \forall v_1 \ldots \forall v_m \forall x_1 \ldots \forall x_n \forall y_1 \ldots \forall y_n$

$$\left(\begin{array}{rl} & x_1 = t_1(x_1,..,x_n,u_1,..,u_m) \\ & \ldots \\ \wedge & x_n = t_n(x_1,..,x_n,u_1,..,u_m) \\ \wedge & y_1 = t_1(y_1,..,y_n,v_1,..,v_m) \\ & \ldots \\ \wedge & y_n = t_n(y_1,..,y_n,v_1,..,v_m) \end{array}\right) \Rightarrow \left(\begin{array}{c} \left(\begin{array}{rl} & \neg\mathrm{dif}\, u_1 v_1 \\ & \ldots \\ \wedge & \neg\mathrm{dif}\, u_m v_m \end{array}\right) \\ \equiv \\ \left(\begin{array}{rl} & \neg\mathrm{dif}\, x_1 y_1 \\ & \ldots \\ \wedge & \neg\mathrm{dif}\, x_n y_n \end{array}\right) \end{array}\right)$$

25 $\forall u_1 \ldots \forall u_m \forall v_1 \ldots \forall v_m \forall x_1 \ldots \forall x_n \forall y_1 \ldots \forall y_n$

$$\left(\begin{array}{rl} & x_1 = t_1(x_1,..,x_n,u_1,..,u_m) \\ & \ldots \\ \wedge & x_n = t_n(x_1,..,x_n,u_1,..,u_m) \\ \wedge & y_1 = t_1(y_1,..,y_n,v_1,..,v_m) \\ & \ldots \\ \wedge & y_n = t_n(y_1,..,y_n,v_1,..,v_m) \end{array}\right) \Rightarrow \left(\begin{array}{c} \left(\begin{array}{rl} & \neg\mathrm{dif}'\, u_1 v_1 \\ & \ldots \\ \wedge & \neg\mathrm{dif}'\, u_m v_m \end{array}\right) \\ \equiv \\ \left(\begin{array}{rl} & \neg\mathrm{dif}'\, x_1 y_1 \\ & \ldots \\ \wedge & \neg\mathrm{dif}'\, x_n y_n \end{array}\right) \end{array}\right)$$

Voici des exemples de propagation d'égalités par l'axiome 24

```
>> gelin(X,Y), gelin(Y,X),
   U = f(U,X), V = f(V,Y),
   Z = beq(U,V).
Z = 1,
U = f(U,X),
V = f(V,Y),
Y ~ real,
X ~ real.

>> gelin(X,Y), gelin(Y,X),
   U = f(U,X), V = f(V,Y),
   dif(U,V).
false.
```

et en voici par l'axiome 25

```
>> X ~ finite, Y ~ infinite, Z = beq([X,U],[Y,V]).
Z = 0,
V ~ tree,
U ~ tree,
Y ~ infinite,
X ~ finite.

>> X = f(U), Y = g(V), Z = bdif([X,U],[Y,V]).
Z = 1,
Y = g(V),
X = f(U),
V ~ tree,
U ~ tree.

>> X = Y+1, Z = beq([X,U],[Y,V]).
Z = 0,
V ~ tree,
U ~ tree,
Y ~ real,
X ~ real.
```

2.5 Structures enrichies

2.5.1 Notion générale de programme

Etant donnée une structure ϕ, comprenant notamment une famille de relations $\phi(r)$ indicées par des symboles de rel(ϕ), on s'intéresse à définir de nouvelles relations indicées par des symboles dits *de prédicats* pris dans un ensemble *pred* disjoint de rel(ϕ). Une *définition de symbole de prédicat* n-aire r sera une formule de la forme

$$\forall x_1 \ldots \forall x_n \, (r\, x_1 \ldots x_n \equiv p). \tag{2.6}$$

où p est une formule construite sur l'ensemble de symboles $V \cup \text{op}(\phi) \cup \text{rel}(\phi) \cup \textit{pred}$ ne faisant pas intervenir d'autres occurrences libres de variables que celles de $x_1, \ldots, x_n$. Un ensemble de définitions de prédicats distincts sera un *programme*.

En Prolog IV, où la structure ϕ sera bien entendu π_4, on se limitera à des formules p qui sont des contraintes, c'est-à-dire, on le rappelle, des formules positives existentielles. L'ensemble *pred* sera constitué des couples id/n où id est un identificateur ISO non réservé dans l'arité n (voir figure 2.4 à la page 74).

2.5.2 Programme sous forme clausale

La syntaxe classique d'un programme Prolog reflète plus le fait que le membre droit d'une définition de la forme 2.6 implique le membre gauche que le fait qu'il lui est équivalent. La définition est écrite sous forme d'une ou de plusieurs implications. Dans chacune de ces implications on a coutume de ne pas faire figurer la quantification universelle extérieure et d'adopter une syntaxe de clause.

Une *clause* est une expression de la forme (à droite on donne la syntaxe Prolog IV)

$$r\, s_1 \ldots s_n \;:\!-\; p \qquad\qquad \texttt{r(}s_1\texttt{,} \ldots \texttt{,}s_n\texttt{)} \;:\!-\; p\texttt{.} \tag{2.7}$$

où r est un prédicat n-aire, les s_i des termes ou des pseudo-termes et p une contrainte construite sur l'ensemble de symboles $V \cup \text{op}(\pi_4) \cup \text{rel}(\pi_4) \cup \textit{pred}$. Si p est la contrainte *true* on peut remplacer l'expression 2.7 par $r\, s_1 \ldots s_n$.

Une *forme normale* de la clause 2.7 est une clause de la forme

$$r\, x_1 \ldots x_n \;:\!-\; \exists y_1 \ldots \exists y_m \, (x_1 \sim s_1 \wedge \cdots \wedge x_n \sim s_n \wedge p)$$

où $x_1, \ldots, x_n$ sont des variables distinctes, ne figurant ni dans les s_i ni dans p et où les y_i forment un sur-ensemble des variables qui ont des occurrences libres dans p. Lorsque les s_i sont des termes, les signes $\sim$ peuvent bien entendu être remplacés par des signes d'égalité.

Un *paquet* de clauses définissant le prédicat r est une suite non vide de clause, toutes ayant le même prédicat de tête r. La définition du prédicat r représentée par ce paquet est

$$\forall x_1 \ldots \forall x_n \, (r\, x_1 \ldots x_n \;\equiv\; p_1 \vee \cdots \vee p_m)$$

où $(r\, x_1 \ldots x_n \;:\!-\; p_1)$, …, $(r\, x_1 \ldots x_n \;:\!-\; p_m)$ est la suite des clauses du paquet mises sous forme normale de façon à ce qu'elles aient toutes le même membre gauche $r\, x_1 \ldots x_n$.

Un *programme sous forme clausale* est une suite de paquets de clauses définissant des prédicats distincts. Le programme représenté par cette suite de paquets de clauses est constitué de l'ensemble des définition de symboles de prédicats $r \in pred$ qui sont

1. soit représentés par un paquet de clauses définissant r,
2. soit de la forme $\forall x_1 \ldots \forall x_n \, (r\, x_1 \ldots x_n \;\equiv\; false)$, lorsque r est un symbole prédicat de qui n'est défini par aucun paquet de clauses.

Par exemple pour définir la somme des éléments d'une liste de trois éléments on introduira la définition du prédicat binaire sigma (écrite avec quelques virgules et parenthèses en plus et le pseudo-terme $a{+}b{+}c$),

$$\forall x \forall l \; (\text{sigma}(x, l) \;\equiv\; \exists a \exists b \exists c (x \sim a{+}b{+}c \wedge l = .(a., .(b, .c, [\,]))))$$

qui s'écrira sous forme clausale en Prolog IV

```
sigma(X,L) :- A ex B ex C ex
              X = A.+.B.+.C, L = [A,B,C].
```

Après avoir compilé ce programme on pourra poser la requête

```
>> sigma(X,.(1,.(2,.(3,[])))).
X = 6.
```

Si d'une façon générale on veut exprimer la relation qui lie une liste numérique avec la somme de ses éléments on écrira :

$$\forall l \forall x \left(\text{sigma}(x, l) \equiv \left(\begin{array}{c} l = [\,] \wedge x = 1 \\ \vee \\ \exists l' \exists y' \exists x' \left(\begin{array}{l} l = .(y', l') \\ \wedge \;\text{plus}(y', x', x) \\ \wedge \;\text{sigma}(x', l') \end{array} \right) \end{array} \right) \right) \tag{2.8}$$

Pour calculer la somme de 2,3 et 4 on résoudra alors la contrainte

$$\exists l \; (l = .(2, .(3, .(4, [\,]))) \;\wedge\; \text{sigma}(x, l))$$

Ceci amènera à concevoir le programme Prolog IV

```
sigma(X,L) :- L = [], X = 0;
              Lp ex Yp ex Xp ex
              L = .(Yp,Lp), plus(X,Yp,Xp), sigma(Xp,Lp).
```

qui peut aussi s'écrire

```
sigma(0,[]).
sigma(X,.(Yp,Lp)) :- plus(X,Yp,Xp), sigma(Xp,Lp).
```

et à lancer la requête

```
>> L ex L = .(2,.(3,.(4,[]))), sigma(X,L).
X = 9.
```

2.5.3 La machine Prolog IV

La définition 2.8 de la relation nommée sigma en fonction d'elle-même nécessite une mise au point.

Ce que l'on cherche à faire c'est de calculer les solutions d'une contrainte p dans la structure π_4 enrichie d'un ensemble de relations définies par un programme $\mathcal{P}$. Cette structure enrichie n'étant pas forcément unique, on ne

peut, à vrai dire, résoudre p dans «la structure enrichie». Il faut supposer qu'il existe une contrainte q telle que

1. q ne fasse intervenir que des symboles de π_4,
2. $\mathcal{P} \cup \mathcal{T}(\pi_4) \models p \equiv q$,

et résoudre q dans la structure π_4. On est ramené au problème évoqué à la page 79. On s'attachera donc à calculer une contrainte q qui satisfait les points 1 et 2 précédents et dont les solutions sont le plus «visibles» possible. Comme nous l'avons aussi mentionné, on ne pourra assurer que q soit la contrainte *false* si q n'a aucune solution dans π_4. On assurera que q soit la contrainte *false* si q n'a de solutions dans aucune des structures ϕ qui satisfonts les axiomes $\mathcal{T}(\pi_4)$, c'est-à-dire si $\mathcal{T}(\pi_4) \models q \equiv \mathit{false}$.

La façon dont q est calculée à partir de p est l'essence même du déroulement d'un programme P à partir de la requête p. Nous allons décrire ce déroulement à l'aide d'une machine abstraite dite *machine Prolog IV* formalisée à l'aide d'un ensemble de règles de réécriture de sous-formules en sous-formules permettant de transformer p en q.

Appelons *conjonction existentielle* une contrainte de la forme

$$\exists x_1 \ldots \exists x_k \, (a_1 \wedge \cdots \wedge a_l),$$

où les a_i sont des contraintes atomiques et appelons *contrainte résolue dans* $\mathcal{T}(\pi_4)$ une contrainte p qui ne fait pas intervenir d'autres symboles que ceux de π_4 et qui est obligatoirement la contrainte *false* si $\mathcal{T}(\pi_4) \models p \equiv \mathit{false}$.

Machine Prolog IV On se donne une contrainte p et un programme $\mathcal{P}$ et l'on veut calculer une contrainte q telle que :

1. $\mathcal{P} \cup \mathcal{T}(\pi_4) \models p \equiv q$,
2. q est de la forme $m_1 \vee \ldots \vee m_n \vee \mathit{false}$ avec n éventuellement nul,
3. chaque m_i est une conjonction existentielle résolue dans $\mathcal{T}(\pi_4)$.

On part de la contrainte

$$\langle \mathit{true} \rangle \wedge (p \vee \mathit{false}), \tag{2.9}$$

les chevrons servant uniquement à délimiter des occurrences de sous-formules plus importantes que d'autres. On applique autant de fois que possibles les règles de réécriture qui suivent sur l'occurrence la plus à gauche possible d'une expression de la forme $\langle m \rangle$. Si le processus s'arrête on obtient une formule de la forme

$$\langle m_1 \rangle \vee \cdots \vee \langle m_n \rangle \vee \langle \mathit{false} \rangle,$$

chaque m_i étant une conjonction existentielle. La contrainte $m_1 \vee \cdots \vee m_n \vee \mathit{false}$, est alors la contrainte recherchée q. Si le processus ne s'arrête pas c'est que le programmeur a posé une mauvaise requête ou écrit un mauvais programme !

Règles de réécriture

1	$\exists x\,\langle \mathit{false}\rangle$	$\Longrightarrow$	$\langle \mathit{false}\rangle$
2	$\langle \mathit{false}\rangle \vee p$	$\Longrightarrow$	p
3	$\langle \mathit{false}\rangle \wedge p$	$\Longrightarrow$	$\langle \mathit{false}\rangle$
4	$\exists x\,\langle m\rangle$	$\Longrightarrow$	$\langle \text{quasisolved}(\exists x\, m)\rangle$
5	$\exists x\,(\langle m\rangle \vee p)$	$\Longrightarrow$	$(\exists x\,\langle m\rangle) \vee (\exists x\, p)$
6	$(\langle m\rangle \vee p) \vee q$	$\Longrightarrow$	$\langle m\rangle \vee (p \vee q)$
7	$(\langle m\rangle \vee p) \wedge q$	$\Longrightarrow$	$(\langle m\rangle \wedge p) \vee (p \wedge q)$
8	$\langle m\rangle \wedge (p \vee q)$	$\Longrightarrow$	$(\langle m\rangle \wedge p) \vee (\langle m\rangle \wedge q)$
9	$\langle m\rangle \wedge (p \wedge q)$	$\Longrightarrow$	$(\langle m\rangle \wedge p) \wedge q$
10	$\langle m\rangle \wedge \exists x p(x)$	$\Longrightarrow$	$\exists x'(\langle m\rangle \wedge p(x')$
11	$\langle m\rangle \wedge a$	$\Longrightarrow$	$\langle \text{quasisolved}(m \wedge a)\rangle$
12	$\langle m\rangle \wedge r\, s_1 \ldots s_n$	$\Longrightarrow$	$\langle m\rangle \wedge \exists x_1 \ldots \exists x_n \left(\begin{pmatrix} x_1 \sim s_1 \\ \wedge \ldots \wedge \\ x_n \sim s_n \end{pmatrix} \wedge \text{body}(r\, x_1 \ldots x_n) \right)$

avec les conventions suivantes

- $p, p(x), q$ sont des contraintes quelconques et $p(x')$ est la contrainte $p(x)$ dans laquelle on a substitué la variable x' à chaque occurrence libre de la variable x,
- x, x' sont des variables telles que dans la règle 10, ni m, ni $p(x)$, n'ait d'occurrence libre de x'.
- a est une contrainte atomique ne faisant pas intervenir d'autres symboles que ceux de π_4,
- m est une conjonction existentielle résolue dans $\mathcal{T}(\pi_4)$ et distincte de *false*,
- quasisolved(r) est une conjonction existentielle résolue dans $\mathcal{T}(\pi_4)$ telle que
  $\mathcal{T}(\pi_4) \models r \equiv \text{quasisolved}(r)$,
- r est un symbole de prédicat, les s_i sont des termes ou des pseudo-termes et les x_i sont des variables ne figurant pas dans les s_i,
- body$(r\, x_1, \ldots, x_n)$ est une contrainte telle que, au nom des variables près, la proposition qui suit appartienne à $\mathcal{P}$,
  $\forall x_1 \ldots \forall x_n\ (r\, x_1, \ldots, x_n \equiv \text{body}(r\, x_1, \ldots, x_n))$.

Les règles 7 et 8 distribuent le connecteur $\wedge$ sur le connecteur $\vee$ pour obtenir une disjonction de conjonctions. C'est la façon la plus simpliste de résoudre une contrainte booléenne et la mise en œuvre de cette distribution échoit à la machinerie « backtrackante » que l'on retrouve dans toutes les implantations de Prolog. Par exemple la résolution de la contrainte $(x=1 \vee x=2 \vee x=3) \wedge (y=1 \vee y=2 \vee y=3)$ produit :

```
>> (X=1; X=2; X=3), (Y=1; Y=2; Y=3).
Y = 1, X = 1;
Y = 2, X = 1;
Y = 3, X = 1;
Y = 1, X = 2;
Y = 2, X = 2;
Y = 3, X = 2;
Y = 1, X = 3;
Y = 2, X = 3;
Y = 3, X = 3.
```

C'est l'occasion de rappeler que les variables sont ou commencent par des majuscules, que la virgule remplace le connecteur $\wedge$, le point virgule le connecteur $\vee$, et que le point marque la fin de la contrainte. Cette mise sous forme disjonctive est hautement combinatoire puisque la taille de la formule finale peut être une fonction exponentielle de la taille de la formule de départ. Bien entendu toute explosion combinatoire est de la responsabilité du programmeur !

Les règles 6 et 9 privilégient l'association gauche-droite pour les conjonctions et les disjonctions. Les règles 5 et 10 déplacent les quantificateurs aux bons niveaux en augmentant leur portée dans les conjonctions et en les distribuant sur les disjonctions. Ces règles de réécriture ont un effet relativement mineur qui peut être simulé en utilisant un codage standard pour les conjonctions, en renommant certaines variables et en notant celles qui ne sont pas quantifiées.

Les règles 1,2,3,4 et 11 simplifient la formule en faisant appel à un «solveur» dans la théorie $\mathcal{T}(\pi_4)$. Par exemple pour la contrainte $x > y \wedge (x=1 \vee x=2 \vee x=3) \wedge (y=1 \vee y=2 \vee y=3)$ on obtient :

```
>> gt(X,Y), (X=1; X=2; X=3), (Y=1; Y=2; Y=3).
Y = 1, X = 2;
Y = 1, X = 3;
Y = 2, X = 3.
```

et pour la contrainte $\exists x \exists y\ (z=[x,y] \wedge x>y \wedge (x=1 \vee x=2 \vee x=3) \wedge (y=1 \vee y=2 \vee y=3))$ on obtient :

```
>> X ex Y ex Z = [X,Y], gt(X,Y),
     (X =1; X=2; X=3), (Y=1; Y=2; Y=3).
Z = [1,1];
Z = [2,1];
Z = [3,1];
Z = [3,2];
Z = [3,3].
```

Enfin, à peu de choses près, la règle 12 remplace le membre gauche d'une définition de prédicat par son membre droit ainsi que le ferait l'expansion d'une macro-définition.

2.6 Exemples de programmes en Prolog IV

Pour conclure voici un ensemble d'exemples diversifiés de programme Prolog IV qui, on l'espère, donnera un bon aperçu des fonctionnalités du langage : le traitement de listes, le traitement d'entiers et de booléens, la résolution de contraintes générales sur les nombres réels et la résolution de contraintes linéaires.

2.6.1 Contraintes sur les listes

Parmi les contraintes Prolog IV portant sur les listes deux retiennent particulièrement l'attention : la concaténation et la sélection du ième élément d'une liste. Les 4 premiers exemples sont là pour illustrer ce point.

Traduction bidirectionnelle de nombres arabes en nombres romains

Ce premier exemple montre l'intérêt de disposer d'une contrainte de concaténation de liste, même avec des axiomes aussi pauvres que les axiomes 12 et 13 de la page 84. Il s'agit de traduire la notation décimale, en fait arabe, d'un entier en sa notation romaine et vice-versa. L'entier arabe est écrit sous forme d'une liste de chiffres et l'entier romain sous forme d'une liste de caractères pris dans 'I', 'V', 'X', 'L', 'C', 'D', 'M' et '?'. Ces caractères représentent respectivement 1, 5, 10, 50, 100, 500, 1000 et 5000. Conformément à la norme ISO ils sont représentés par des identificateurs d'une lettre écrits entre apostrophes. Voici le programme.

```
araberomain(U,W) :-
    tr(U,['?','M','D','C','L','X','V','I'], W).

tr([], S, []).
tr(U o [0], S o [V,I], W) :-
                  dif([0] o U, U o [0]), tr(U, S, W).
tr(U o [1], S o [V,I], W o [I]) :-       tr(U, S, W).
tr(U o [2], S o [V,I], W o [I,I]) :-     tr(U, S, W).
tr(U o [3], S o [V,I], W o [I,I,I]) :-   tr(U, S, W).
tr(U o [4], S o [V,I], W o [I,V]) :-     tr(U, S, W).
tr(U o [5], S o [V,I], W o [V]) :-       tr(U, S, W).
tr(U o [6], S o [V,I], W o [V,I]) :-     tr(U, S, W).
tr(U o [7], S o [V,I], W o [V,I,I]) :-   tr(U, S, W).
tr(U o [8], S o [V,I], W o [V,I,I,I]) :- tr(U, S, W).
tr(U o [9], S o [X,V,I], W o [I,X]) :-   tr(U, S o [X], W).
```

Rappelons que la notation `X o Y` remplace `conc(X,Y)`. Les requêtes qui suivent traduisent des nombres de l'arabe vers le romain puis les retraduisent en arabe.

```
>> araberomain([1,2,3,4],W), araberomain(U,W).
U = [1,2,3,4],
W = ['M','C','C','X','X','X','I','V'].

>> araberomain([5,6,7,8],W), araberomain(U,W).
U = [5,6,7,8],
W = [?,'D','C','L','X','X','V','I','I','I'].

>> araberomain([9,9,9],W), araberomain(U,W).
U = [9,9,9],
W = ['C','M','X','C','I','X'].
```

Problème de coloration de Shur

Dans une concaténation, la taille de la liste obtenue est la somme des tailles des listes concaténées. Ce deuxième exemple illustre le début d'arithmétique caché dans ce fait. Il concerne un problème de coloration liée à une propriété connue sous le nom de lemme de I. Shur.

On se donne m couleurs et un entier positif n. Il s'agit d'attribuer à chaque entier i de la suite finie $1, 2, \ldots, n$, une couleur x_i prise parmi les m couleurs de façon à ce que pour tout i, j, k pris entre 1 et n on ait

$i + j = k$ entraîne les couleurs x_i, x_j, x_k ne sont pas toutes identiques.

I. Shur a montré que quel que soit m il existe toujours un entier n suffisamment grand pour lequel ce problème n'a pas de solution.

Intéressons nous au cas particulier où $m = 3$, les couleurs étant a, b, c. Le coloriage $[x_1, \ldots, x_n]$ se définit récursivement sur n par :

```
coloriage([]).
coloriage(X o [C]):-
   coloriage(X),
   ajoutcorrect(X, C),
   couleur(C).

ajoutcorrect([], C).
ajoutcorrect([A], C) :-
   dif(A, C).
ajoutcorrect([A] o X o [B], C) :-
   ajoutcorrect(X, C),
   dif([A,B], [C,C]).

couleur(a).
couleur(b).
couleur(c).
```

Pour $n = 13$, on obtient les 18 coloriages :

```
>> size(X) = 13, coloriage(X).
X = [a,b,b,a,c,c,a,c,c,a,b,b,a];
X = [a,b,b,a,c,c,b,c,c,a,b,b,a];
X = [a,b,b,a,c,c,c,c,c,a,b,b,a];
X = [a,c,c,a,b,b,a,b,b,a,c,c,a];
X = [a,c,c,a,b,b,b,b,b,a,c,c,a];
X = [a,c,c,a,b,b,c,b,b,a,c,c,a];
X = [b,a,a,b,c,c,a,c,c,b,a,a,b];
X = [b,a,a,b,c,c,b,c,c,b,a,a,b];
X = [b,a,a,b,c,c,c,c,c,b,a,a,b];
X = [b,c,c,b,a,a,a,a,a,b,c,c,b];
X = [b,c,c,b,a,a,b,a,a,b,c,c,b];
X = [b,c,c,b,a,a,c,a,a,b,c,c,b];
X = [c,a,a,c,b,b,a,b,b,c,a,a,c];
X = [c,a,a,c,b,b,b,b,b,c,a,a,c];
X = [c,a,a,c,b,b,c,b,b,c,a,a,c];
X = [c,b,b,c,a,a,a,a,a,c,b,b,c];
X = [c,b,b,c,a,a,b,a,a,c,b,b,c];
X = [c,b,b,c,a,a,c,a,a,c,b,b,c].
```

et à partir de $n = 14$ on n'obtient plus de coloriages :

```
>> size(X) = 14, coloriage(X).
false.
```

Transposition de matrice par concaténations

Ecrire un programme de transposition de matrice en Prolog est un exercice plus difficile qu'il ne semble à première vue. La matrice se représente naturellement comme un liste de ligne, chaque ligne étant une liste de nombres. La difficulté provient du fait qu'il faut aussi pouvoir accéder aux colonnes. Pour ceci on prévoit un prédicat `colonneun/3` qui permet d'obtenir la première colonne et le reste de la matrice amputée de celle-ci. Il faut aussi pouvoir parler de la matrice formée d'une liste de lignes toutes vides. C'est là qu'intervient le prédicat `zerocolonnes/1` qui utilise une contrainte de concaténation produisant une liste d'éléments identiques, ici des listes vides. Par symétrie nous avons aussi introduit les prédicats plus faciles `ligneun/3` et `zerolignes/1`. Voici le programme :

```
transposition(A,B) :-
   zerolignes(A),
   zerocolonnes(B).
transposition(A,B) :-
   ligneun(A,X,C),
   colonneun(B,X,D),
   transposition(C,D).

zerolignes([]).

zerocolonnes(B) :-
   conc([[]],B) = conc(B,[[]]).

ligneun([X|A],X,A).

colonneun([],[],[]).
colonneun([[Aaa|Aa]|A],[Aaa|X],[Aa|B]) :-
   colonneun(A,X,B).
```

et voici une requête qui transpose une matrice et la retranspose pour obtenir la matrice d'origine.

```
>> transposition([[1,3,5],[2,4,6]],B),
    transposition(B,A).
A = [[1,3,5],[2,4,6]],
B = [[1,2],[3,4],[5,6]].
```

Transposition de matrice par indiçages

Qui pense à la transposition d'une matrice pense à la formule $a_{ij} = b_{ji}$. Voici un deuxième programme de transposition reposant sur cette formule :

```
transpositionbis(A,B) :-
   M = size(A),
   contraintes(M,c0,[A,N]),
   N = size(B),
   contraintes(N,c0,[B,M]),
   contraintes(M,c1,[A,B]).

contraintes(0,A,L).
contraintes(I,A,L) :-
   ge(I,1),
   contrainte(A,[I|L]),
   contraintes(I.-.1,A,L).

contrainte(c0,[I,A,N]) :-
   N = size(A:I).
contrainte(c1,[I,A,B]) :-
   N = size(A:I),
   contraintes(N,c2,[I,A,B]).
contrainte(c2,[J,I,A,B]) :-
   A:I:J = B:J:I.
```

L'écriture `A:I:J` est interprétée comme `((A:I):J)` qui elle-même est interprétée comme le pseudo-terme `index(index(A,I),J)`.

```
>> transpositionbis([[1,2],[3,4],[5,6]],B),
    transpositionbis(B,A).
A = [[1,2],[3,4],[5,6]],
B = [[1,3,5],[2,4,6]].
```

2.6.2 Contraintes sur les entiers et les booléens

Dans de nombreux problèmes combinatoires les entiers et les booléens, en fait les entiers 0 et 1, jouent un rôle de premier plan. En voici la preuve à travers quelques exemples.

Enumération d'entiers naturels

Ce programme énumère, sans en oublier aucune, toutes les listes d'entiers naturels, c'est-à-dire d'entiers positifs ou nuls. Cet ensemble de liste étant infini, le programme ne s'arrête que s'il est appelé dans un contexte qui borne ces entiers. L'énumération est accélérée par les conflits entre les nombreuses contraintes posées et ce contexte. Voici le programme :

```
enumeration(L) :- nonnegatifs(L), depuis(0,L).

nonnegatifs([]).
nonnegatifs([N|L]) :-
   int(N),
   ge(N,0),
   nonnegatifs(L).

depuis(N,[]).
depuis(N,La) :-
   dif(La,[]),
   superieurs(N,La,Lb),
   depuis(N.+.1,Lb).

superieurs(N,[],[]).
superieurs(N,[N|La],Lb) :-
   superieurs(N,La,Lb).
superieurs(N,[Na|La],[Na|Lb]) :-
   lt(N,Na),
   superieurs(N,La,Lb).
```

et voici une énumération sous contexte :

```
>> X ~ cc(2,5), Y ~ cc(1,4), Z ~ cc(3,6),
   Z = plus(X,Y), enumeration([X,Y,Z]).
Z = 3, Y = 1, X = 2;
Z = 4, Y = 1, X = 3;
Z = 5, Y = 1, X = 4;
Z = 6, Y = 1, X = 5;
Z = 4, Y = 2, X = 2;
Z = 5, Y = 3, X = 2;
Z = 6, Y = 4, X = 2;
Z = 5, Y = 2, X = 3;
Z = 6, Y = 2, X = 4;
Z = 6, Y = 3, X = 3.
```

Pour information voici un autre programme d'énumération, qui peut être plus efficace, mais qui fait appel à deux meta-prédicats prédéfinis `glb/2` et `int_size/2` décrits dans la partie «prédicats prédéfinis Prolog IV» et au prédicat prédéfini ISO de coupure d'espace de recherche. Dans ce programme on suppose que les éléments à énumérer ont été préalablement contraints à être des entiers.

```
metaenumeration([]).
metaenumeration([Xa|X]) :-
   pluspetit([Xa|X],Xa,Y,Ya),
   glb(Ya,M),
   suite(Ya,M),
   metaenumeration(Y).

suite(Ya,Ya).
suite(Ya,M) :- gt(Ya,M).

pluspetit([Xa|X],Xb,Y,Ya) :-
   int_size(Xa,1),!,
   pluspetit(X,Xb,Y,Ya).
pluspetit([Xa|X],Xb,[Xa|Y],Yb) :-
   glb(Xa,M),
   glb(Xb,N),
   lt(M,N),!,
   pluspetit(X,Xa,Y,Yb).
pluspetit([Xa|X],Xb,[Xa|Y],Yb) :-
   pluspetit(X,Xb,Y,Yb).
pluspetit([],Xb,[],Xb).
```

```
>> X ~ int, Y ~ int, Z ~ int,
   X ~ cc(2,5), Y ~ cc(1,4), Z ~ cc(3,6),
   Z = plus(X,Y), metaenumeration([X,Y,Z]).
Z = 3, Y = 1, X = 2;
Z = 4, Y = 1, X = 3;
Z = 5, Y = 1, X = 4;
Z = 6, Y = 1, X = 5;
Z = 4, Y = 2, X = 2;
Z = 5, Y = 3, X = 2;
Z = 6, Y = 4, X = 2;
Z = 5, Y = 2, X = 3;
Z = 6, Y = 2, X = 4;
Z = 6, Y = 3, X = 3.
```

Suite magique

Etant donné n, on s'intéresse à trouver les suites finies x de la forme $(x_0,\ldots,x_{n-1})$ telles que chaque x_i soit le nombre d'occurrences de l'entier i dans x. Les solutions sont les solutions de la conjonction $P_1(x)\wedge\cdots\wedge P_n(x)$ de contraintes

$$P_i(x):\quad x_{i-1}=\sum_{j=0}^{n-1}(x_j=i-1),$$

où l'expression $(x_j=i-1)$ vaut 1 ou 0 suivant que l'égalité représentée est ou n'est pas vérifiée. Voici le programme qui fait appel au programme précédent d'énumération d'entiers :

```
suitemagique(X) :-
   npremierescontraintes(size(X),X),
   enumeration(X).

npremierescontraintes(0,X).
npremierescontraintes(I,X) :-
   gt(I,0),
   J = I.-.1,
   nboccurrences(J,X,X:I),
   npremierescontraintes(J,X).

nboccurrences(A,[],0).
nboccurrences(A,[B|X],M) :-
   M = N.+.beq(A,B),
   nboccurrences(A,X,N).
```

et voici quelques calculs de solutions :

```
>> size(X) = 4, suitemagique(X).
X = [2,0,2,0];
X = [1,2,1,0].
>> size(X) = 5, suitemagique(X).
X = [2,1,2,0,0].
>> size(X) = 6, suitemagique(X).
false.
>> size(X) = 10, suitemagique(X).
X = [6,2,1,0,0,0,1,0,0,0].
```

Problème des 21 carrés connus

On se donne n carrés de dimensions $a_1 \times a_1, \ldots, a_n \times a_n$ et on se propose de les placer dans un carré de dimensions $m \times m$. Les données sont $n = 21$, $m = 112$ et le n-uplet $(a_1, \ldots, a_n)$ est égal à

$$(550, 42, 37, 35, 33, 29, 27, 25, 24, 19, 18, 17, 16, 15, 11, 9, 8, 7, 6, 4, 2).$$

Les solutions sont les solutions de toutes les contraintes qui suivent. La contrainte 3 élimine l'essentiel des symétries en contraignant le premier carré à avoir son coin inférieur gauche à l'intérieur du quart inférieur gauche du grand carré et sous la diagonale positive de ce quart. Les contraintes de la forme 4a et 4b sont redondantes mais sont fondamentales pour augmenter la propagation d'informations. L'esprit général de ce jeu de contraintes respectent les idées exposées dans un papier de N. Beldiceanu.

1a	entier$(x_j) \wedge x_j \in [0, m - a_j]$,	pour $j = 1, \ldots, n$,
1b	entier$(y_j) \wedge y_i \in [0, m - a_j]$,	pour $j = 1, \ldots, n$,
2	$(x_j + a_j \leq x_k \vee x_k + a_k \leq x_j) \vee$ $(y_j + a_j \leq y_k \vee y_k + a_k \leq y_j)$,	pour $j = 1, \ldots, n$, pour $k = j + 1, \ldots, n$,
3	$x_j \leq \frac{1}{2}(m - a_j) \ \wedge \ y_j \leq x_j$	pour $j = 1$,
4a	$m = \sum_{j=1}^{n} a_j \times (i \in (x_j, x_j + a_j])$,	pour $i = 1, \ldots, m$,
4b	$m = \sum_{j=1}^{n} a_j \times (i \in (y_j, y_j + a_j])$,	pour $i = 1, \ldots, m$

Voici le programme qui fait appel à nos programmes précédents de transposition et d'énumération d'entiers.

```
remplissage(F) :-
   M = 112,
   N = size(A),
   A = [50,42,37,35,33,29,27,25,24,
       19,18,17,16,15,11,9,8,7,6,4,2],
   contraintes(N,c1,[X,A,M]),
   contraintes(N,c1,[Y,A,M]),
   contraintes(N,c2,[X,Y,A]),
   contraintes(1,c3,[X,Y,A,M]),
   contraintes(M,c4,[X,A,M]),
   contraintes(M,c4,[Y,A,M]),
   transposition([X,Y,A],F),
   enumeration(X),
   enumeration(Y).

% Creation de conjunctions de contraintes

contraintes(0,C,L).
contraintes(I,C,L) :-
   ge(I,1),
   contrainte(C,[I|L]),
   contraintes(I.-.1,C,L).

% Les différents types de contraintes

contrainte(c1,[I,X,A,M]) :-
   Xi = index(X,I), Ai = index(A,I),
   Xi ~ int, Xi ~ cc(0,M.-.Ai).

contrainte(c2,[J,X,Y,A]) :-
   contraintes(minus(J,1),c2bis,[J,X,Y,A]).
contrainte(c2bis,[K,J,X,Y,A]) :-
   Xj = index(X,J), Yj = index(Y,J), Aj = index(A,J),
   Xk = index(X,K), Yk = index(Y,K), Ak = index(A,K),
   or(boutoo(Xk.-.Xj,.-.Ak,Aj),boutoo(Yk.-.Yj,.-.Ak,Aj)).

contrainte(c3,[J,X,Y,A,M]) :-
   Xj = index(X,J), Yj = index(Y,J), Aj = index(A,J),
   le(Xj,(M.-.Aj)./.2), le(Yj,Xj).

contrainte(c4,[I,[],[],0]).
contrainte(c4,[I,[Xa|X],[Aa|A],Ma.+.M]) :-
   Ma = boc(I,Xa,Xa.+.Aa).*.Aa,
   contrainte(c4,[I,X,A,M]).
```

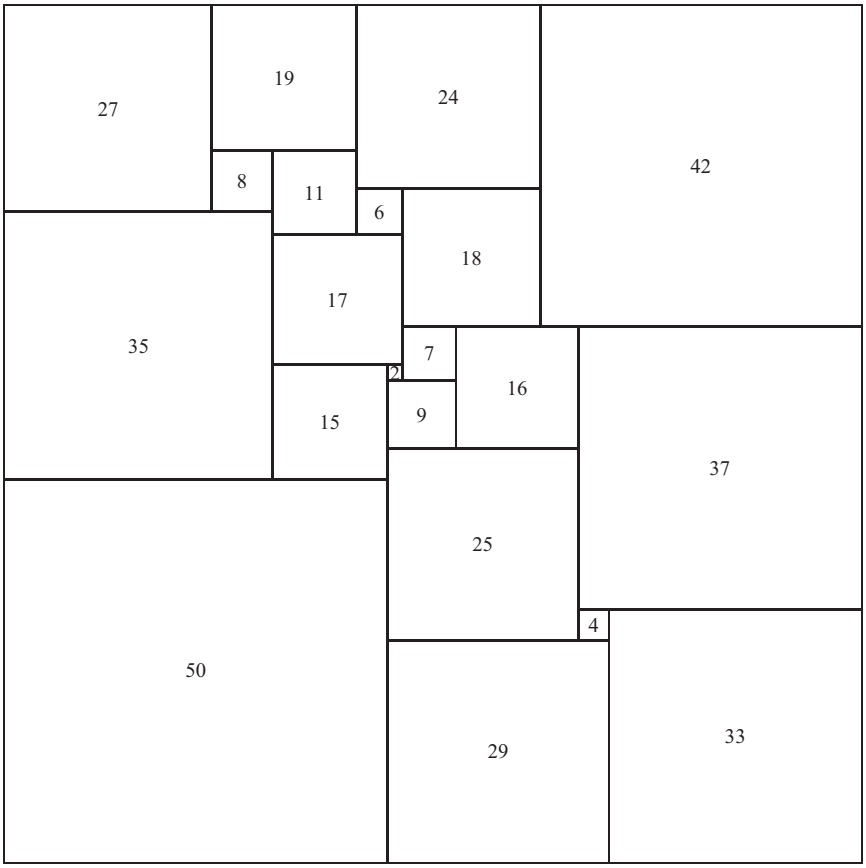
Attention pour exécuter la requête qui suit il faut avoir lancé Prolog IV avec en paramètres

```
-heap 4000000 -local 40000 -choice 200000
```

Pour plus d'information sur ces paramètres, voir la partie « environnement général » du manuel.

```
>> remplissage(F).

F = [[0,0,50],[70,70,42],[75,33,37],[0,50,35],
[79,0,33],[50,0,29],[0,85,27],[50,29,25],
[46,88,24],[27,93,19],[52,70,18],[35,65,17],
[59,54,16],[35,50,15],[35,82,11],[50,54,9],
[27,85,8],[52,63,7],[46,82,6],[75,29,4],
[50,63,2]]

F = [[0,0,50],[70,70,42],[33,75,37],[50,0,35],
[0,79,33],[0,50,29],[85,0,27],[29,50,25],
[88,46,24],[93,27,19],[70,52,18],[65,35,17],
[54,59,16],[50,35,15],[82,35,11],[54,50,9],
[85,27,8],[63,52,7],[82,46,6],[29,75,4],
[63,50,2]]
```

On obtient deux solutions symétriques qui se résument au dessin qui suit.



2.6.3 Contraintes sur les réels

Passons au traitement du continu en isolant des nombres algébriques.

Racines des polynomes de Wilkinson

L'exemple qui suit est un benchmark pour mettre à rude épreuve le calcul des racines d'un polynôme, surtout si ce calcul est fait avec des nombres binaires flottants IEEE simple précision. On considère le polynome dit de Wilkinson

$W(x) : (x+1)(x+2)\ldots(x+20)$

qui bien entendu admet 20 racines. Si on l'altère légèrement en considérant le polynôme

$W'(x) : \mathrm{W}(x) + 2^{-23}x^{19}$

on ne doit plus que trouver 10 racines. Voyons si Prolog IV est à la hauteur.

```
wilkinson(Y, X) :-
   produitmonomes(Y, X, 20).

produitmonomes(1, X, 0).
produitmonomes(Y.*.(X.+.N), X, N) :-
   ge(N, 1),
   produitmonomes(Y, X, N.-.1).

wilkinsonbis(Y.+.power(1/2,23).*.power(X,19), X) :-
  wilkinson(Y, X).
```

On fait alors usage du prédicat prédéfini `realsplit1/` décrit dans la partie « prédicats prédéfinis de Prolog IV » du manuel. On onbtient les 20 racines par :

```
>> wilkinson(0, X), realsplit([X]).
X = -20;
X = -19;
X = -18;
X = -17;
X = -16;
X = -15;
X = -14;
X = -13;
X = -12;
X = -11;
X = -10;
X = -9;
X = -8;
X = -7;
X = -6;
X = -5;
X = -4;
X = -3;
X = -2;
X = -1.
```

Dans le cas du polynôme altéré on obtient une multitude de racines apparemment parasites, mais si on limite x à être dans l'intervalle $[-100, 100]$, on trouve bien 10 encadrements :

```
>> X ~ cc(-100,100), wilkinsonbis(0, X),
       realsplit([X]).
X ~ cc(-`>20.846977`,-`<20.84688`);
X ~ cc(-`>8.917252`,-`>8.917247`);
X ~ oo(-`>8.007267`,-`<8.007267`);
X ~ oo(-`<6.999698`,-`>6.999697`);
X ~ cc(-`>6.000007`,-`<6.000007`);
X ~ oo(-5,-`>4.9999995`);
X ~ oo(-`>4.0000004`,-4);
X ~ oo(-3,-`>2.9999997`);
X ~ oo(-`>2.0000002`,-2);
X ~ oo(-1,-`>0.9999999`).
```

Racines confirmées du polynome altéré de Wilkinson

Les 10 encadrements précédents garantissent qu'en dehors de ceux-ci il n'y a pas de racines du polynôme altéré mais ne garantissent pas que dans chaque encadrement il y a en exactement une. Il pourrait n'y en avoir aucune ou il pourrait y en avoir plusieurs. Pour lever ce doute Michel Van Caneghem vérifie que dans chaque encadrement

1. la dérivée du polynôme altéré ne s'annule pas et
2. les valeurs du polynôme altéré, prises à ses bornes, sont de signes contraires.

Il est donc nécessaire de programmer le calcul de la dérivée du polynôme altéré par le prédicat `dwilkinsonbis/2`.

```
dwilkinsonbis(Y.+.19.*.power(1/2,23).*.power(X,18), X) :-
   dwilkinson(Y, X).

dwilkinson(Y, X) :-
   dproduitmonomes(Y, X, 20).

dproduitmonomes(0, X, 0).
dproduitmonomes(Y.+.Z.*.(X.-.N), X, N) :-
   ge(N, 1),
   produitmonomes(Y, X, N.-.1),
   dproduitmonomes(Z, X, N.-.1).
```

On peut alors lancer une requête plus élaborée, qui chaque fois qu'en sortie la variable `Certain` prend la valeur `oui`, garantit l'existence et l'unicité de la racine dans son encadrement.

```
>> X ~ cc(-100,100),
   wilkinsonbis(0, X),
   realsplit([X]),
   Epsilon ex Xa ex Xb ex Xc ex Ya ex Yb ex Yc ex
   Epsilon = 1./.1000,
   Xa = X.-.Epsilon,
   Xc = X.+.Epsilon,
   Xb ~ cc(Xa, Xc),
   wilkinsonbis(Ya, Xa),
   wilkinsonbis(Yc, Xc),
   dwilkinsonbis(Yb, Xb),
   Certain = if(blt(Ya.*.Yc,0).*.bdif([Yb],[0]),oui,non).
Certain = oui, X ~ cc(-`>20.846977`,-`<20.84688`);
Certain = oui, X ~ cc(-`>8.917252`,-`>8.917247`);
Certain = oui, X ~ oo(-`>8.007267`,-`<8.007267`);
Certain = oui, X ~ oo(-`<6.999698`,-`>6.999697`);
Certain = oui, X ~ cc(-`>6.000007`,-`<6.000007`);
Certain = oui, X ~ oo(-5,-`>4.9999995`);
Certain = oui, X ~ oo(-`>4.0000004`,-4);
Certain = oui, X ~ oo(-3,-`>2.9999997`);
Certain = oui, X ~ oo(-`>2.0000002`,-2);
Certain = oui, X ~ oo(-1,-`>0.9999999`).
```

2.6.4 Contraintes linéaires

Les deux exemples qui suivent rappellent qu'en Prolog IV on dispose des mêmes contraintes linéaires qu'en Prolog III à condition d'utiliser d'utiliser des pseudo-opérations et des relations dont les noms se terminent par `lin`.

Produit matriciel

Le premier exemple est la quintessence du linéaire sans relation d'ordre. Il s'agit d'exprimer la contrainte $A \times B = C$, où A, B, C sont des matrices dimensionnées correctement pour la multiplication matricielle, deux d'entre elles au moins étant de dimensions connues. Voici le programme :

```
matricefoismatrice(A,B,C) :-
   zerolignes(A),
   zerolignes(C).
matricefoismatrice(A,B,C) :-
   ligneun(A,X,Ap),
   ligneun(C,Y,Cp),
   size(X) = size(B),
   vecteurfoismatrice(X,B,Y),
   matricefoismatrice(Ap,B,Cp).

vecteurfoismatrice(X,B,[]) :-
   zerocolonnes(B).
vecteurfoismatrice(X,B,[R|Y]) :-
   colonneun(B,Z,Bp),
   vecteurfoisvecteur(X,Z,R),
   vecteurfoismatrice(X,Bp,Y).

vecteurfoisvecteur([],[],0).
vecteurfoisvecteur([R|X],[S|Y],R*S+T) :-
   vecteurfoisvecteur(X,Y,T).

zerolignes([]).

zerocolonnes(B) :-
   conc([[]],B) = conc(B,[[]]).

ligneun([X|A],X,A).

colonneun([],[],[]).
colonneun([[Aaa|Aa]|A],[Aaa|X],[Aa|B]) :-
   colonneun(A,X,B).
```

Les notations `R*S` et `R+T` correspondent aux pseudo-termes `timeslin(R,S)` et `pluslin(R,T)`. On peut alors lancer la requête :

```
>> A ex B ex C ex
   A = [[1,2],[3,4],[5,6]],
   B = [[1,3,5],[2,4,6]],
   C = [[5,11,17],[11,25,39],[17,39,61]],
   matricefoismatrice(Ap,B,C),
   matricefoismatrice(A,Bp,C),
   matricefoismatrice(A,B,Cp).

Cp = [[5,11,17],[11,25,39],[17,39,61]],
Bp = [[1,3,5],[2,4,6]],
Ap = [[1,2],[3,4],[5,6]].
```

Problème des 9 carrés inconnus

Le deuxième exemple a fait le succès de Prolog III[8]. Il s'agit d'assembler n carrés de tailles ditinctes pour former un rectangle. Les tailles des carrés et les tailles du rectangle sont inconnues. Seul l'entier n est connu et vaut 9. Voici la transposition du programme Prolog III en Prolog IV. C'est un bon exercice de syntaxe.

```
rectangleRempli(A, C) :-
   gelin(A,1),
   carresDistincts(C),
   zoneRemplie([-1,A,1], L, C, []).

carresDistincts([]).
carresDistincts([B|C]) :-
   gtlin(B,0),
   carresDistincts(C),
   horsDe(B, C).

horsDe(B, []).
horsDe(B, [Bp|C]) :-
   dif(B, Bp),
   horsDe(B, C).

zoneRemplie([V|L], [V|L], C, C) :-
   gelin(V, 0).
zoneRemplie([V|L], Lppp, [B|C], Cpp) :-
   lt(V, 0),
   carrePlace(B, L, Lp),
   zoneRemplie(Lp, Lpp, C, Cp),
   zoneRemplie([V+B,B|Lpp], Lppp, Cp, Cpp).

carrePlace(B, [H,0,Hp|L], Lp) :-
   gtlin(B, H),
   carrePlace(B, [H+Hp|L], Lp).
carrePlace(B, [H,V|L], [-B+V|L]) :-
   B = H.
carrePlace(B, [H|L], [-B,H-B|L]) :-
   ltlin(B, H).
```

Et voici les 8 solutions du problème, qui en fait sont au nombre de 2 si on tient compte des symétries. Ici `A` est la longueur du rectangle trouvé, sa largeur étant supposée valoir 1, et `C` est la liste des tailles des 9 carrés.

8. Alain Colmerauer. An Introduction to Prolog III, Communications of the ACM, 33(7):68-90, 1990.

```
>> size(C)=9, rectangleRempli(A, C).
A = 33/32,
C = [15/32,9/16,1/4,7/32,1/8,7/16,1/32,5/16,9/32];
A = 69/61,
C = [33/61,36/61,28/61,5/61,2/61,9/61,25/61,7/61,16/61];
A = 33/32,
C = [9/16,15/32,7/32,1/4,7/16,1/8,5/16,1/32,9/32];
A = 69/61,
C = [36/61,33/61,5/61,28/61,25/61,9/61,2/61,7/61,16/61];
A = 33/32,
C = [9/32,5/16,7/16,1/4,1/32,7/32,1/8,9/16,15/32];
A = 69/61,
C = [28/61,16/61,25/61,7/61,9/61,5/61,2/61,36/61,33/61];
A = 69/61,
C = [25/61,16/61,28/61,9/61,7/61,2/61,5/61,36/61,33/61];
A = 33/32,
C = [7/16,5/16,9/32,1/32,1/4,1/8,7/32,9/16,15/32].
```

3

Relations Prolog IV

3.1 Introduction

L'OBJET DU PRÉSENT CHAPITRE est de donner une description opérationnelle de l'interprétation par Prolog IV des divers symboles de relations réservés introduits dans les concepts de base.

Veuillez lire avec attention la présente introduction, qui rappelle quelques concepts de base importants pour une bonne compréhension du fonctionnement de Prolog IV, ainsi que les conventions et notations utilisées dans le reste du chapitre.

3.1.1 Rappels des concepts de base

Sous-domaines privilégiés

Un des concepts importants de Prolog IV est la définition de ses *sous-domaines privilégiés*. Ces sous-domaines privilégiés sont en fait des sous-ensembles d'ensembles d'arbres. La notion de sous-domaine est donc très similaire à la notion de type [1].

Bien entendu, Prolog IV n'est pas un langage fortement typé, c'est-à-dire que ses variables ne sont pas typées, et aucune vérification de typage n'est effectuée lors de la compilation. Toutefois, le fait qu'une variable soit contrainte à appartenir à un sous-domaine privilégié [2] permet souvent de déclencher certains traitements.

Les formes des sous-domaines privilégiés de Prolog IV sont les suivantes (les définitions proviennent du chapitre sur les concepts de base) :

1. l'ensemble de tous les arbres,
2. l'ensemble des arbres finis,
3. l'ensemble des arbres infinis,
4. l'ensemble des listes,

1. D'un point de vue théorique, on peut noter que l'ensemble de ces sous-domaines privilégiés forme un treillis de types.
2. Ce qui peut s'apparenter à un typage de la variable.

5. l'ensemble des arbres qui ne sont pas des listes,
6. pour chaque entier positif n, l'ensemble des listes de taille n,
7. pour chaque n-uplet $A_1, \ldots, A_n$ d'intervalles IEEE, l'ensemble des liste $[a_1, \ldots, a_n]$ avec $a_i \in A_i$,
8. l'ensemble des arbres d'un seul nœud,
9. l'ensemble des arbres de plus d'un nœud,
10. l'ensemble des identificateurs,
11. l'ensemble des arbres qui ne sont pas un identificateur,
12. pour chaque intervalle IEEE A, l'ensemble A,
13. l'ensemble des arbres qui ne sont pas un nombre réel,
14. pour chaque arbre doublement rationnel a, l'ensemble $\{a\}$,
15. l'ensemble vide.

Cette liste appelle un certain nombre de définitions et remarques :

– Un intervalle IEEE est un intervalle dont les bornes, si elles existent, sont des nombres flottants de la norme IEEE 754 (ou rationnels IEEE).

– Un arbre doublement rationnel est un arbre rationnel (dont le nombre de sous-arbres est fini), dont aucun nœud n'est étiqueté par un nombre réel non rationnel. En fait, c'est un arbre qui peut être décrit exactement avec Prolog IV.

– Il est important de noter l'absence de certains ensembles de cette liste. En particulier, l'ensemble des nombres entiers n'est pas un sous-domaine privilégié. On note également qu'il n'existe pas de sous-domaines correspondant aux ensembles du type «toutes les listes dont les éléments sont des identificateurs». Ces remarques sont importantes pour comprendre les algorithmes de résolution de Prolog IV.

Quand Prolog IV est en mode «unions d'intervalles», il faut remplacer dans les définitions des sous-domaines 7 et 12 la locution «intervalles IEEE» par la locution «unions d'intervalles IEEE».

Approximation

Dans la résolution des contraintes, Prolog IV utilise des approximations, basées sur les sous-domaines privilégiés définis ci-dessus. De plus, cette résolution consiste d'un point de vue théorique à l'application d'un nombre restreint d'axiomes. Toutes les réponses de Prolog IV à des requêtes peuvent être expliquées en fonction des définitions des sous-domaines privilégiés et des axiomes. Toutefois, il n'est pas toujours très simple d'arriver à ces explications. Nous avons essayé d'expliquer tout au long du présent chapitre certains des cas les plus classiques.

Ici, nous allons simplement voir quelques cas d'approximations. Nous utiliserons ici des pseudo-termes simples, que nous expliquerons de manière informelle par rapport aux divers types de sous-domaines privilégiés.

– `X \~ 1 .`
On a ici un cas très simple, définissant le singleton $\{1\}$, qui peut être

approximé par lui-même (formes 12 ou 14).

- `X \~ cc(1,2) .`
  On définit ici l'intervalle $[1, 2]$, qui peut être approximé par lui-même (forme 12).

- `X ~ cc(0,1/3) .`
  On définit ici l'intervalle $[0, \frac{1}{3}]$, qui n'est pas un sous-domaine privilégié, puisque $\frac{1}{3}$ n'est pas un rationnel IEEE. Son approximation est donc le plus petit intervalle IEEE le contenant.

- `X ~ [1,cc(1,2),real] .`
  On a ici une liste $[a, b, c]$, avec $a = 1$, $b \in [1, 2]$ et $c \in \mathbf{R}$. Les domaines de a, b et c peuvent être approximés exactement par des intervalles IEEE (le type 12 est applicable dans les trois cas). De ce fait, la liste $[a, b, c]$ est donc un sous-domaine privilégié de type 7.

- `X ~ [1,cc(1,2),Y] .`
  On a ici une liste $[a, b, c]$, avec $a = 1$, $b \in [1, 2]$ et c un arbre quelconque. Puisque c ne peut pas être approximé par un intervalle, la liste $[a, b, c]$ n'est pas un sous-domaine privilégié de type 7, mais seulement un sous-domaine privilégié de type 6, c'est-à-dire une liste de trois éléments (où les éléments ne sont pas qualifiés).

Passons maintenant à des pseudo-termes et contraintes un peu plus complexes :

- `X \~ oo(0,1/3) n oo(1/3,1) .`
  On définit ici l'intervalle $(0, \frac{1}{3}) \cap (\frac{1}{3}, 1)$, soit l'ensemble vide (les deux intervalles sont disjoints). Ici, Prolog IV effectue l'approximation des deux intervalles séparément, et les «élargit» donc un peu autour de la valeur $\frac{1}{3}$. Ensuite, en en prenant l'intersection, on obtient un intervalle non vide (en fait, le plus petit intervalle IEEE contenant la valeur $\frac{1}{3}$.

- `X \~ binlist(1,[1,cc(1,2),Y]) .`
  On a ici défini un booléen qui doit prendre la valeur 1 (vrai) si la valeur 1 appartient à la liste $[a, b, c]$, avec $a = 1$, $b \in [1, 2]$ et c un arbre quelconque. Intuituvement, on voit que cette condition est effectivement vérifiée. Or, Prolog IV est incapable d'atteindre cette conclusion. En fait, comme on l'a vu précédemment, l'ensemble dénoté par la liste $[a, b, c]$ n'est pas un sous-domaine privilégié, et il est approximé par l'ensemble des listes de 3 éléments. Avec cette approximation, il est impossible de conclure sur l'appartenance. Pour pouvoir conclure, il suffit que la variable `Y` du pseudo-terme soit contrainte à prendre une valeur réelle. La liste $[a, b, c]$ sera alors un sous-domaine privilégié de type 7, permettant ainsi à Prolog IV de déterminer la soution avec plus de précision.

Nous avons ici essayé de présenter quelques cas extrêmes, afin de montrer qu'il faut toujours garder à l'esprit les problèmes d'approximation. Toutefois, ces problèmes surgissent dans des cas relativement rares, et dans des programmes assez «pointus», et ne représentent pas une gêne quotidienne pour la programmation.

3.1.2 Conventions et notations

Notations

– Dans toutes les descriptions de nos relations, nous adopterons les lettres suivantes pour désigner des sous-ensembles importants d'arbres :
  A : ensemble des arbres,
  B : ensemble $\{0, 1\}$,
  F : ensemble des arbres d'un nœud,
  L : ensemble des listes,
  Z : ensemble des entiers,
  R : ensemble des nombres réels.
– Nous avons dans ce document utilisé la notation anglo-saxonne pour les intervalles. La table ci-dessous montre les équivalences entre les notations française et anglo-saxonne :

Anglo-saxonne	*Française*
$[a, b]$	$[a, b]$
$[a, b)$	$[a, b[$
$(a, b]$	$]a, b]$
(a, b)	$]a, b[$
$(-\infty, a]$	$]-\infty, a]$
$(-\infty, a)$	$]-\infty, a[$
$[a, +\infty)$	$[a, +\infty[$
$(a, +\infty)$	$]a, +\infty[$
$(-\infty, +\infty)$	$]-\infty, +\infty[$

– Dans tous les exemples présentés ci-dessous, nous supposerons que Prolog IV est dans sa configuration par défaut, soit en mode syntaxique spécifique et en mode «intervalles simple». Pour tous les exemples qui utilisent les unions d'intervalles, la commande permettant le passage en mode «unions d'intervalles» est explicitement placée dans le code.

Nommage des relations

Le nommage des relations présentées dans le présent chapitre obéit à certaines règles précises. Certaines de ces règles sont rappelées ci-dessous.

– Les relations numériques de base existent toutes en deux versions : une est traitée par le solveur linéaire, et l'autre par le solveur sur les intervalles. Les noms des versions «linéaires» sont les mêmes que ceux des versions «intervalles», auxquels on a ajouté le suffixe «-lin». La table ci-dessous montre toutes ces relations, avec leurs raccourcis quand ceux-ci

sont disponibles :

Intervalles	Raccourci	Linéaire	Raccourci
plus/3	.+.	pluslin/3	+
minus/3	.-.	minuslin/3	-
times/3	.*.	timeslin/3	*
div/3	./.	divlin/3	/
uplus/2	.+.	upluslin/2	+
uminus/2	.-.	uminuslin/2	-
ge/2		gelin/2	
gt/2		gtlin/2	
le/2		lelin/2	
lt/2		ltlin/2	

– Pour faire référence à l'appartenance ou à la non-appartenance à un intervalle, des relations différentes sont utilisées selon que l'intervalle est ouvert ou fermé de chaque côté. La convention de nommage utilisée est que le type de l'intervalle est représenté par deux lettres *gd*. La lettre *g* vaut c (pour *closed*) si l'intervalle est fermé à gauche, et o (pour *open*) si l'intervalle est ouvert à gauche, et la lettre *d* est définie similairement pour la droite. La table ci-dessous montre toutes les relations qui utilisent cette convention :

I	$x \in I$	$x \notin I$	$x \in I?$	$x \notin I?$
$[a,b]$	cc/3	outcc/3	bcc/4	boutcc/4
$[a,b)$	co/3	outco/3	bco/4	boutco/4
$(a,b]$	oc/3	outoc/3	boc/4	boutoc/4
(a,b)	oo/3	outoo/3	boo/4	boutoo/4

– Un certain nombre de relations de « typage » sont associées à une négation. Le symbole de cette relation négative est construit en ajoutant le préfixe n au symbole de la relation originale. La table ci-dessous en donne la liste :

Relation	*Négation*
identifier/1	nidentifier/1
int/1	nint/1
leaf/1	nleaf/1
list/1	nlist/1
prime/1	nprime/1
real/1	nreal/1
tree/1	ntree/1

Note : La plupart des relations ci-dessus correspondent à des sous-domaines privilégiés de Prolog IV, et sont donc susceptibles d'apparaître dans des réponses données par Prolog IV. Les exceptions sont int/1, nint/1, prime/1 et nprime/1.

– De nombreuses relations sur les intervalles sont associées à une pseudo-opération produisant un booléen qui est vrai pour les éléments de la relation initiale. Si on considère une relation r/n, cette pseudo-opération a donc une arité $n+1$, et son nom est br (le préfixe b signifie *booléen*). La table ci-dessous liste toutes les relations qui ont une pseudo-opération

associée :

Relation	*Pseudo*	*Relation*	*Pseudo*
and/2	band/3	nidentifier/1	bnidentifier/2
cc/3	bcc/3	nint/1	bnint/2
co/3	bco/3	nleaf/1	bnleaf/2
dif/2	bdif/3	nlist/1	bnlist/2
eq/2	beq/3	not/1	bnot/2
equiv/2	bequiv/3	nprime/1	bnprime/2
finite/1	bfinite/1	nreal/1	bnreal/2
ge/2	bge/3	oc/3	boc/4
gt/2	bgt/3	oo/3	boo/4
identifier/1	bidentifier/2	or/2	bor/3
impl/2	bimpl/3	outcc/3	boutcc/4
infinite/1	binfinite/2	outco/3	boutco/4
inlist/2	binlist/3	outlist/2	boutlist/3
int/1	bint/2	outoc/3	boutoc/4
le/2	ble/3	outoo/3	boutoo/4
leaf/1	bleaf/2	prime/1	bprime/2
list/1	blist/2	real/1	breal/2
lt/2	blt/3	xor/2	bxor/3

3.2 Liste alphabétique

DANS LES PAGES qui suivent, on trouvera la liste de tous les symboles r de relation auxquels Prolog IV associe une relation $\pi_4(r)$ bien précise. Pour faciliter les recherches, cette liste est donnée dans l'ordre alphabétique. La distinction entre symboles de relation et relations est nécessaire : deux symboles distincts – comme par exemple **plus** et **pluslin** – peuvent représenter une même relation mais être traités différemment dans les algorithmes de résolution de contraintes.

abs/2 — Valeur absolue

Définition : **abs/2** définit la relation $\{(a,b) \in \mathbf{R}^2 \mid a = |b|\}$.

Description : **abs/2** est une relation sur les réels, qui est traitée par le solveur sur les intervalles. Elle établit une relation entre une valeur numérique et sa valeur absolue.

Exemples :

```
>> abs(A, B).
B ~ real,
A ~ ge(0).
>> abs(1, B).
B ~ cc(-1,1).
>> abs(A, -1).
A = 1.
>> A = abs(2).
A = 2.
>> A ~ abs(cc(-2,1)).
A ~ cc(0,2).
```

Voir également : ceil/2, floor/2.

and/2 — Et

Définition : **and/2** définit la relation $\{(a,b) \in \mathbf{B}^2 \mid a = 1 \text{ et } b = 1\}$.

Description : **and/2** est une relation sur les booléens, qui est traitée par le solveur sur les intervalles. Une contrainte **and**(x, y) réussit si ses deux arguments valent 1.

Cette contrainte est normalement utilisée pour construire une conjonction d'autres contraintes exprimées par des pseudo-opérations produisant des booléens.

Autre notation : **and**(a, b) s'écrit aussi a **and** b.

Exemples :

```
>> and(A, B).
B = 1,
A = 1.
>> and(A, 0).
false.
>> and(A, bnot(A)).
false.
>> and(bcc(A,1,4), boc(A,2,5)).
A ~ oc(2,4).
```

Voir également : band/3, equiv/2, impl/2, not/1, or/2, xor/2.

arccos/2 — Arc-cosinus

Définition : **arccos/2** définit la relation $\{(a,b) \in \mathbf{R}^2 \mid 0 \leq a \leq \pi \text{ et } b = \cos a\}$.

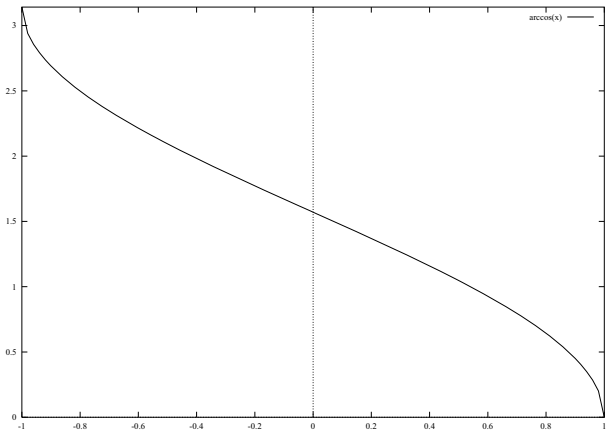
Description : **arccos/2** est une relation sur les réels, qui est traitée par le solveur sur les intervalles. Elle établit une relation entre une valeur numérique et son arccosinus.

Cette relation diffère de la relation **cos/2** par le fait que son premier argument est contraint à appartenir à l'intervalle $[0, \pi]$. Elle n'est donc pas sujette aux problèmes des fonctions périodiques.

Exemples :

```
>> arccos(A, B).
B ~ cc(-1,1),
A ~ co(0,`>3.1415927`).
>> arccos(0, B).
B = 1.
>> arccos(pi./.2, B).
B ~ oo(-`>4.371139e-8`,`>7.54979e-8`).
>> arccos(pi, B).
B ~ co(-1,-`>0.9999999`).
>> arccos(A, -1).
A ~ co(`>3.1415922`,`>3.1415927`).
>> A = 2.*.arccos(0).
A ~ cc(`>3.1415925`,`>3.1415927`).
>> arccos(A, 1).
A = 0.
>> A = 4.*.arccos(sqrt(2)./.2).
A ~ cc(`>3.1415922`,`>3.1415927`).
>> arccos(A, A).
A ~ cc(`>0.739085`,`>0.7390852`).
>> arccos(2.*.pi,B).
false.
```

Graphe :



Voir également : arcsin/2, arctan/2, cos/2, cot/2, pi/2, sin/2, tan/2.

arcsin/2 — Arc-sinus

Définition : **arcsin/2** définit la relation $\{(a,b) \in \mathbf{R}^2 \mid -\pi/2 \leq a \leq \pi/2 \text{ et } b = \sin a\}$.

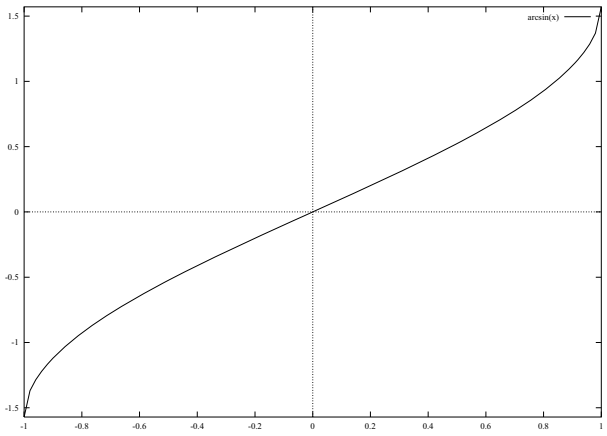
Description : **arcsin/2** est une relation sur les réels, qui est traitée par le solveur sur les intervalles. Elle établit une relation entre une valeur numérique et son arcsinus.

Cette relation diffère de la relation **sin/2** par le fait que son premier argument est contraint à appartenir à l'intervalle $[-\frac{\pi}{2}, \frac{\pi}{2}]$. Elle n'est donc pas sujette aux problèmes des fonctions périodiques.

Exemples :

```
>> arcsin(A, B).
B ~ cc(-1,1),
A ~ oo(-'>1.5707963','>1.5707963').
>> arcsin(0, B).
B = 0.
>> arcsin(pi./.2, B).
B ~ oc('>0.9999999',1).
>> arcsin(A, 0).
A = 0.
>> A = 2.*.arcsin(1).
A ~ oo('>3.1415925','>3.1415927').
>> A = 4.*.arcsin(sqrt(2)./.2).
A ~ oo('>3.1415922','>3.141593').
>> arcsin(A, A).
A ~ oo(-'>0.017607333','<0.01740003').
>> arcsin(pi,B).
false.
```

Graphe :



Voir également : arccos/2, arctan/2, cos/2, cot/2, pi/2, sin/2, tan/2.

arctan/2 — Arc-tangente

Définition : **arctan/2** définit la relation $\{(a,b) \in \mathbf{R}^2 \mid -\pi/2 < a < \pi/2 \text{ et } b = \tan a\}$.

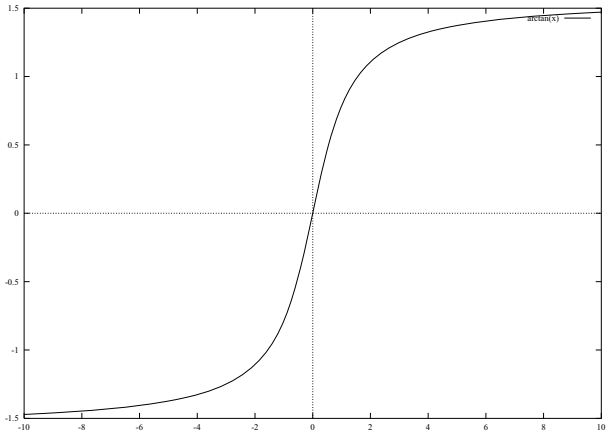
Description : arctan/2 est une relation sur les réels, qui est traitée par le solveur sur les intervalles. Elle établit une relation entre une valeur numérique et son arctangente.

Cette relation diffère de la relation **tan/2** par le fait que son premier argument est contraint à appartenir à l'intervalle $(-\frac{\pi}{2}, \frac{\pi}{2})$. Elle n'est donc pas sujette aux problèmes des fonctions périodiques.

Exemples :

```
>> arctan(A, B).
B ~ real,
A ~ oo(-'>1.5707963','>1.5707963').
>> arctan(0, B).
B = 0.
>> arctan(pi./.4, B).
B ~ oo('<0.9999999','>1.0000001').
>> arctan(A, 0).
A = 0.
>> A = 4.*.arctan(1).
A ~ cc('>3.1415925','>3.1415927').
>> arctan(pi,B).
false.
```

Graphe :



Voir également : arccos/2, arcsin/2, cos/2, cot/2, pi/2, sin/2, tan/2.

band/3 — Et

Définition : **band/3** définit la relation $\{(a,b,c) \in \mathbf{B}^3 \mid a = 1 \text{ ssi } (b,c) \in \pi_4(\text{and}/2)\}$.

Description : band/3 est une relation sur les booléens, qui est traitée par le solveur sur les intervalles. Elle établit une relation entre trois booléens, dont le premier est la conjonction des deux autres.

Cette relation peut être utilisée pour construire des expressions booléennes complexes, ou encore comme connecteur dans la construction d'expressions numériques complexes.

Autre notation : **band**(a, b, c) s'écrit aussi $a = b$ **band** c.

Exemples :

```
>> band(A, B, C).
C ~ cc(0,1),
B ~ cc(0,1),
A ~ cc(0,1).
>> band(0, B, C).
C ~ cc(0,1),
B ~ cc(0,1).
>> band(1, B, C).
C = 1,
B = 1.
>> band(A, 0, C).
A = 0,
C ~ cc(0,1).
>> band(0, 1, C).
C = 0.
>> eq(B, C), band(A, B, C).
B = C,
A ~ cc(0,1),
C ~ cc(0,1).
```

Voir également : and/2, bequiv/3, bimpl/3, bnot/2, bor/3, bxor/3.

bcc/4 — Appartenance à un intervalle

Définition : **bcc/4** définit la relation $\{(a, b, c, d) \in \mathbf{B} \times \mathbf{R}^3 \mid a = 1 \text{ ssi } (b, c, d) \in \pi_4(\text{cc/3})\}$.

Description : **bcc/4** est une relation liant une variable booléenne à trois variables numériques, qui est traitée par le solveur sur les intervalles. Elle établit une relation entre un booléen et le fait qu'une valeur appartienne à un intervalle fermé des deux côtés de la forme $[c, d]$.

En associant une valeur booléenne à la valeur de vérité d'une contrainte d'appartenance à un intervalle, cette relation permet soit de construire des expressions numériques complexes, soit de construire des contraintes complexes. Une énumération sur la valeur de la variable booléenne permet alors de choisir les contraintes à appliquer.

Exemples :

```
>> bcc(A, B, C, D).
D ~ real,
C ~ real,
B ~ real,
A ~ cc(0,1).
>> bcc(A, B, 0, 1).
B ~ real,
A ~ cc(0,1).
>> bcc(1, B, 0, 2).
B ~ cc(0,2).
>> bcc(A, 0, 0, 2).
A = 1.
>> bcc(A, 1, 0, 2).
A = 1.
>> bcc(A, 2, 0, 2).
A = 1.
>> set_prolog_flag(interval_mode,union).
true.
>> bcc(0, B, 0, 2).
B ~ lt(0)u gt(2).
>> set_prolog_flag(interval_mode,simple).
true.
```

Note : La relation **bcc/4** est une des pseudo-opérations produisant des booléens. Quelques conseils rapides sur l'utilisation de ces relations sont donnés ici. Ces conseils sont également valables pour les autres pseudo-opérations produisant des booléens.

En utilisant **bcc/4** dans une disjonction, aucune réduction de domaine des variables numériques n'est effectuée tant qu'il n'est pas possible de déterminer la valeur de la variable booléenne. Dans l'exemple suivant, la réduction attendue est effectuée, car les valeurs des variables booléennes sont connues :

```
>> or(bcc(A,1,2),bcc(A,6,7)), cc(A, 5,10).
A ~ cc(6,7).
```

Par contre, dans l'exemple ci-dessous, aucune réduction n'est effectuée, car il est impossible de donner des valeurs aux variables booléennes :

```
>> or(bcc(A,1,2),bcc(A,6,7)), A ~ cc(1,10).
A ~ cc(1,10).
```

De manière à obtenir une réduction dans un tel cas, il est préférable d'utiliser la relation **u/3**, dont la propagation est immédiate (sans passer par des variables booléennes). Dans ce cas, on utilise la relation **cc/3** au lieu de **bcc/4** :

```
>> A ~ cc(1,2) u cc(4,5), A ~ cc(1,10).
A ~ cc(1,5).
```

Dans les contraintes comme **bcc/4**, le premier argument est un booléen. Toutefois, ce booléen est également un réel, et il est donc possible de l'utiliser dans des contraintes numériques. L'une des applications immédiates est de construire un « opérateur de cardinalité ». Le but est ici de spécifier qu'un certain nombre de contraintes parmi une liste donnée doivent être vérifiées:

```
card([],0).
card([X|L], X .+. S) :-
  card(L, S).
```

Le programme ci-dessus se contente de calculer la somme des éléments d'une liste. Cette somme représente ici le nombre de contraintes vérifiées. Dans

le premier exemple, on donne une liste de 4 contraintes, en demandant que 2 d'entre elles soient vérifiées :

```
>> L ~ [ bcc(X,1,2), bcc(X,2,3), bcc(X,2,4), bcc(X,3,4)],
     card(L,2).
L ~ [cc(0,1),cc(0,1),cc(0,1),cc(0,1)],
X ~ real.
```

Le résultat n'étant pas assez précis, il nous faut effectuer une énumération sur `L` pour savoir quelles contraintes sont vérifiées :

```
>> L ~ [ bcc(X,1,2), bcc(X,2,3), bcc(X,2,4), bcc(X,3,4)],
     card(L,2), boolsplit(L).
L = [0,0,1,1],
X ~ oc(3,4);
L = [0,1,1,0],
X ~ oo(2,3).
```

Il y a donc deux combinaisons dans lesquelles deux des quatre contraintes sont vérifiées. Pour terminer, on peut vouloir toutes les solutions, pour tous les nombres possible de contraintes vérifiées. Il nous faut alors ajouter une énumération sur les nombre de contraintes vérifiées :

```
>> L ~ [ bcc(X,1,2), bcc(X,2,3), bcc(X,2,4), bcc(X,3,4)],
     card(L,N), intsplit([N]), boolsplit(L).
N = 0,
L = [0,0,0,0],
X ~ real;
N = 1,
L = [1,0,0,0],
X ~ co(1,2);
N = 2,
L = [0,0,1,1],
X ~ oc(3,4);
N = 2,
L = [0,1,1,0],
X ~ oo(2,3);
N = 3,
X = 3,
L = [0,1,1,1];
N = 3,
X = 2,
L = [1,1,1,0].
```

Voir également : bco/4, boc/4, boo/4, boutcc/4, cc/3, outcc/3.

bco/4 — Appartenance à un intervalle

Définition : **bco/4** définit la relation $\{(a,b,c,d) \in \mathbf{B}\times\mathbf{R}^3 \mid a = 1 \text{ ssi } (b,c,d) \in \pi_4(\text{co}/3)\}$.

Description : **bco/4** est une relation liant une variable booléenne à trois variables numériques, qui est traitée par le solveur sur les intervalles. Elle établit une relation entre un booléen et le fait qu'une valeur appartienne à un intervalle fermé à gauche et ouvert à droite de la forme $[c,d)$.

En associant une valeur booléenne à la valeur de vérité d'une contrainte d'appartenance à un intervalle, cette relation permet soit de construire des expres-

sions numériques complexes, soit de construire des contraintes complexes. Une énumération sur la valeur de la variable booléenne permet alors de choisir les contraintes à appliquer.

Exemples :

```
>> bco(A, B, C, D).
D ~ real,
C ~ real,
B ~ real,
A ~ cc(0,1).
>> bco(A, B, 0, 1).
B ~ real,
A ~ cc(0,1).
>> bco(1, B, 0, 2).
B ~ co(0,2).
>> bco(A, 0, 0, 2).
A = 1.
>> bco(A, 1, 0, 2).
A = 1.
>> bco(A, 2, 0, 2).
A = 0.
>> set_prolog_flag(interval_mode,union).
true.
>> bco(0, B, 0, 2).
B ~ lt(0)u ge(2).
>> set_prolog_flag(interval_mode,simple).
true.
```

Note : Voir **bcc/4** pour de plus amples informations concernant l'utilisation des prédicats de vérification d'appartenance à des intervalles.

Voir également : bcc/4, boc/4, boo/4, boutco/4, co/3, outco/3.

bdif/3 — Inégalité

Définition : **bdif/3** définit la relation $\{(a,b,c) \in \mathbf{B} \times \mathbf{A}^2 \mid a = 1 \text{ ssi } (b,c) \in \pi_4(\text{dif/2})\}$.

Description : **bdif/3** est une relation liant une variable booléenne à deux variables quelconques, qui est traitée par le solveur général. Elle établit une relation entre un booléen et le fait que deux valeurs soient distinctes.

En associant une valeur booléenne à la valeur de vérité d'une diséquation, cette relation permet soit de construire des contraintes complexes, soit de construire des contraintes. Une énumération sur la valeur de la variable booléenne permet alors de choisir les contraintes à appliquer.

Exemples :

```
>> bdif(A, B, C).
C ~ tree,
B ~ tree,
A ~ cc(0,1).
>> bdif(A, 1, 2).
A = 1.
>> bdif(A, 1, 1).
A = 0.
```

Dans le cas suivant, la réduction est impossible.

```
>> bdif(A, 1, A).
A ~ cc(0,1).
```

On peut énumérer les valeurs possibles de `A` pour se rendre compte que la contrainte n'a pas de solution.

```
>> bdif(A, 1, A), boolsplit([A]).
false.
```

Par contre, elle est ici possible, puisque `id` n'appartient pas au le domaine de la variable `A`.

```
>> bdif(A, id, A).
A = 1.
>> bdif(A, f(X), f(Y)).
Y ~ tree,
X ~ tree,
A ~ cc(0,1).
```

La réduction ne s'effectue pas dans la contrainte suivante, car les domaines de `X` et `Y` sont inchangés par la pose de la contrainte `dif`.

```
>> bdif(A, f(X), f(Y)), dif(X,Y).
Y ~ tree,
X ~ tree,
A ~ cc(0,1).
>> bdif(1, foo, X).
X ~ tree.
>> bdif(0, foo, X).
X = foo.
```

Note : D'un point de vue opérationnel, **bdif/3** est fortement lié à `beq/3`, et pourrait en fait être défini de la manière suivante :

```
bdif(A, X, Y) :-
  A ~ bnot(beq(X,Y)).
```

Voir beq/3 pour des commentaires sur l'utilisation de bdif/3.

Voir également : beq/3, dif/2, eq/2.

beq/3 — Egalité

Définition : **beq/3** définit la relation $\{(a,b,c) \in \mathbf{B} \times \mathbf{A}^2 \mid a = 1 \text{ ssi } (b,c) \in \pi_4(\text{eq}/2)\}$.

Description : `beq/3` est une relation liant une variable booléenne à deux variables quelconques, qui est traitée par le solveur général. Elle établit une relation entre un booléen et le fait que deux valeurs soient égales.

En associant une valeur booléenne à la valeur de vérité d'une équation, cette relation permet soit de construire des contraintes complexes, soit de construire des contraintes. Une énumération sur la valeur de la variable booléenne permet alors de choisir les contraintes à appliquer.

Exemples :

```
>> beq(A, B, C).
C ~ tree,
B ~ tree,
A ~ cc(0,1).
>> beq(A, 1, 2).
A = 0.
>> beq(A, 1, 1).
A = 1.
>> beq(A, 1, A).
A ~ cc(0,1).
>> beq(A, 1, A), boolsplit([A]).
A = 0;
A = 1.
```

Ici, une conclusion est possible, puisque les domaines de `id` et de la variable `A` (qui est une variable booléenne) sont disjoints :

```
>> beq(A, id, A).
A = 0.
>> beq(A, f(X), f(Y)).
Y ~ tree,
X ~ tree,
A ~ cc(0,1).
```

La pose d'une diséquation ne permet souvent pas de réduire le domaine d'une variable, et ne déclenche donc pas de réduction au niveau de la variable booléenne :

```
>> beq(A, f(X), f(Y)), dif(X,Y).
Y ~ tree,
X ~ tree,
A ~ cc(0,1).
>> beq(A, f(X), f(Y)), X = Y .
X = Y,
A = 1,
Y ~ tree.
>> beq(1, foo, X).
X = foo.
>> beq(0, foo, X).
X ~ tree.
```

Note : Contrairement aux autres pseudo-opérations produisant des booléens, les relations **beq/3** et **bdif/3** peuvent réagir à l'égalité entre variables. Par exemple, dans l'exemple suivant, on obtient une réduction de la variable booléenne bien que les domaines des autres variables demeurent inchangés :

```
>> beq(A, f(X), f(Y)), X = Y .
X = Y,
A = 1,
Y ~ tree.
```

Par contre, comme nous l'avons montré dans les exemples, les relations **beq/3** et **bdif/3** ne détectent pas les non-égalités entre variables dans le cas général (c'est-à-dire sans que leurs domaines ne soient disjoints). Par exemple, nous avons :

```
>> beq(A, f(X), f(Y)), dif(X,Y).
Y ~ tree,
X ~ tree,
A ~ cc(0,1).
```

Toutefois, le solveur linéaire permet de déduire des non-égalités entre variables, et ces non-égalités peuvent être remontées à **beq/3** et **bdif/3**, comme on le voit dans les deux exemples suivants :

```
>> beq(A, X, X+1).
A = 0,
X ~ real.
>> bdif(A, X, X+1).
A = 1,
X ~ real.
```

Voir également : bdif/3, dif/2, eq/2.

bequiv/3 — Equivalence

Définition : **bequiv/3** définit la relation $\{(a,b,c) \in \mathbf{B}^3 \mid a = 1 \text{ ssi } (b,c) \in \pi_4(\text{equiv/2})\}$.

Description : **bequiv/3** est une relation sur les booléens, qui est traitée par le solveur sur les intervalles. Elle établit une relation entre trois booléens, dont le premier est vrai (1) si les deux autres sont équivalents.

Cette relation peut être utilisée pour construire des expressions booléennes complexes, ou encore comme connecteur dans la construction d'expressions numériques complexes.

Autre notation : **bequiv**(a,b,c) s'écrit aussi $a = b$ **bequiv** c.

Exemples :

```
>> bequiv(A, B, C).
C ~ cc(0,1),
B ~ cc(0,1),
A ~ cc(0,1).
>> bequiv(A, 1, 1).
A = 1.
>> bequiv(1, B, C).
B = C,
C ~ cc(0,1).
```

Ici, on n'a pas de réduction, à cause de l'indépendance entre la relation **bequiv/3** et la contrainte d'égalité implicite :

```
>> bequiv(A, B, B).
B ~ cc(0,1),
A ~ cc(0,1).
>> bequiv(0, B, C).
C ~ cc(0,1),
B ~ cc(0,1).
>> bequiv(0, 1, C).
C = 0.
```

Note : Nous avons vu dans les exemples que la contrainte `bequiv(A,B,B)` ne permettait pas de réduire le domaine de `A`. Cela est dû au fait que la relation **bequiv/3** ne permet pas de détecter les égalités entre variables. Il est possible d'écrire une version plus puissante de **bequiv/3** en utilisant la relation **beq/3** qui, elle, détecte ces égalités :

```
my_bequiv(A, B, C) :-
  B ~ 0 u 1 ,
  C ~ 0 u 1 ,
  beq(A, B, C).
```

Les deux premières contraintes contraignent les variables `B` et `C` à prendre des valeurs booléennes, et la troisième contraint ces mêmes variables à être égales. On a alors des déductions plus puissantes :

```
>> my_bequiv(A,B,B).
A = 1,
B ~ cc(0,1).
```

Voir également : band/3, bimpl/3, bnot/2, bor/3, bxor/3,equiv/2.

bfinite/2 — Etre fini

Définition : **bfinite/2** définit la relation $\{(a,b) \in \mathbf{B} \times \mathbf{A} \mid a = 1 \text{ ssi } b \in \pi_4(\text{finite}/1)\}$.

Description : **bfinite/2** est une relation liant une variable booléenne à une variable quelconque, qui est traitée par le solveur général et le solveur sur les intervalles. Elle établit une relation entre un booléen et le fait qu'une valeur représente ou non un arbre fini.

En associant une valeur booléenne à une information de typage d'une variable, cette relation permet par exemple de construire des contraintes complexes. Une énumération sur la valeur de la variable booléenne permet alors de choisir les contraintes à appliquer.

Exemples :

```
>> bfinite(A,B).
B ~ tree,
A ~ cc(0,1).
>> bfinite(0,B).
B ~ infinite.
>> bfinite(1,B).
B ~ finite.
>> bfinite(A,cc(1,3)).
A = 1.
>> X = f(X), bfinite(A,X).
A = 0,
X = f(X).
```

Voir également : binfinite/2, finite/1, infinite/1.

bge/3 — Supérieur ou égal

Définition : **bge/3** définit la relation $\{(a, b, c) \in \mathbf{B} \times \mathbf{R}^2 \mid a = 1 \text{ ssi } (b, c) \in \pi_4(\text{ge}/2)\}$.

Description : **bge/3** est une relation liant une variable booléenne à deux variables numériques, qui est traitée par le solveur sur les intervalles. Elle établit une relation entre un booléen et le fait qu'une valeur soit ou non supérieure ou égale à une autre.

En associant une valeur booléenne à la valeur de vérité d'une comparaison numérique, cette relation permet de construire des contraintes complexes. Une énumération sur la valeur de la variable booléenne permet alors d'effectuer une partition sur le domaine de la variable.

Exemples :

```
>> bge(A,B,C).
C ~ real,
B ~ real,
A ~ cc(0,1).
>> bge(0, B, C).
C ~ real,
B ~ real.
>> bge(1, B, C).
C ~ real,
B ~ real.
>> bge(1, B, 2).
B ~ ge(2).
>> bge(A, cc(1,3), cc(2,4)).
A ~ cc(0,1).
>> bge(A, cc(1,2), cc(3,4)).
A = 0.
>> bge(A, cc(5,6), cc(6,7)).
A ~ cc(0,1).
>> bge(A, cc(6,7), cc(5,6)).
A = 1.
>> A ~ bge(X,2), boolsplit([A]).
A = 0,
X ~ lt(2);
A = 1,
X ~ ge(2).
```

Note : Voir **bcc/4** pour de plus amples informations concernant l'utilisation de ce type de contraintes dans des contraintes complexes.

Voir également : bgt/3, ble/3, blt/3, ge/2.

bgt/3 — Supérieur

Définition : **bgt/3** définit la relation $\{(a,b,c) \in \mathbf{B} \times \mathbf{R}^2 \mid a = 1 \text{ ssi } (b,c) \in \pi_4(\text{gt/2})\}$.

Description : **bgt/3** est une relation liant une variable booléenne à deux variables numériques, qui est traitée par le solveur sur les intervalles. Elle établit une relation entre un booléen et le fait qu'une valeur soit ou non supérieure strictement à une autre.

En associant une valeur booléenne à la valeur de vérité d'une comparaison numérique, cette relation permet de construire des contraintes complexes. Une énumération sur la valeur de la variable booléenne permet alors d'effectuer une partition sur le domaine de la variable.

Exemples :

```
>> bgt(A, B, C).
C ~ real,
B ~ real,
A ~ cc(0,1).
>> bgt(0, B, C).
C ~ real,
B ~ real.
>> bgt(1, B, C).
C ~ real,
B ~ real.
>> bgt(1, B, 2).
B ~ gt(2).
>> bgt(A, cc(1,3), cc(2,4)).
A ~ cc(0,1).
>> bgt(A, cc(1,2), cc(3,4)).
A = 0.
>> bgt(A, cc(5,6), cc(6,7)).
A = 0.
>> bgt(A, cc(6,7), cc(5,6)).
A ~ cc(0,1).
>> A ~ bgt(X,2), boolsplit([A]).
A = 0,
X ~ le(2);
A = 1,
X ~ gt(2).
```

Note : Voir **bcc/4** pour de plus amples informations concernant l'utilisation de ce type de contraintes dans des contraintes complexes.

Voir également : bge/3, ble/3, blt/3, gt/2.

bidentifier/2 — Etre un identificateur

Définition : **bidentifier/2** définit la relation $\{(a,b) \in \mathbf{B} \times \mathbf{F} \mid a = 1 \text{ ssi } b \in \pi_4(\text{identifier/1})\}$.

Description : **bidentifier/2** est une relation liant une variable booléenne à une variable quelconque, qui est traitée par le solveur général et le solveur sur les intervalles. Elle établit une relation entre un booléen et le fait qu'une valeur représente ou non un identificateur.

En associant une valeur booléenne à une information de typage d'une variable, cette relation permet par exemple de construire des contraintes complexes. Une énumération sur la valeur de la variable booléenne permet alors de choisir les contraintes à appliquer.

Exemples :

```
>> bidentifier(A,B).
B ~ tree,
A ~ cc(0,1).
>> bidentifier(0,B).
B ~ nidentifier.
>> bidentifier(1,B).
B ~ identifier.
>> bidentifier(A,list).
A ~ cc(0,1).
>> list(L), bidentifier(A,L), boolsplit([A]).
A = 0,
L ~ [tree|list];
A = 1,
L = [].
>> bidentifier(B, toto).
B = 1.
>> bidentifier(B, toto(1)).
B = 0.
```

Voir également : bnidentifier/2, identifier/1, nidentifier/1.

bimpl/3 — Implication

Définition : **bimpl/3** définit la relation $\{(a,b,c) \in \mathbf{B}^3 \mid a = 1 \text{ ssi } (b,c) \in \pi_4(\text{impl/2})\}$.

Description : **bimpl/3** est une relation sur les booléens, qui est traitée par le solveur sur les intervalles. Elle établit une relation entre trois booléens, dont le premier est vrai si le troisième est impliqué par le second.

Cette relation peut être utilisée pour construire des expressions booléennes complexes, ou encore comme connecteur dans la construction d'expressions numériques complexes.

Autre notation : **bimpl**(a,b,c) s'écrit aussi $a = b$ **bimpl** c.

Exemples :

```
>> bimpl(A, B, C).
C ~ cc(0,1),
B ~ cc(0,1),
A ~ cc(0,1).
>> bimpl(0, B, C).
C = 0,
B = 1.
>> bimpl(1, B, C).
C ~ cc(0,1),
B ~ cc(0,1).
>> bimpl(A, 0, C).
A = 1,
C ~ cc(0,1).
>> bimpl(A, 1, C).
A = C,
C ~ cc(0,1).
>> bimpl(A, B, 0).
B ~ cc(0,1),
A ~ cc(0,1).
>> bimpl(A, B, 1).
A = 1,
B ~ cc(0,1).
>> eq(B, C), bimpl(A, B, C).
B = C,
A ~ cc(0,1),
C ~ cc(0,1).
```

Voir également : band/3, bequiv/3, bnot/2, bor/3, bxor/3, impl/2.

binfinite/2 — Etre infini

Définition : **binfinite/2** définit la relation $\{(a,b) \in \mathbf{B} \times \mathbf{A} \mid a = 1 \text{ ssi } b \in \pi_4(\text{infinite/1})\}$.

binfinite/2 est une relation liant une variable booléenne à une variable quelconque, qui est traitée par le solveur général et le solveur sur les intervalles. Elle établit une relation entre un booléen et le fait qu'une valeur représente ou non un terme infini.

En associant une valeur booléenne à une information de typage d'une variable, cette relation permet par exemple de construire des contraintes complexes.

Une énumération sur la valeur de la variable booléenne permet alors de choisir les contraintes à appliquer.

Exemples :

```
>> binfinite(A,B).
B ~ tree,
A ~ cc(0,1).
>> binfinite(0, A).
A ~ finite.
>> binfinite(1, B).
B ~ infinite.
>> binfinite(A, cc(1,3)).
A = 0.
>> X = f(X), binfinite(A, X).
A = 1,
X = f(X).
>> binfinite(0,[X]).
X ~ finite.
```

Voir également : bfinite/2, finite/1, infinite/1.

binlist/3 — Appartenance à une liste

Définition : **binlist/3** définit la relation $\{(a, b, c) \in \mathbf{B} \times \mathbf{A} \times \mathbf{L} \mid a = 1 \text{ ssi } (b, c) \in \pi_4(\text{inlist/2})\}$.

Description : **binlist/3** est une relation liant un booléen, un arbre quelconque et une liste, qui est traitée par le solveur général. Elle établit une relation entre un booléen et le fait qu'un élément appartienne ou non à une liste.

En associant une valeur booléenne à la valeur de vérité d'une contrainte d'appartenance à une liste, cette relation permet de construire des contraintes complexes, et d'utiliser les variables booléennes dans des énumérations.

Exemples :

```
>> binlist(A, B, C).
B ~ tree,
C ~ list,
A ~ cc(0,1).
>> binlist(0, B, C).
B ~ tree,
C ~ list.
>> binlist(1, B, C).
B ~ tree,
C ~ list.
>> binlist(A, 1, [1,2,3]).
A = 1.
>> binlist(A, B, []).
A = 0,
B ~ tree.
>> binlist(A, ge(4), [1,2,3]).
A = 0.
>> binlist(A, B, [1,2,3]).
B ~ real,
A ~ cc(0,1).
>> set_prolog_flag(interval_mode,union).
true.
>> binlist(A, B, [1,2,3]).
B ~ real,
A ~ 0 u 1.
>> set_prolog_flag(interval_mode,simple).
true.
```

Note : Voir **inlist/2** pour des remarques importantes sur le fonctionnement des contraintes d'appartenance à une liste.

Voir également : boutlist/3, inlist/2, outlist/2.

bint/2 — Etre un entier

Définition : **bint/2** définit la relation $\{(a,b) \in \mathbf{B} \times \mathbf{R} \mid a = 1 \text{ ssi } b \in \pi_4(\text{int}/1)\}$.

Description : **bint/2** est une relation liant une variable booléenne à une variable numérique, qui est traitée par le solveur sur les intervalles. Elle établit une relation entre un booléen et le fait qu'une valeur représente ou non un entier.

En associant une valeur booléenne à une information de typage d'une variable, cette relation permet par exemple de construire des contraintes complexes. Une énumération sur la valeur de la variable booléenne permet alors de choisir les contraintes à appliquer.

Exemples :

```
>> bint(A, B).
B ~ real,
A ~ cc(0,1).
>> bint(0, B).
B ~ real.
>> bint(1, B).
B ~ real.
>> bint(A, pi).
A = 0.
>> bint(A, oo(1,2)).
A = 0.
>> oc(X, 1, 2), bint(B,X), boolsplit([B]).
B = 0,
X ~ oo(1,2);
B = 1,
X = 2.
```

En mode intervalle simples, la contrainte «int» ne réduit pas suffisamment les domaines des variables pour permettre de conclure :

```
>> bint(A,I), I ~ int n cc(1,10).
I ~ cc(1,10),
A ~ cc(0,1).
```

Par contre, en mode union d'intervalles, une conclusion peut être obtenue :

```
>> set_prolog_flag(interval_mode,union).
true.
>> bint(A,I), I ~ int n cc(1,10).
A = 1,
I ~ 1 u 2 u 3 u 4 u 5 u 6 u 7 u 8 u 9 u 10.
>> set_prolog_flag(interval_mode,simple).
true.
```

Note : **bint/2** est définie sur l'ensemble $\mathbf{B} \times \mathbf{R}$. Dans le cas où son deuxième argument prend une valeur non numérique, cette contrainte n'affecte pas la valeur 0 à son premier argument, mais échoue, comme on le voit dans l'exemple suivant.

```
>> bint(A,toto).
false.
```

Ceci est bien entendu dû au fait que l'ensemble des entiers ne constituent pas un sous-domaine de Prolog IV, mais seulement une contrainte numérique, qui est donc traitée en tant que telle.

Attention en utilisant la relation **bint/3** en mode union d'intervalles. Si le domaine d'une variable n'est pas restreint, des problèmes de mémoire peuvent

survenir. Voir **int/1** pour plus de détails.

Voir également : bnint/2, int/1, nint/1.

ble/3 — Inférieur ou égal

Définition : **ble/3** définit la relation $\{(a, b, c) \in \mathbf{B} \times \mathbf{R}^2 \mid a = 1 \text{ ssi } (b, c) \in \pi_4(\text{le/2})\}$.

Description : **ble/3** est une relation liant une variable booléenne à deux variables numériques, qui est traitée par le solveur sur les intervalles. Elle établit une relation entre un booléen et le fait qu'une valeur soit ou non inférieure ou égale à une autre.

En associant une valeur booléenne à la valeur de vérité d'une comparaison numérique, cette relation permet de construire des contraintes complexes. Une énumération sur la valeur de la variable booléenne permet alors d'effectuer une partition sur le domaine de la variable.

Exemples :

```
>> ble(A, B, C).
C ~ real,
B ~ real,
A ~ cc(0,1).
>> ble(0, B, C).
C ~ real,
B ~ real.
>> ble(1, B, C).
C ~ real,
B ~ real.
>> ble(1, B, 2).
B ~ le(2).
>> ble(A, cc(1,3), cc(2,4)).
A ~ cc(0,1).
>> ble(A, cc(1,2), cc(3,4)).
A = 1.
>> ble(A, cc(5,6), cc(6,7)).
A = 1.
>> ble(A, cc(6,7), cc(5,6)).
A ~ cc(0,1).
>> A ~ ble(X,2), boolsplit([A]).
A = 0,
X ~ gt(2);
A = 1,
X ~ le(2).
```

Note : Voir **bcc/4** pour de plus amples informations concernant l'utilisation de ce type de contraintes dans des contraintes complexes.

Voir également : bge/3, bgt/3, blt/3, le/2.

bleaf/2 — Etre un arbre d'un nœud

Définition : **bleaf/2** définit la relation $\{(a,b) \in \mathbf{B} \times \mathbf{A} \mid a = 1 \text{ ssi } b \in \pi_4(\text{leaf/1})\}$.

Description : **bleaf/2** est une relation liant une variable booléenne à une variable quelconque, qui est traitée par le solveur général et le solveur sur les intervalles. Elle établit une relation entre un booléen et le fait qu'une valeur représente ou non un arbre réduit à un seul nœud.

En associant une valeur booléenne à une information de typage d'une variable, cette relation permet par exemple de construire des contraintes complexes. Une énumération sur la valeur de la variable booléenne permet alors de choisir les contraintes à appliquer.

Exemples :

```
>> bleaf(A,B).
B ~ tree,
A ~ cc(0,1).
>> bleaf(0,B).
B ~ nleaf.
>> bleaf(1, B).
B ~ leaf.
```

Bien entendu, aucun arbre infini ne peut être un arbre d'un nœud :

```
>> bleaf(A, infinite).
A = 0.
>> bleaf(A, cc(1,2)).
A = 1.
>> list(L), bleaf(A, L), boolsplit([A]).
A = 0,
L ~ [tree|list];
A = 1,
L = [].
```

Voir également : bnleaf/2, leaf/1, nleaf/1.

blist/2 — Etre une liste

Définition : **blist/2** définit la relation $\{(a,b) \in \mathbf{B} \times \mathbf{A} \mid a = 1 \text{ ssi } b \in \pi_4(\text{list/1})\}$.

Description : **blist/2** est une relation liant une variable booléenne à une variable quelconque, qui est traitée par le solveur général et le solveur sur les intervalles. Elle établit une relation entre un booléen et le fait qu'une valeur représente ou non une liste.

En associant une valeur booléenne à une information de typage d'une variable, cette relation permet par exemple de construire des contraintes complexes. Une énumération sur la valeur de la variable booléenne permet alors de choisir les contraintes à appliquer.

Exemples :

```
>> blist(A, B).
B ~ tree,
A ~ cc(0,1).
>> blist(0, B).
B ~ nlist.
>> blist(1, B).
B ~ list.
>> leaf(X), blist(A, X).
A ~ cc(0,1),
X ~ leaf.
>> leaf(X), blist(A, X), boolsplit([A]).
A = 0,
X ~ leaf,
X ~ nlist;
A = 1,
X = [].
>> blist(A, [X,Y]).
A = 1,
Y ~ tree,
X ~ tree.
```

Voir également : bnlist/2, list/1, nlist/1.

blt/3 — Inférieur

Définition : **blt/3** définit la relation $\{(a, b, c) \in \mathbf{B} \times \mathbf{R}^2 \mid a = 1 \text{ ssi } (b, c) \in \pi_4(\text{lt}/2)\}$.

Description : **blt/3** est une relation liant une variable booléenne à deux variables numériques, qui est traitée par le solveur sur les intervalles. Elle établit une relation entre un booléen et le fait qu'une valeur soit ou non inférieure strictement à une autre.

En associant une valeur booléenne à la valeur de vérité d'une comparaison numérique, cette relation permet de construire des contraintes complexes. Une énumération sur la valeur de la variable booléenne permet alors d'effectuer une partition sur le domaine de la variable.

Exemples :

```
>> blt(A, B, C).
C ~ real,
B ~ real,
A ~ cc(0,1).
>> blt(0, B, C).
C ~ real,
B ~ real.
>> blt(1, B, C).
C ~ real,
B ~ real.
>> blt(1, B, 2).
B ~ lt(2).
>> blt(A, cc(1,3), cc(2,4)).
A ~ cc(0,1).
>> blt(A, cc(1,2), cc(3,4)).
A = 1.
>> blt(A, cc(5,6), cc(6,7)).
A ~ cc(0,1).
>> blt(A, cc(6,7), cc(5,6)).
A = 0.
>> A ~ blt(X,2), boolsplit([A]).
A = 0,
X ~ ge(2);
A = 1,
X ~ lt(2).
```

Note : Voir **bcc/4** pour de plus amples informations concernant l'utilisation de ce type de contraintes dans des contraintes complexes.

Voir également : bge/3, bgt/3, ble/3, lt/2.

bnidentifier/2 — Ne pas être un identificateur

Définition : **bnidentifier/2** définit la relation $\{(a,b) \in \mathbf{B} \times \mathbf{F} \mid a = 1 \text{ ssi } b \in \pi_4(\text{nidentifier/1})\}$.

Description : **bnidentifier/2** est une relation liant une variable booléenne à une variable quelconque, qui est traitée par le solveur général et le solveur sur les intervalles. Elle établit une relation entre un booléen et le fait qu'une valeur représente ou non un identificateur.

En associant une valeur booléenne à une information de typage d'une variable, cette relation permet par exemple de construire des contraintes complexes. Une énumération sur la valeur de la variable booléenne permet alors de choisir les contraintes à appliquer.

Exemples :

```
>> bnidentifier(A, B).
B ~ tree,
A ~ cc(0,1).
>> bnidentifier(0, B).
B ~ identifier.
>> bnidentifier(1, B).
B ~ nidentifier.
>> bnidentifier(A, list).
A ~ cc(0,1).
>> list(L), bnidentifier(A,L), boolsplit([A]).
A = 0,
L = [];
A = 1,
L ~ [tree|list].
>> bnidentifier(A, toto).
A = 0.
>> bnidentifier(A, toto(1)).
A = 1.
```

Voir également : bidentifier/2, identifier/1, nidentifier/1.

bnint/2 — Ne pas être un entier

Définition : **bnint/2** définit la relation $\{(a, b) \in \mathbf{B} \times \mathbf{R} \mid a = 1 \text{ ssi } b \in \pi_4(\text{nint/1})\}$.

Description : **bnint/2** est une relation liant une variable booléenne à une variable numérique, qui est traitée par le solveur sur les intervalles. Elle établit une relation entre un booléen et le fait qu'une valeur représente ou non un nombre non-entier.

Exemples :

```
>> bnint(A, B).
B ~ real,
A ~ cc(0,1).
>> bnint(0, B).
B ~ real.
>> bnint(1, B).
B ~ real.
>> bnint(A, pi).
A = 1.
>> bnint(A, oo(1,2)).
A = 1.
>> oc(X, 1, 2), bnint(B,X), boolsplit([B]).
B = 0,
X = 2;
B = 1,
X ~ oo(1,2).
```

Note : Voir **bint/2** pour des remarques importantes sur le fonctionnement des contraintes **bint/2** et **bnint/2**.

Voir également : bint/2, int/1, nint/1.

bnleaf/2 — Etre un arbre de plus d'un nœud

Définition : **bnleaf/2** définit la relation $\{(a,b) \in \mathbf{B} \times \mathbf{A} \mid a = 1 \text{ ssi } b \in \pi_4(\text{nleaf/1})\}$.

Description : **bnleaf/2** est une relation liant une variable booléenne à une variable quelconque, qui est traitée par le solveur général et le solveur sur les intervalles. Elle établit une relation entre un booléen et le fait qu'une valeur représente ou non un arbre réduit à un seul nœud.

En associant une valeur booléenne à une information de typage d'une variable, cette relation permet par exemple de construire des contraintes complexes. Une énumération sur la valeur de la variable booléenne permet alors de choisir les contraintes à appliquer.

Exemples :

```
>> bnleaf(A,B).
B ~ tree,
A ~ cc(0,1).
>> bnleaf(0, B).
B ~ leaf.
>> bnleaf(1, B).
B ~ nleaf.
>> bnleaf(A, infinite).
A = 1.
>> bnleaf(A, cc(1,2)).
A = 0.
>> list(L), bnleaf(A, L), boolsplit([A]).
A = 0,
L = [];
A = 1,
L ~ [tree|list].
```

Voir également : bleaf/2, leaf/1, nleaf/1.

bnlist/2 — Ne pas être une liste

Définition : **bnlist/2** définit la relation $\{(a,b) \in \mathbf{B} \times \mathbf{A} \mid a = 1 \text{ ssi } b \in \pi_4(\text{list}/1)\}$.

Description : **bnlist/2** est une relation liant une variable booléenne à une variable quelconque, qui est traitée par le solveur général et le solveur sur les intervalles. Elle établit une relation entre un booléen et le fait qu'une valeur représente ou non une liste.

En associant une valeur booléenne à une information de typage d'une variable, cette relation permet par exemple de construire des contraintes complexes. Une énumération sur la valeur de la variable booléenne permet alors de choisir les contraintes à appliquer.

Exemples :

```
>> bnlist(A, B).
B ~ tree,
A ~ cc(0,1).
>> bnlist(0, B).
B ~ list.
>> bnlist(1, B).
B ~ nlist.
>> leaf(X), bnlist(A, X).
A ~ cc(0,1),
X ~ leaf.
>> leaf(X), bnlist(A, X), boolsplit([A]).
A = 0,
X = [];
A = 1,
X ~ leaf,
X ~ nlist.
>> bnlist(A, [X,Y]).
A = 0,
Y ~ tree,
X ~ tree.
```

Voir également : blist/2, list/1, nlist/1.

bnot/2 — Négation

Définition : **bnot/2** définit la relation $\{(a,b) \in \mathbf{B}^2 \mid a = 1 \text{ ssi } b \in \pi_4(\text{not}/1)\}$.

bnot/2 est une relation sur les booléens, qui est traitée par le solveur sur les intervalles. Elle établit une relation entre deux booléens, dont le premier est la négation du second.

Cette relation peut être utilisée pour construire des expressions booléennes complexes, ou encore comme connecteur dans la construction d'expressions numériques complexes.

Autre notation : **bnot**(a,b) s'écrit aussi $a =$ **bnot** b.

Exemples :

```
>> bnot(A, B).
B ~ cc(0,1),
A ~ cc(0,1).
>> bnot(0, B).
B = 1.
>> bnot(1, B).
B = 0.
>> bnot(A, 0).
A = 1.
>> bnot(A, 1).
A = 0.
```

Voir également : band/3, bequiv/3, bimpl/3, bor/3, bxor/3, not/1.

bnprime/2 — Ne pas être un entier premier

Définition : **bnprime/2** définit la relation $\{(a,b) \in \mathbf{B} \times \mathbf{R} \mid a = 1 \text{ ssi } b \in \pi_4(\text{nprime/1})\}$.

Description : La relation **bnprime/2** n'est pas implantée dans la version actuelle de Prolog IV. Toutefois, étant donné qu'elle fait partie de la définition du langage, le symbole a été réservé.

Exemples :

```
>> bnprime(A, B).
Relation not yet implemented: bnprime
false.
```

Voir également : bint/2, bnint/2, bprime/2, int/1, nint/1, nprime/2, prime/1.

bnreal/2 — Ne pas être un réel

Définition : **bnreal/2** définit la relation $\{(a,b) \in \mathbf{B} \times \mathbf{A} \mid a = 1 \text{ ssi } b \in \pi_4(\text{nreal/1})\}$.

Description : **bnreal/2** est une relation liant une variable booléenne à une variable quelconque, qui est traitée par le solveur général et le solveur sur les intervalles. Elle établit une relation entre un booléen et le fait qu'une valeur représente ou non un nombre réel.

En associant une valeur booléenne à une information de typage d'une variable, cette relation permet par exemple de construire des contraintes complexes. Une énumération sur la valeur de la variable booléenne permet alors de choisir les contraintes à appliquer.

Exemples :

```
>> bnreal(A,B).
B ~ tree,
A ~ cc(0,1).
>> bnreal(0, B).
B ~ real.
>> bnreal(1, B).
B ~ nreal.
>> bnreal(A, f(X)).
A = 1,
X ~ tree.
>> bnreal(A, cc(1,2)).
A = 0.
```

Voir également : breal/2, nreal/1, real/1.

boc/4 — Appartenance à un intervalle

Définition : **boc/4** définit la relation $\{(a,b,c,d) \in \mathbf{B}\times\mathbf{R}^3 \mid a = 1 \text{ ssi } (b,c,d) \in \pi_4(\text{oc/3})\}$.

Description : **boc/4** est une relation liant une variable booléenne à trois variables numériques, qui est traitée par le solveur sur les intervalles. Elle établit une relation entre un booléen et le fait qu'une valeur appartienne à un intervalle ouvert à gauche et fermé à droite de la forme $(c,d]$.

En associant une valeur booléenne à la valeur de vérité d'une contrainte d'appartenance à un intervalle, cette relation permet soit de construire des expressions numériques complexes, soit de construire des contraintes complexes. Une énumération sur la valeur de la variable booléenne permet alors de choisir les contraintes à appliquer.

Exemples :

```
>> boc(A, B, C, D).
D ~ real,
C ~ real,
B ~ real,
A ~ cc(0,1).
>> boc(A, B, 0, 1).
B ~ real,
A ~ cc(0,1).
>> boc(1, B, 0, 2).
B ~ oc(0,2).
>> boc(A, 0, 0, 2).
A = 0.
>> boc(A, 1, 0, 2).
A = 1.
>> boc(A, 2, 0, 2).
A = 1.
>> set_prolog_flag(interval_mode,union).
true.
>> boc(0, B, 0, 2).
B ~ le(0)u gt(2).
>> set_prolog_flag(interval_mode,simple).
true.
```

Note : Voir **bcc/4** pour de plus amples informations concernant l'utilisation des prédicats de vérification d'appartenance à des intervalles.

Voir également : bcc/4, bco/4, boo/4, boutoc/4, oc/3, outco/3, outoc/3.

boo/4 — Appartenance à un intervalle

Définition : **boo/4** définit la relation $\{(a,b,c,d) \in \mathbf{B}\times\mathbf{R}^3 \mid a = 1 \text{ ssi } (b,c,d) \in \pi_4(\text{oo/3})\}$.

Description : **boo/4** est une relation liant une variable booléenne à trois variables numériques, qui est traitée par le solveur sur les intervalles. Elle établit une relation entre un booléen et le fait qu'une valeur appartienne à un intervalle ouvert des deux côtés de la forme (c,d).

En associant une valeur booléenne à la valeur de vérité d'une contrainte d'appartenance à un intervalle, cette relation permet soit de construire des expres-

sions numériques complexes, soit de construire des contraintes complexes. Une énumération sur la valeur de la variable booléenne permet alors de choisir les contraintes à appliquer.

Exemples :

```
>> boo(A, B, C, D).
D ~ real,
C ~ real,
B ~ real,
A ~ cc(0,1).
>> boo(A, B, 0, 1).
B ~ real,
A ~ cc(0,1).
>> boo(1, B, 0, 2).
B ~ oo(0,2).
>> boo(A, 0, 0, 2).
A = 0.
>> boo(A, 1, 0, 2).
A = 1.
>> boo(A, 2, 0, 2).
A = 0.
>> set_prolog_flag(interval_mode,union).
true.
>> boo(0, B, 0, 2).
B ~ le(0)u ge(2).
>> set_prolog_flag(interval_mode,simple).
true.
```

Note : Voir **bcc/4** pour de plus amples informations concernant l'utilisation des prédicats de vérification d'appartenance à des intervalles.

Voir également : bcc/4, bco/4, boc/4, boutoo/4, oo/3, outoo/3.

bor/3 — Ou

Définition : **bor/3** définit la relation $\{(a,b,c) \in \mathbf{B}^3 \mid a = 1 \text{ ssi } (b,c) \in \pi_4(\text{or/2})\}$.

Description : **bor/3** est une relation sur les booléens, qui est traitée par le solveur sur les intervalles. Elle établit une relation entre trois booléens, dont le premier est la disjonction des deux autres.

Cette relation peut être utilisée pour construire des expressions booléennes complexes, ou encore comme connecteur dans la construction d'expressions numériques complexes.

Autre notation : **bor**(a,b,c) s'écrit aussi $a = b$ **bor** c.

Exemples :

```
>> bor(A, B, C).
C ~ cc(0,1),
B ~ cc(0,1),
A ~ cc(0,1).
>> bor(0, B, C).
C = 0,
B = 0.
>> bor(1, B, C).
C ~ cc(0,1),
B ~ cc(0,1).
>> bor(1, 0, C).
C = 1.
>> bor(1, 1, C).
C ~ cc(0,1).
>> bor(A, 1, C).
A = 1,
C ~ cc(0,1).
```

Voir également : band/3, bequiv/3, bimpl/3, bnot/2, bxor/3, or/2.

boutcc/4 — Non-appartenance à un intervalle

Définition : **boutcc/4** définit la relation $\{(a,b,c,d) \in \mathbf{B} \times \mathbf{R}^3 \mid a = 1 \text{ ssi } (b,c,d) \in \pi_4(\text{outcc/3})\}$.

Description : **boutcc/4** est une relation liant une variable booléenne à trois variables numériques, qui est traitée par le solveur sur les intervalles. Elle établit une relation entre un booléen et le fait qu'une valeur n'appartienne pas à un intervalle fermé des deux côtés de la forme $[c,d]$.

En associant une valeur booléenne à la valeur de vérité d'une contrainte de non-appartenance à un intervalle, cette relation permet soit de construire des expressions numériques complexes, soit de construire des contraintes complexes. Une énumération sur la valeur de la variable booléenne permet alors de choisir les contraintes à appliquer.

Exemples :

```
>> boutcc(A, B, C, D).
D ~ real,
C ~ real,
B ~ real,
A ~ cc(0,1).
>> boutcc(A, B, 0, 1).
B ~ real,
A ~ cc(0,1).
>> boutcc(1, B, 0, 2).
B ~ real.
>> boutcc(0, B, 0, 2).
B ~ cc(0,2).
>> boutcc(A, 0, 0, 2).
A = 0.
>> boutcc(A, 1, 0, 2).
A = 0.
>> boutcc(A, 2, 0, 2).
A = 0.
>> set_prolog_flag(interval_mode,union).
true.
>> boutcc(0, B, 0, 2).
B ~ cc(0,2).
>> boutcc(1, B, 0, 2).
B ~ lt(0)u gt(2).
>> set_prolog_flag(interval_mode,simple).
true.
```

Voir également : bcc/4, boutco/4, boutoc/4, boutoo/4, cc/3, outcc/3.

boutco/4 — Non-appartenance à un intervalle

Définition : **boutco/4** définit la relation $\{(a,b,c,d) \in \mathbf{B} \times \mathbf{R}^2 \mid a = 1 \text{ ssi } (b,c,d) \in \pi_4(\text{outco/3})\}$.

Description : **boutco/4** est une relation liant une variable booléenne à trois variables numériques, qui est traitée par le solveur sur les intervalles. Elle établit une relation entre un booléen et le fait qu'une valeur n'appartienne pas à un intervalle fermé à gauche et ouvert à droite de la forme $[c,d)$.

En associant une valeur booléenne à la valeur de vérité d'une contrainte de non-appartenance à un intervalle, cette relation permet soit de construire des expressions numériques complexes, soit de construire des contraintes complexes. Une énumération sur la valeur de la variable booléenne permet alors de choisir les contraintes à appliquer.

Exemples :

```
>> boutco(A, B, C, D).
D ~ real,
C ~ real,
B ~ real,
A ~ cc(0,1).
>> boutco(A, B, 0, 1).
B ~ real,
A ~ cc(0,1).
>> boutco(1, B, 0, 2).
B ~ real.
>> boutco(0, B, 0, 2).
B ~ co(0,2).
>> boutco(A, 0, 0, 2).
A = 0.
>> boutco(A, 1, 0, 2).
A = 0.
>> boutco(A, 2, 0, 2).
A = 1.
>> set_prolog_flag(interval_mode,union).
true.
>> boutco(0, B, 0, 2).
B ~ co(0,2).
>> boutco(1, B, 0, 2).
B ~ lt(0)u ge(2).
>> set_prolog_flag(interval_mode,simple).
true.
```

Voir également : bco/4, boutcc/4, boutoc/4, boutoo/4, co/3, outco/3.

boutlist/3 — Non-appartenance à une liste

Définition : **boutlist/3** définit la relation $\{(a,b,c) \in \mathbf{B} \times \mathbf{A} \times \mathbf{L} \mid a = 1 \text{ ssi } (b,c) \in \pi_4(\text{outlist/2})\}$.

Description : **boutlist/3** est une relation liant un booléen, un arbre quelconque et une liste, qui est traitée par le solveur général. Elle établit une relation entre un booléen et le fait qu'un élément appartienne ou non à une liste.

En associant une valeur booléenne à la valeur de vérité d'une contrainte de non-appartenance à une liste, cette relation permet de construire des contraintes complexes, et d'utiliser les variables booléennes dans des énumérations.

Exemples :

```
>> boutlist(A, B, C).
B ~ tree,
C ~ list,
A ~ cc(0,1).
>> boutlist(1, B, C).
B ~ tree,
C ~ list.
>> boutlist(1, B, [1,2,3]).
B ~ tree.
>> boutlist(0, B, [1,2,3]).
B ~ cc(1,3).
>> boutlist(A, 1, [1,2,3]).
A = 0.
>> boutlist(A, B, []).
A = 1,
B ~ tree.
```

Note : Voir **inlist/2** pour des remarques importantes sur le fonctionnement des contraintes d'appartenance à une liste.

Voir également : binlist/3, inlist/2, outlist/2.

boutoc/4 — Non-appartenance à un intervalle

Définition : **boutoc/4** définit la relation $\{(a,b,c,d) \in \mathbf{B} \times \mathbf{R}^2 \mid a = 1 \text{ ssi } (b,c,d) \in \pi_4(\text{outoc/3})\}$.

Description : **boutoc/4** est une relation liant une variable booléenne à trois variables numériques, qui est traitée par le solveur sur les intervalles. Elle établit une relation entre un booléen et le fait qu'une valeur n'appartienne pas à un intervalle ouvert à gauche et fermé à droite de la forme $(c,d]$.

En associant une valeur booléenne à la valeur de vérité d'une contrainte de non-appartenance à un intervalle, cette relation permet soit de construire des expressions numériques complexes, soit de construire des contraintes complexes. Une énumération sur la valeur de la variable booléenne permet alors de choisir les contraintes à appliquer.

Exemples :

```
>> boutoc(A, B, C, D).
D ~ real,
C ~ real,
B ~ real,
A ~ cc(0,1).
>> boutoc(A, B, 0, 1).
B ~ real,
A ~ cc(0,1).
>> boutoc(1, B, 0, 2).
B ~ real.
>> boutoc(0, B, 0, 2).
B ~ oc(0,2).
>> boutoc(A, 0, 0, 2).
A = 1.
>> boutoc(A, 1, 0, 2).
A = 0.
>> boutoc(A, 2, 0, 2).
A = 0.
>> set_prolog_flag(interval_mode,union).
true.
>> boutoc(0, B, 0, 2).
B ~ oc(0,2).
>> boutoc(1, B, 0, 2).
B ~ le(0)u gt(2).
>> set_prolog_flag(interval_mode,simple).
true.
```

Voir également : boc/4, boutcc/4, boutco/4, boutoo/4, oc/3, outoc/3.

boutoo/4 — Non-appartenance à un intervalle

Définition : **boutoo/4** définit la relation $\{(a,b,c,d) \in \mathbf{B} \times \mathbf{R}^2 \mid a = 1 \text{ ssi } (b,c,d) \in \pi_4(\text{outoo/3})\}$.

Description : **boutoo/4** est une relation liant une variable booléenne à trois variables numériques, qui est traitée par le solveur sur les intervalles. Elle établit une relation entre un booléen et le fait qu'une valeur n'appartienne pas à un intervalle ouvert des deux côtés de la forme (c,d).

En associant une valeur booléenne à la valeur de vérité d'une contrainte de non-appartenance à un intervalle, cette relation permet soit de construire des expressions numériques complexes, soit de construire des contraintes complexes. Une énumération sur la valeur de la variable booléenne permet alors de choisir les contraintes à appliquer.

Exemples :

```
>> boutoo(A, B, C, D).
D ~ real,
C ~ real,
B ~ real,
A ~ cc(0,1).
>> boutoo(A, B, 0, 1).
B ~ real,
A ~ cc(0,1).
>> boutoo(1, B, 0, 2).
B ~ real.
>> boutoo(0, B, 0, 2).
B ~ oo(0,2).
>> boutoo(A, 0, 0, 2).
A = 1.
>> boutoo(A, 1, 0, 2).
A = 0.
>> boutoo(A, 2, 0, 2).
A = 1.
>> set_prolog_flag(interval_mode,union).
true.
>> boutoo(0, B, 0, 2).
B ~ oo(0,2).
>> boutoo(1, B, 0, 2).
B ~ le(0)u ge(2).
>> set_prolog_flag(interval_mode,simple).
true.
```

Voir également : boo/4, boutcc/4, boutco/4, boutoc/4, oo/3, outoo/3.

bprime/2 — Etre un entier premier

Définition : **bprime/2** définit la relation $\{(a,b) \in \mathbf{B} \times \mathbf{R} \mid a = 1 \text{ ssi } b \in \pi_4(\text{prime/1})\}$.

Description : La relation **bnprime/2** n'est pas implantée dans la version actuelle de Prolog IV. Toutefois, étant donné qu'elle fait partie de la définition du langage, le symbole a été réservé.

Exemples :

```
>> bprime(A, B).
Relation not yet implemented: bprime
false.
```

Voir également : bint/2, bnint/2, bnprime/2, int/1, nint/1, nprime/2, prime/1.

breal/2 — Etre un réel

Définition : **breal/2** définit la relation $\{(a, b) \in \mathbf{B} \times \mathbf{A} \mid a = 1 \text{ ssi } b \in \pi_4(\text{real}/1)\}$.

Description : **breal/2** est une relation liant une variable booléenne à une variable quelconque, qui est traitée par le solveur général et le solveur sur les intervalles. Elle établit une relation entre un booléen et le fait qu'une valeur représente ou non un nombre réel.

En associant une valeur booléenne à une information de typage d'une variable, cette relation permet par exemple de construire des contraintes complexes. Une énumération sur la valeur de la variable booléenne permet alors de choisir les contraintes à appliquer.

Exemples :

```
>> breal(A, B).
B ~ tree,
A ~ cc(0,1).
>> breal(0, B).
B ~ nreal.
>> breal(1, B).
B ~ real.
>> breal(A, toto).
A = 0.
>> breal(A, cc(1, 3)).
A = 1.
```

Voir également : bnreal/2, nreal/1, real/1.

bxor/3 — Ou exclusif

Définition : **bxor/3** définit la relation $\{(a, b, c) \in \mathbf{B}^3 \mid a = 1 \text{ ssi } (b, c) \in \pi_4(\text{xor}/2)\}$.

Description : **bxor/3** est une relation sur les booléens, qui est traitée par le solveur sur les intervalles. Elle établit une relation entre trois booléens, dont le premier est vrai si le second et le troisième soit différents (c'est-à-dire un ou exclusif).

Cette relation peut être utilisée pour construire des expressions booléennes complexes, ou encore comme connecteur dans la construction d'expressions numériques complexes.

Autre notation : **bxor**(a, b, c) s'écrit aussi $a = b$ **bxor** c.

Exemples :

```
>> bxor(A, B, C).
C ~ cc(0,1),
B ~ cc(0,1),
A ~ cc(0,1).
>> bxor(0, B, C).
B = C,
C ~ cc(0,1).
>> bxor(1, B, C).
C ~ cc(0,1),
B ~ cc(0,1).
>> bxor(A, 0, C).
A = C,
C ~ cc(0,1).
>> bxor(A, 1, C).
C ~ cc(0,1),
A ~ cc(0,1).
>> bxor(1, 0, C).
C = 1.
>> bxor(1, 1, C).
C = 0.
>> eq(B, C), bxor(A, B, C).
B = C,
A ~ cc(0,1),
C ~ cc(0,1).
>> dif(B, C), bxor(A, B, C).
A ~ cc(0,1),
C ~ cc(0,1),
B ~ cc(0,1).
```

Voir également : band/3, bequiv/3, bimpl/3, bnot/2, bor/3, xor/2.

cc/3 — Appartenance à un intervalle

Définition : **cc/3** définit la relation $\{(a,b,c) \in \mathbf{R}^3 \mid a \in [b,c]\}$.

Description : **cc/3** est une relation sur les nombres réels, qui est traitée par le solveur sur les intervalles. Elle établit une relation entre un nombre et les bornes d'un intervalle fermé des deux côtés de la forme $[b,c]$ le contenant.

cc/3 est très couramment utilisée en Prolog IV, dans la mesure où cette relation est utilisée pour fixer le domaine d'une variable. Son utilisation la plus courante est donc sous la forme **cc**(A, v_1, v_2) où v_1 et v_2 sont des constantes numériques. Toutefois, cette relation peut également être utilisée pour effectuer des encadrements.

Exemples :

```
>> cc(A, B, C).
C ~ real,
B ~ real,
A ~ real.
>> cc(A, 1, 2).
A ~ cc(1,2).
>> cc(A, 0, 0).
A = 0.
```

La requête suivante indique que les intervalles fermés qui contiennent 0 ont une borne inférieure inférieure ou égale à 0 et une borne supérieure supérieure

ou égale à 0.

```
>> cc(0, B, C).
C ~ ge(0),
B ~ le(0).
```

La borne supérieure doit être supérieure ou égale à la borne inférieure.

```
>> cc(A, 0, C).
C ~ ge(0),
A ~ ge(0).
>> A ~ cc(0,2), B ~ cc(1,3), C = A .+. B .
C ~ cc(1,5),
B ~ cc(1,3),
A ~ cc(0,2).
```

Voir également : bcc/4, boutcc/4, co/3, oc/3, oc/3, oo/3, outcc/3.

ceil/2 — Plus petit entier supérieur

Définition : **ceil/2** définit la relation $\{(a,b) \in \mathbf{Z} \times \mathbf{R} \mid a = \lceil b \rceil\}$.

Description : **ceil/2** est une relation liant deux variables numériques, qui est traitée par le solveur sur les intervalles. Elle établit une relation entre un nombre et le plus petit entier qui lui est supérieur ou égal.

Exemples :

```
>> ceil(A, B).
B ~ real,
A ~ real.
>> ceil(A, 1.5).
A = 2.
>> ceil(A, 1).
A = 1.
>> ceil(A, -1.5) .
A = -1.
>> ceil(2, B).
B ~ oc(1,2).
>> set_prolog_flag(interval_mode,union).
true.
>> ceil(A, cc(1,2)).
A ~ 1 u 2.
>> ceil(A, oo(1,2)).
A = 2.
>> set_prolog_flag(interval_mode,union).
true.
```

Note : Etant donné qu'un des arguments de **ceil/2** est défini comme étant un entier, il est important que le domaine des variables soit restreint à une valeur raisonnable avant d'utiliser cette contrainte en mode union d'intervalles. Sinon, des problèmes de mémoire peuvent se produire. Voir **int/1** pour de plus amples détails.

Voir également : abs/2, floor/2, int/1.

co/3 — Appartenance à un intervalle

Définition : **co/3** définit la relation $\{(a,b,c) \in \mathbf{R}^3 \mid a \in [b,c)\}$.

Description : **co/3** est une relation sur les nombres réels, qui est traitée par le solveur sur les intervalles. Elle établit une relation entre un nombre et les bornes d'un intervalle fermé à gauche et ouvert à droite de la forme $[b,c)$ le contenant.

Exemples :

```
>> co(A, B, C).
C ~ real,
B ~ real,
A ~ real.
>> co(A, 1, 2).
A ~ co(1,2).
>> co(0, B, C).
C ~ gt(0),
B ~ le(0).
>> co(A, 0, 0).
false.
>> co(A, 0, C).
C ~ gt(0),
A ~ ge(0).
>> A ~ cc(0,2), B ~ co(1,3), C = A .+. B .
C ~ co(1,5),
B ~ co(1,3),
A ~ cc(0,2).
```

Note : Voir cc/3 pour une explication plus précise de certains exemples ci-dessus.

Voir également : bco/4, boutco/4, cc/3, oc/3, oo/3, outco/3.

conc/3
o — Concaténation de listes

Définition : **conc/3** définit la relation $\{(a,b,c) \in \mathbf{L}^3 \mid a = b \bullet c\}$.

Autre notation : **conc**(a,b,c) s'écrit aussi $a = b$ **o** c.

Description : **conc/3** est une relation sur les listes, qui est traitée par le solveur général. Elle établit une relation entre trois listes, dont la première est la concaténation des deux autres.

Exemples :

```
>> conc(A, B, C).
C ~ list,
B ~ list,
A ~ list.
>> A ~ conc([1,2,3],[4,5]).
A = [1,2,3,4,5].
>> [1,X,Y,4,5] ~ [Z,2] o [3|T] .
T = [4,5],
Z = 1,
Y = 3,
X = 2.
>> [1] o X = X o [1], size(X) = 10 .
X = [1,1,1,1,1,1,1,1,1,1].
```

Dans la requête suivante, il est impossible de conclure, car deux des listes impliquées dans la concaténation ont des longueurs inconnues.

```
>> [1|X] = [2,3] o Y .
Y ~ list,
X ~ list.
```

Dès que la longueur d'une deuxième liste est connue, l'échec est détecté.

```
>> [1|X] = [2,3] o Y , size(X) = 4 .
false.
```

Note : Comme les exemples ci-dessus le montrent, les contraintes **conc**(a, b, c) ne peuvent être résolues que quand les longueurs d'au moins deux des listes impliquées sont connues. Il convient donc de prendre des précautions en écrivant et en utilisant des programmes basés sur la concaténation, de manière à s'assurer que suffisamment d'informations sont connues.

Voir également : size/2, index/3.

cos/2 — Cosinus

Définition : **cos/2** définit la relation $\{(a, b) \in \mathbf{R}^2 \mid a = \cos b\}$.

Description : **cos/2** est une relation sur les réels, qui est traitée par le solveur sur les intervalles. Elle établit une relation entre une valeur numérique et son cosinus.

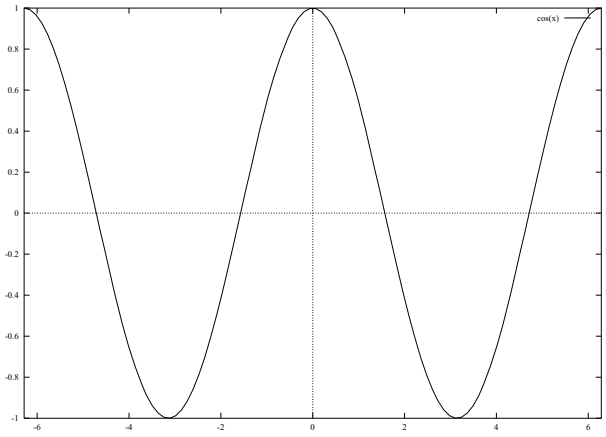
Exemples :

```
>> cos(A, B).
B ~ real,
A ~ cc(-1,1).
>> cos(A, 0).
A = 1.
>> A = 2.*.square(cos(pi./.4)).
A ~ oo(‘<0.9999998‘,‘>1.0000002‘).
>> A = 4.*.square(cos(pi./.3)).
A ~ oo(‘<0.9999999‘,‘>1.0000005‘).
>> cos(A, pi./.2).
A ~ oo(-‘>4.371139e-8‘,‘>7.54979e-8‘).
>> cos(A, A).
A ~ oo(‘>0.739085‘,‘>0.7390852‘).
```

Note : La fonction cosinus est périodique. Pour toute valeur possible $v \in [0, 1]$, il existe donc une infinité de $x \in \mathbf{R}$ tels que $\cos x = v$. Ceci pose un problème en mode unions d'intervalles, puisque certaines équations peuvent amener le système à créer une union d'intervalles trop importante pour tenir en mémoire. Il faut donc faire très attention en utilisant des relations représentant des fonctions périodiques, et il est nécessaire de borner les variables utilisées. Dans les exemples ci-dessous, on voit que l'ordre dans lequel les contraintes sont placées est très important dans les requêtes :

```
>> set_prolog_flag(interval_mode,union).
true.
>> cos(X) ~ 0.5, X ~ cc(0, 2 .*. pi).
error: heap_overflow
>> X ~ cc(0, 2 .*. pi), cos(X) ~ 0.5 .
X ~ cc('>1.0471974','>1.0471975')u oo('>5.235987','>5.235988').
>> set_prolog_flag(interval_mode,simple).
true.
```

Graphe :



Voir également : arccos/2, arcsin/2, arctan/2, cot/2, pi/2, sin/2, tan/2.

cosh/2 — Cosinus hyperbolique

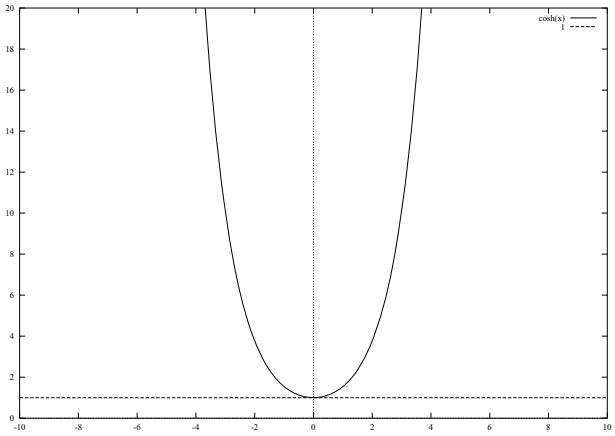
Définition : **cosh/2** définit la relation $\{(a,b) \in \mathbf{R}^2 \mid a = \cosh b\}$.

Description : **cosh/2** est une relation sur les réels, qui est traitée par le solveur sur les intervalles. Elle établit une relation entre une valeur numérique et son cosinus hyperbolique.

Exemples :

```
>> cosh(A, B).
B ~ real,
A ~ ge(1).
>> cosh(1, B).
B = 0.
>> cosh(A, 0).
A = 1.
>> cosh(A, 1).
A ~ cc('>1.5430805','>1.5430806').
>> cosh(A, 10).
A ~ cc('>11013.232','>11013.233').
>> cosh(A, 100).
A ~ ge('<4e38').
>> cosh(A, A).
A ~ ge('<4e38').
```

Graphe :



Voir également : coth/2, sinh/2, tanh/2.

cot/2 — Cotangente

Définition : **cot/2** définit la relation $\{(a, b) \in \mathbf{R}^2 \mid b \bmod \pi \neq 0 \text{ et } a = \cot b\}$.

Description : cot/2 est une relation sur les réels, qui est traitée par le solveur sur les intervalles. Elle établit une relation entre une valeur numérique et sa cotangente.

Exemples :

```
>> cot(A, B).
B ~ real,
A ~ real.
>> cot(0, B).
B ~ real.
>> cot(1, B).
B ~ real.
>> cot(A, 0).
false.
>> cot(A, pi./.4).
A ~ oo(`<0.9999995`,`>1.0000004`).
>> cot(A, pi./.2).
A ~ oo(-`>3.2584136e-7`,`<3.894144e-7`).
```

Il existe une infinité de solutions à l'équation $a = \cot a$:

```
>> cot(A, A).
A ~ real.
```

En restreignant l'espace de recherche entre 0 et 2π, on obtient une réduction, mais pas une solution (en effet, il reste deux solutions dans cet intervalle) :

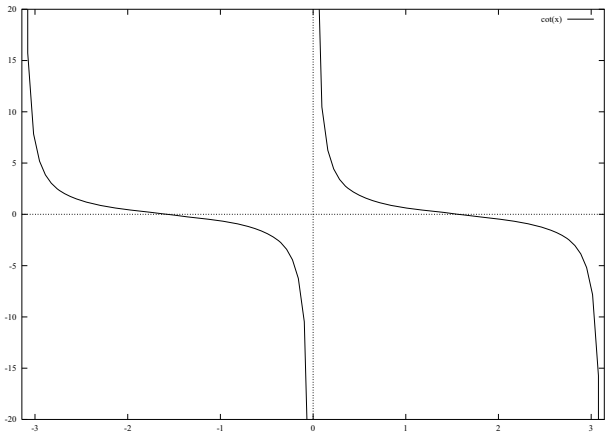
```
>> cot(A,A), A ~ oo(0,2.*.pi).
A ~ oo(`>0.21881127`,`>4.4969735`).
```

En restreignant l'espace de recherche entre 0 et π, on obtient une solution :

```
>> cot(A,A), A ~ oo(0, pi).
A ~ oo(`<0.8603334`,`>0.8603338`).
```

Note : Voir **cos/2** pour des remarques importantes concernant l'utilisation de relations associées à des fonctions périodiques en mode unions d'intervalles.

Graphe :



Voir également : arccos/2, arcsin/2, arctan/2, cos/2, pi/2, sin/2, tan/2.

coth/2 — Cotangente hyperbolique

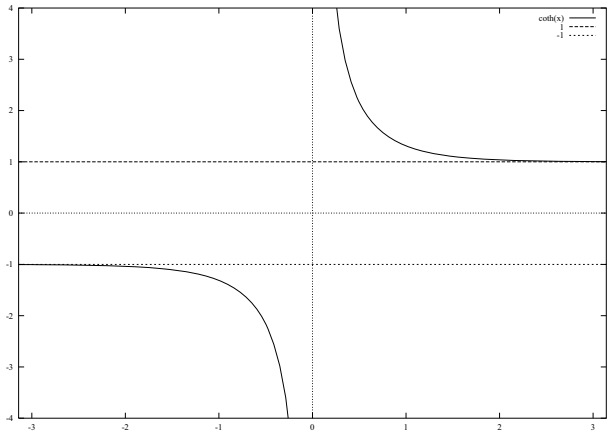
Définition : **coth/2** définit la relation $\{(a,b) \in \mathbf{R}^2 \mid b \neq 0 \text{ et } a = \coth b\}$.

Description : **coth/2** est une relation sur les réels, qui est traitée par le solveur sur les intervalles. Elle établit une relation entre une valeur numérique et sa cotangente hyperbolique.

Exemples :

```
>> coth(A, B).
B ~ real,
A ~ real.
>> ge(A, 0), coth(A, B).
B ~ gt(0),
A ~ gt(1).
>> le(A, 0), coth(A, B).
B ~ lt(0),
A ~ lt(-1).
>> coth(A, 1).
A ~ cc(‘>1.3130352‘,‘>1.3130353‘).
>> coth(A, 10).
A ~ oc(1,‘>1.0000001‘).
>> coth(A, A).
A ~ real.
```

Graphe :



Voir également : cosh/2, sinh/2, tanh/2.

dif/2 — Inégalité

Définition : **dif/2** définit la relation $\{(a, b) \in \mathbf{A}^2 \mid a \neq b\}$.

Description : **dif/2** est une relation générale entre des arbres quelconques. Elle établit une relation entre deux valeurs différentes. Une contrainte construite avec **dif/2** est souvent appelée une diséquation.

Exemples :

```
>> dif(A, B).
B ~ tree,
A ~ tree.
>> dif(alain, michel).
true.
>> dif(alain, alain).
false.
>> dif(f(X), f(Y)).
Y ~ tree,
X ~ tree.
>> dif(f(X+Y), f(X-Y)).
Y ~ real,
X ~ real.
>> dif(f(X+Y), f(X-Y)), Y = 0 .
false.
```

En mode «intervalles simples», la seule réduction possible d'un intervalle est l'exclusion d'une borne :

```
>> X ~ cc(1,2), dif(X,2).
X ~ co(1,2).
>> set_prolog_flag(interval_mode,union).
true.
>> dif(X,2).
X ~ tree.
```

En mode «union d'intervalles», des «trous» sont créés dans un intervalle :

```
>> X ~ cc(1,3), dif(X, 2).
X ~ co(1,2)u oc(2,3).
>> set_prolog_flag(interval_mode,simple).
true.
```

Note : La relation **dif/2** ayant un lien fort avec l'égalité, elle bénéficie en Prolog IV d'un traitement spécifique, qui permet de garantir son fonctionnement complet par rapport aux systèmes de contraintes constitués uniquement d'équations et de diséquations sur les arbres et de contraintes linéaires.

Voir également : eq/2, bdif/3, outlist/2.

div/3
./. Division

Définition : **div/3** définit la relation $\{(a, b, c) \in \mathbf{R}^3 \mid c \neq 0 \text{ et } a = b/c\}$.

Description : **div/3** est une relation sur les réels, qui est traitée par le solveur sur les intervalles. Elle établit une relation entre deux nombres et leur division.

Autre notation : **div**(a, b, c) s'écrit aussi $a = b$ **./.** c.

Exemples :

```
>> div(A, B, C).
C ~ real,
B ~ real,
A ~ real.
>> div(A, 1, 3) .
A = 1/3.
>> A = div(cc(1,2), cc(1,2)).
A ~ cc(0.5,2).
>> X ~ cc(0,1), A = cc(1,2) ./. X .
A ~ ge(1),
X ~ oc(0,1).
>> 1 ~ B ./. C .
B = C,
C ~ real.
```

Quand le domaine de la variable est l'ensemble des réels, la contrainte suivante ne réduit pas le domaine, et n'est donc pas résolue (nous avons en fait deux contraintes, dont une contrainte d'égalité) :

```
>> X = X ./. 2 .
X ~ real.
```

Par contre, si le domaine de la variable est plus petit, la réduction s'effectue correctement. On n'obtient toutefois pas un résultat exact, dans la mesure où l'équation est résolue par approximation et non par pur raisonnement :

```
>> X ~ cc(0,10), X = X ./. 2 .
X ~ cc(0,'>1e-45').
>> X ~ cc(-1,1), A = cc(1,2) ./. X .
A ~ real,
X ~ cc(-1,1).
```

En mode «unions d'intervalle», Les résultats sont parfois beaucoup plus précis ; on remarque entre autres l'exclusion de 0 pour le dénominateur :

```
>> set_prolog_flag(interval_mode, union).
true.
>> X ~ cc(-1,1), A = cc(1,2) ./. X .
A ~ le(-1)u ge(1),
X ~ co(-1,0)u oc(0,1).
>> set_prolog_flag(interval_mode,simple).
true.
```

Note : Quand deux des arguments de **div/3** sont réduits à des singletons, le troisième est calculé de manière exacte. Nous avons donc :

```
>> div(A, 1, 3 ) .
A = 1/3.
```

Il y a bien entendu une exception notable à cette règle :

```
>> div(0,0,A).
A ~ real.
```

Voir également : divlin/3, minus/3, plus/3, times/3, uminus/2, uplus/2.

divlin/3

/ Division

Définition : **divlin/3** définit la relation $\{(a,b,c) \in \mathbf{R}^3 \mid c \neq 0 \text{ et } a = b/c\}$.

Description : **divlin/3** est une relation sur les réels, qui est traitée par le solveur linéaire. Elle établit une relation entre deux nombres et leur division.

Autre notation : **divlin**(a,b,c) s'écrit aussi $a = b \,/\, c$.

Exemples :

```
>> divlin(A, B, C).
C ~ real,
B ~ real,
A ~ real.
>> A ~ divlin(1, 3).
A = 1/3.
>> divlin(1, B, C).
B = C,
C ~ real.
>> divlin(A, B, 1).
A = B,
B ~ real.
>> X = X/2 .
X = 0.
```

Note : La division «linéaire» est retardée jusqu'à ce qu'elle puisse être transformée en une contrainte linéaire. La requête suivante ne fixe donc pas la valeur de la variable `A` :

```
>> A ~ B / B.
B ~ real,
A ~ real.
```

Pour plus de détails sur le retardement de contraintes dans le solveur linéaire, voir **timeslin/3**.

Voir également : div/3, minuslin/3, pluslin/3, timeslin/3, uminuslin/2, upluslin/2.

eq/2

= Egalité

Définition : **eq/2** définit la relation $\{(a, b) \in \mathbf{A}^2 \mid a = b\}$.

Description : **eq/2** est une relation générale sur les arbres, qui est traitée par le solveur général. Elle établit une relation entre deux nombres égaux.

La relation **eq/2** est à la base de Prolog IV, et est utilisée de manière implicite lors de l'unification de deux termes, et en particulier lors de l'exécution de programmes.

Autre notation : **eq**(a, b) s'écrit aussi $a = b$.

Exemples :

```
>> eq(A, B).
A = B,
B ~ tree.
>> eq(alain, michel).
false.
>> eq(alain, alain).
true.
>> eq(f(X),f(Y)).
X = Y,
Y ~ tree.
>> eq(f(X+Y),f(X-Y)).
Y = 0,
X ~ real.
>> X ~ cc(1,2), Y ~ cc(2,3), eq(X,Y).
Y = 2,
X = 2.
```

Note : La relation **eq/2** est implantée de manière complète dans le solveur linéaire. Elle peut donc être résolue beaucoup plus efficacement dans ce solveur que dans le solveur sur les intervalles, comme le montrent les exemples ci-dessous :

```
>> eq(1+X, X).
false.
>> eq(1.+. X, X).
X ~ real.
```

Bien entendu, l'égalité sur les arbres est également traitée de manière complète, comme dans tout système Prolog.

```
>> [X,Y,Z,T] = [T,Z,X,Y].
Z = T,
Y = T,
X = T,
T ~ tree.
```

Voir également : beq/3, dif/2.

equiv/2 — Equivalence

Définition : **equiv/2** définit la relation $\{(a,b) \in \mathbf{B}^2 \mid a = b\}$.

Description : **equiv/2** est une relation sur les booléens, qui est traitée par le solveur sur les intervalles. Une contrainte **equiv**(a,b) établit une relation entre deux booléens identiques a et b.

Cette contrainte est normalement utilisée pour construire une combinaison d'autres contraintes exprimées par des pseudo-opérations produisant des booléens.

Autre notation : **equiv**(a,b) s'écrit aussi a **equiv** b.

Exemples :

```
>> equiv(A, B).
A = B,
B ~ cc(0,1).
>> equiv(1, B).
B = 1.
>> equiv(A, 0).
A = 0.
```

Voir également : and/2, bequiv/3, impl/2, not/1, or/2, xor/2.

exp/2 — Exponentielle

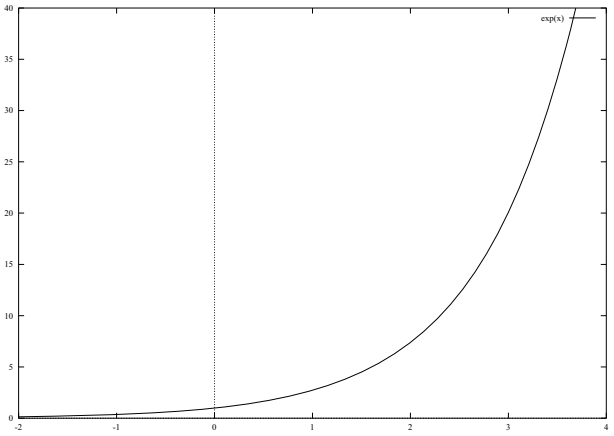
Définition : **exp/2** définit la relation $\{(a,b) \in \mathbf{R}^2 \mid a = \mathrm{e}^b\}$.

Description : **exp/2** est une relation sur les réels, qui est traitée par le solveur sur les intervalles. Elle établit une relation entre une valeur numérique x et la valeur e^x (où $e = 2.718281828459\ldots$).

Exemples :

```
>> exp(A, B).
B ~ real,
A ~ gt(0).
>> exp(1, B).
B = 0.
>> A ~ exp(-1).
A ~ cc(`>0.3678794`,`>0.36787945`).
>> exp(A, 0).
A = 1.
>> A ~ exp(1).
A ~ cc(`>2.7182817`,`<2.718282`).
>> exp(A, 2).
A ~ cc(`<7.389056`,`>7.389056`).
>> exp(A, 10).
A ~ cc(`>22026.464`,`>22026.466`).
>> A ~ exp(100).
A ~ ge(`<4e38`).
>> exp(A, A).
A ~ gt(`<4e38`).
>> 100 ~ exp(C).
C ~ cc(`<4.60517`,`>4.60517`).
```

Graphe :



Voir également : ln/2, log/2, power/3, root/3, square/2, sqrt/2.

finite/1 — Etre fini

Définition : **finite/1** définit la relation $\{a \in \mathbf{A} \mid a \text{ est fini}\}$.

Description : **finite/1** est une relation générale sur les arbres, qui est traitée par le solveur général. Une contrainte **finite**(a) est vérifiée quand son argument a représente un arbre fini.

finite/1 est une relation, et pas une primitive de vérification. Si une variable `X` est contrainte par `finite(X)`, elle ne peut plus s'unifier avec un arbre infini.

Exemples :

```
>> finite(A).
A ~ finite.
>> X = f(X), finite(X).
false.
>> finite([X|L]).
L ~ finite,
X ~ finite.
```

Voir également : bfinite/2, binfinite/2, infinite/1.

floor/2 — Plus grand entier inférieur

Définition : **floor/2** définit la relation $\{(a,b) \in \mathbf{Z} \times \mathbf{R} \mid a = \lfloor b \rfloor\}$.

Description : **floor/2** est une relation liant deux variables numériques, qui est traitée par le solveur sur les intervalles. Elle établit une relation entre un nombre et le plus grand entier qui lui est inférieur ou égal.

Exemples :

```
>> floor(A, B).
B ~ real,
A ~ real.
>> floor(A, 1.5).
A = 1.
>> floor(A, 1).
A = 1.
>> floor(A, -1.5).
A = -2.
>> floor(2, B).
B ~ co(2,3).
>> set_prolog_flag(interval_mode,union).
true.
>> floor(A, cc(1,2)).
A ~ 1 u 2.
>> floor(A, oo(1,2)).
A = 1.
>> set_prolog_flag(interval_mode,union).
true.
```

Note : Etant donné qu'un des arguments de **floor/2** est défini comme étant un entier, il est important que le domaine de cet argument soit restreint à une valeur raisonnable avant d'utiliser cette contrainte en mode union d'intervalles. Sinon, des problèmes de mémoire peuvent se produire. Voir **int/1** pour de plus amples détails.

Voir également : abs/2, ceil/2, int/1.

gcd/3 — Plus grand diviseur commun

Définition : **gcd/3** définit la relation $\{(a,b,c) \in \mathbf{Z}^3 \mid a \geq 0, b \geq 0, c \geq 0 \text{ et } a = \text{pgcd}(b,c)\}$.

Description : La relation **gcd/3** n'est pas implantée dans la version actuelle de Prolog IV. Toutefois, étant donné qu'elle fait partie de la définition du langage, le symbole a été réservé.

Exemples :

```
>> gcd(A, B, C).
Relation not yet implemented: gcd
false.
```

Voir également : intdiv/3, lcm/3.

ge/2 — Supérieur ou égal

Définition : **ge/2** définit la relation $\{(a,b) \in \mathbf{R}^2 \mid a \geq b\}$.

Description : **ge/2** est une relation sur les réels, qui est traitée par le solveur sur les intervalles. Elle établit une relation entre deux valeurs numériques telles que la première est supérieure ou égale à la seconde.

Exemples :

```
>> ge(A, B).
B ~ real,
A ~ real.
>> ge(A, 0).
A ~ ge(0).
>> A ~ cc(0,4), B ~ cc(1,3), ge(A, B).
B ~ cc(1,3),
A ~ cc(1,4).
```

Note : La relation **ge/2** apparaît souvent dans les réponses de Prolog IV, dans la mesure où un pseudo-terme **ge**(v) désigne en fait l'intervalle $[v, +\infty)$.

Voir également : bge/3, gelin/2, gt/2, le/2, lt/2.

gelin/2 — Supérieur ou égal

Définition : **gelin/2** définit la relation $\{(a,b) \in \mathbf{R}^2 \mid a \geq b\}$.

Description : **gelin/2** est une relation sur les réels, qui est traitée par le solveur linéaire. Elle établit une relation entre deux valeurs numériques telles que la première est supérieure ou égale à la seconde.

Exemples :

```
>> gelin(A, B).
B ~ real,
A ~ real.
>> gelin(A,B), gelin(B,A).
B ~ real,
A ~ real.
>> gelin(A, B), gelin(B, A), dif(A, B).
false.
```

Voir également : ge/2, gtlin/2, lelin/2, ltlin/2.

gt/2 — Supérieur

Définition : **gt/2** définit la relation $\{(a,b) \in \mathbf{R}^2 \mid a > b\}$.

Description : **gt/2** est une relation sur les réels, qui est traitée par le solveur sur les intervalles. Elle établit une relation entre deux valeurs numériques telles que la première est strictement supérieure à la seconde.

Exemples :

```
>> gt(A, B).
B ~ real,
A ~ real.
>> gt(A, 0).
A ~ gt(0).
>> A ~ cc(0,2), B ~ cc(1,3), gt(A,B).
B ~ co(1,2),
A ~ oc(1,2).
>> gt(A,B), gt(B,A).
B ~ real,
A ~ real.
```

Note : La relation **gt/2** apparaît souvent dans les réponses de Prolog IV, dans la mesure où un pseudo-terme **gt**(v) désigne en fait l'intervalle $(v, +\infty)$.

Voir également : bgt/3, ge/2, gtlin/2, le/2, lt/2.

gtlin/2 — Supérieur

Définition : **gtlin/2** définit la relation $\{(a,b) \in \mathbf{R}^2 \mid a > b\}$.

Description : **gtlin/2** est une relation sur les réels, qui est traitée par le solveur linéaire. Elle établit une relation entre deux valeurs numériques telles que la première est strictement supérieure à la seconde.

Exemples :

```
>> gtlin(A, B).
B ~ real,
A ~ real.
>> gtlin(A, B), gtlin(B, A).
false.
```

Voir également : gelin/2, gt/2, lelin/2, ltlin/2.

identifier/1 — Etre un identificateur

Définition : **identifier/1** définit la relation $\{a \in \mathbf{A} \mid a$ est un identificateur$\}$.

Description : **identifier/1** est une relation générale sur les arbres, qui est traitée par le solveur général. Une contrainte **identifier**(a) est vérifiée quand son argument a représente un identificateur.

identifier/1 est une relation, et pas une primitive de vérification, comme **atom/1**. Une contrainte `identifier(X)` peut donc être appliquée à une variable `X` dont la valeur est inconnue ; elle ne pourra alors plus s'unifier avec un arbre qui n'est pas un identificateur.

Exemples :

```
>> identifier(A).
A ~ identifier.
>> identifier(toto).
true.
>> identifier(cc(1,3)).
false.
>> X ~ list, identifier(X).
X = [].
```

Voir également : bidentifier/2, bidentifier/2, nidentifier/1.

if/4 — Si alors sinon

Définition : **if/4** définit la relation $\{(a, b, c, d) \in \mathbf{A} \times \mathbf{B} \times \mathbf{A}^2 \mid$ si $b = 1$ alors $a = c$ sinon $a = d\}$.

Description : if/4 est une relation liant un booléen et trois arbres quelconques. Quand le booléen est vrai (1), le premier arbre est égal au second ; sinon il est égal au troisième.

Cette contrainte est normalement utilisée pour construire des contraintes complexes dans lesquelles la vérification d'une contrainte conditionne la pose d'autres contraintes.

Exemples :

```
>> if(A, B, C, D).
D ~ tree,
C ~ tree,
A ~ tree,
B ~ cc(0,1).
>> if(A, B, 1, 2).
B ~ cc(0,1),
A ~ cc(1,2).
>> A = if(B, 1, 2), boolsplit([B]).
B = 0,
A = 2;
B = 1,
A = 1.
>> A ~ cc(1,3), A = if(B, cc(1,2), cc(4,5)).
B = 1,
A ~ cc(1,2).
```

Note : Attention, `if` est une relation et non une structure de contrôle. En particulier,

dans la requête suivante, le pseudo-terme `ntree` est transformé en contrainte et évalué, ce qui provoque l'échec de la requête.

```
>> A ~ if(1,tree,ntree).
false.
```

Cette requête est en fait équivalente à la requête suivante :

```
>> if(A,1,C,D), tree(C), ntree(D).
false.
```

On voit bien qu'une contrainte `ntree(D)` apparaît explicitement, ce qui provoque l'échec. En pratique, ce cas de figure se rencontre dans les requêtes du type :

```
>> A ~ if(bge(X,0), ln(X), ln(uminus(X))).
false.
```

Ici, le problème vient du fait que la contrainte implicite `ln(V,X)` contraint `X` à être strictement positif, alors que l'autre contrainte implicite le contraint à être négatif, ce qui déclenche l'échec.

impl/2 — Implication

Définition : **impl/2** définit la relation $\{(a,b) \in \mathbf{B}^2 \mid \text{ si } a = 1 \text{ alors } b = 1\}$.

Description : **impl/2** est une relation sur les booléens, qui est traitée par le solveur sur les intervalles. Une contrainte **impl**(a,b) réussit si son deuxième argument b est impliqué par son premier a.

Cette contrainte est normalement utilisée pour construire une combinaison d'autres contraintes exprimées par des pseudo-opérations produisant des booléens.

Autre notation : **impl**(a,b) s'écrit aussi a **impl** b.

Exemples :

```
>> impl(A, B).
B ~ cc(0,1),
A ~ cc(0,1).
>> impl(0, B).
B ~ cc(0,1).
>> impl(1, B).
B = 1.
>> impl(A, 0).
A = 0.
>> impl(A, 1).
A ~ cc(0,1).
```

Voir également : and/2, bimpl/3, equiv/2, not/1, or/2, xor/2.

index/3

: Nième élément d'une liste

Définition : **index/3** définit la relation $\{(a,b,c) \in \mathbf{A} \times \mathbf{L} \times \mathbf{Z} \mid 1 \leq c \leq |b| \text{ et } a \text{ est le } c^{\text{ième}} \text{ élément de } b\}$.

Description : **index/3** est une relation d'accès à une liste, qui est traitée par le solveur général. Une contrainte **index**(a,b,c) réussit si son troisième argument c est l'index de son premier argument a dans le deuxième b.

Autre notation : **index**(a,b,c) s'écrit aussi $a = b : c$.

Exemples :

```
>> index(A, B, C).
A ~ tree,
B ~ list,
C ~ ge(1).
>> index(A, [3,7,5,2], cc(2,3)).
A ~ cc(5,7).
>> index(A, [3,7,5,2], C).
C ~ cc(1,4),
A ~ cc(2,7).
>> A ~ [3,7,5,2]:2.
A = 7.
>> index(2, [3,7,5,2], X).
X = 4.
```

Dans la requête suivante, la pauvreté du résultat peut surprendre. Cela provient du fait que la variable `X` n'est pas typée comme variable numérique, et donc que la liste n'est pas exclusivement numérique. Or, en raison de la définition des sous-domaines d'approximation de Prolog IV, les algorithmes numériques ne sont déclenchés que sur les listes purement numériques :

```
>> index(2, [3,7,5,X], Y).
X ~ tree,
Y ~ cc(1,4).
```

Ici, on a simplement ajouté un type à la variable `X` :

```
>> index(2, [3,7,5,X], Y), X ~ real .
Y = 4,
X = 2.
>> set_prolog_flag(interval_mode,union).
true.
>> index(A, [3,7,5,2], C).
C ~ 1 u 2 u 3 u 4,
A ~ 2 u 3 u 5 u 7.
>> set_prolog_flag(interval_mode,simple).
true.
```

Note : Dans les exemples ci-dessus, nous avons constaté que les réductions effectuées par le traitement des contraintes construites avec `index/3` ne sont pas toujours simples à comprendre. Voici donc quelques remarques qui peuvent aider (on suppose ici qu'on a une contrainte `index(X,L,I)` :

- Pour que des réductions soient effectuées, il faut au moins que `I` soit connu ou que la longueur de `L` soit connue.
- Pour que des réductions numériques soient effectuées sur `X` et sur les éléments de `L`, il faut que tous les éléments de `L` soient contraints à être

des nombres réels.

– Pour trouver l'index d'un élément non-numérique d'une liste, il faut que tous les éléments de la liste soient connus.

En fait, toutes ces remarques proviennent de la définition des sous-domaines privilégiés de Prolog IV.

Note : Etant donné qu'un des arguments de **index/3** est défini comme étant un entier, il est important que le domaine de cet argument soit restreint à une valeur raisonnable avant d'utiliser cette contrainte en mode union d'intervalles. Sinon, des problèmes de mémoire peuvent se produire. Voir **int/1** pour de plus amples détails.

Voir également : conc/3, size/2.

infinite/1 — Etre infini

Définition : **infinite/1** définit la relation $\{a \in \mathbf{A} \mid a \text{ est infini}\}$.

Description : **infinite/1** est une relation générale sur les arbres. Une contrainte **infinite**(a) réussit si son argument a est un arbre infini.

infinite/1 est une relation, et pas une primitive de vérification. Si une variable `X` est contrainte par `infinite(X)`, elle ne peut plus s'unifier avec un arbre fini.

Exemples :

```
>> infinite(A).
A ~ infinite.
>> eq(X, f(X)), infinite(X).
X = f(X).
>> X ~ leaf n infinite.
false.
```

Voir également : bfinite/2, binfinite/2, finite/1.

inlist/2 — Appartenance à une liste

Définition : **inlist/2** définit la relation $\{(a, b) \in \mathbf{A} \times \mathbf{L} \mid a \text{ figure dans } b\}$.

Description : **inlist/2** est une relation sur les listes, qui est traitée par le solveur général. Une contrainte **inlist**(a, b) réussit si son premier argument a est un élément de son second argument b.

Exemples :

```
>> inlist(A, B).
A ~ tree,
B ~ list.
>> inlist(B, [1,5,3]).
B ~ cc(1,5).
>> X ~ cc(1,5), Y ~ cc(2,7), X ~ inlist([3,Y]).
Y ~ cc(2,7),
X ~ cc(2,5).
```

La requête suivante diffère de la précédente par le fait que 13 est disjoint du le domaine de X. Le système déduit alors que X doit être égale à l'unique autre élément de la liste, soit Y :

```
>> X ~ cc(1,5), Y ~ cc(2,7), X ~ inlist([13,Y]).
X = Y,
Y ~ cc(2,5).
>> inlist(B, []).
false.
>> set_prolog_flag(interval_mode,union).
true.
>> inlist(B, [1,5,3]).
B ~ 1 u 3 u 5.
>> set_prolog_flag(interval_mode,simple).
```

Note : Les réductions effectuées par **inlist/2** sont parfois assez complexes. Voici quelques remarques à ce sujet (en considérant que nous avons la contrainte `inlist(X,L)`) :

- Tant que la longueur de L n'est pas connue, aucune réduction n'est effectuée sur X, puisque tout arbre peut apparaître dans la partie inconnue de la liste.
- Pour que des réductions soient effectuées sur X, il faut également que tous les éléments de la liste soient contraints à être des nombres réels, soit que tous les éléments de la liste soient connus.
- Pour que des réductions soient effectuées sur un élément de L, il faut que tous les éléments de L soient contraints à être des nombres réels et que les domaines de tous les éléments de L sauf un (celui qui va être réduit) soient disjoints du domaine de X.

Les raisonnements sont similaires quand on travaille sur outlist/2, binlist/2 ou boutlist/2. Les mêmes conditions que ci-dessus doivent être respectées pour pouvoir faire des déductions, ce qui peut paraître plus gênant, comme le montrent les exemples ci-dessous :

```
>> A ~ binlist(1,[1,X]).
X ~ tree,
A ~ cc(0,1).
>> A ~ binlist(1,[1,real]).
A = 1.
>> outlist(toto,[toto,identifier]).
true.
>> outlist(toto,[toto,titi]).
false.
```

Voir également : binlist/3, boutlist/3, outlist/2.

int/1 — Etre un entier

Définition : **int/1** définit la relation $\{a \in \mathbf{R} \mid a \text{ est entier}\}$.

Description : **int/1** est une relation sur les réels, qui est traitée par le solveur sur les intervalles. **int/1** réussit quand son argument est un entier.

int/1 est une relation, et pas une primitive de vérification comme **integer/1**. Si une variable `X` est contrainte par `int(X)`, elle ne peut plus s'unifier avec un nombre non entier. Cette information est bien entendu par le solveur sur les intervalles pour effectuer des réductions de domaines.

Exemples :

```
>> int(A).
A ~ real.
>> int(1).
true.
>> A ~ cc(1,2), int(A).
A ~ cc(1,2).
>> A ~ cc(1/2,2), int(A).
A ~ cc(1,2).
>> A ~ oo(1,3), int(A).
A = 2.
>> set_prolog_flag(interval_mode,union).
true.
>> A ~ cc(1,2), int(A).
A ~ 1 u 2.
>> A ~ cc(1,10), int(A ./. 2).
A ~ 2 u 4 u 6 u 8 u 10.
```

La requête ci-dessous est équivalente à la précédente ; seule son expression change. La seconde contrainte exprime le fait qu'il doit exister un entier doit `A` soit le double :

```
>> A ~ cc(1,10), A ~ 2 .*. int .
A ~ 2 u 4 u 6 u 8 u 10.
>> set_prolog_flag(interval_mode,simple).
true.
```

Note : Si la relation **int/1** est appliquée à une variable dont le domaine n'a pas été réduit, une erreur en résultera probablement, à moins que votre machine ne dispose d'une très importante quantité de mémoire. Cela vient du fait que le système essaie de représenter en mémoire la liste de tous les entiers représentables par des rationnels IEEE :

```
>> set_prolog_flag(interval_mode,union).
true.
>> int(A).
error: heap_overflow
>> set_prolog_flag(interval_mode,simple).
true.
```

Quand on combine plusieurs contraintes, il faut donc faire attention à leur ordre, comme on le voit ci-dessous :

```
>> int(A), A ~ cc(1,10).
error: heap_overflow
>> A ~ cc(1,10), int(A).
A ~ 1 u 2 u 3 u 4 u 5 u 6 u 7 u 8 u 9 u 10.
```

Toutefois, en utilisant des pseudo-expressions, Prolog IV se charge de réor-

donner les contraintes de manière à éviter que des problèmes se produisent, si c'est possible.

```
>> A ~ cc(1,10) n int .
A ~ 1 u 2 u 3 u 4 u 5 u 6 u 7 u 8 u 9 u 10.
>> A ~ int n cc(1,10).
A ~ 1 u 2 u 3 u 4 u 5 u 6 u 7 u 8 u 9 u 10.
```

Des problèmes similaires peuvent se poser avec toutes les relations qui ont à voir avec des entiers, à savoir **bint/2**, **bnint/2**, **ceil/2**, **floor/2**, **index/3** et **nint/2**.

Note : L'interprétation de **int/1**, soit l'ensemble des entiers, n'est pas un sous-domaine privilégié de Prolog IV. En conséquence, les contraintes construites avec **int/1** ne sont pas reportées lors de l'écriture des solutions.

Voir également : bint/2, bnint/2, nint/1.

intdiv/3 — Division entière

Définition : **intdiv/3** définit la relation $\{(a, b, c) \in \mathbf{Z}^3 \mid$ il existe $d \in \mathbf{Z}$ tel que $b = ac + d$ et $c > d \geq 0\}$.

Description : La relation **intdiv/3** n'est pas implantée dans la version actuelle de Prolog IV. Toutefois, étant donné qu'elle fait partie de la définition du langage, le symbole a été réservé.

Exemples :

```
>> intdiv(A, B, C).
Relation not yet implemented: intdiv
false.
```

Voir également : gcd/3, lcm/3.

lcm/3 — Plus petit commun multiple

Définition : **lcm/3** définit la relation $\{(a, b, c) \in \mathbf{Z}^3 \mid a \geq 0, b \geq 0, c \geq 0$ et $a = \text{ppcm}(b, c)\}$.

Description : La relation **lcm/3** n'est pas implantée dans la version actuelle de Prolog IV. Toutefois, étant donné qu'elle fait partie de la définition du langage, le symbole a été réservé.

Exemples :

```
>> lcm(A, B, C).
Relation not yet implemented: lcm
false.
```

Voir également : gcd/3, intdiv/3.

le/2 — Inférieur ou égal

Définition : **le/2** définit la relation $\{(a,b) \in \mathbf{R}^2 \mid a \leq b\}$.

Description : **le/2** est une relation sur les réels, qui est traitée par le solveur sur les intervalles. Elle établit une relation entre deux valeurs numériques telles que la première est inférieure ou égale à la seconde.

Exemples :

```
>> le(A, B).
B ~ real,
A ~ real.
>> le(A, 0).
A ~ le(0).
>> A ~ cc(0,4), B ~ cc(1,3), le(A,B).
B ~ cc(1,3),
A ~ cc(0,3).
>> A ~ cc(0,4), B ~ cc(1,3), le(B,A).
B ~ cc(1,3),
A ~ cc(1,4).
```

Note : La relation **le/2** apparaît souvent dans les réponses de Prolog IV, dans la mesure où un pseudo-terme **le**(v) désigne en fait l'intervalle $(-\infty, v]$.

Voir également : ble/3, ge/2, gt/2, lelin/2, lt/2.

leaf/1 — Etre un arbre d'un nœud

Définition : **leaf/1** définit la relation $\{a \in \mathbf{A} \mid a$ est un arbre d'un nœud$\}$.

Description : **leaf/1** est une relation générale sur les arbres, qui est traitée par le solveur général. Une contrainte **leaf**(a) est vérifiée quand son argument a représente un arbre d'un seul nœud.

leaf/1 est une relation, et pas une primitive de vérification comme **atomic/1**. Si une variable `X` est contrainte par `leaf(X)`, elle ne peut plus s'unifier avec un arbre de plus d'un nœud.

Exemples :

```
>> leaf(A).
A ~ leaf.
>> leaf(1).
true.
>> leaf(cc(1,2)).
true.
>> leaf(toto(X)).
false.
>> list(L), leaf(L).
L = [].
```

Voir également : bleaf/2, bnleaf/2, nleaf/1.

lelin/2 — Inférieur ou égal

Définition : **lelin/2** définit la relation $\{(a,b) \in \mathbf{R}^2 \mid a \leq b\}$.

Description : **lelin/2** est une relation sur les réels, qui est traitée par le solveur linéaire. Elle établit une relation entre deux valeurs numériques telles que la première est inférieure ou égale à la seconde.

Exemples :

```
>> lelin(A, B).
B ~ real,
A ~ real.
>> lelin(A, B), lelin(B, A).
B = A,
A ~ real.
>> lelin(A, B), lelin(B, A), dif(A, B).
false.
```

Voir également : gelin/2, gtlin/2, le/2, ltlin/2.

list/1 — Etre une liste

Définition : **list/1** définit la relation $\{a \in \mathbf{A} \mid a$ est une liste$\}$.

Description : **list/1** est une relation générale sur les arbres, qui est traitée par le solveur général. Une contrainte **list**(a) est vérifiée quand son argument a représente une liste.

list/1 est une relation, et pas une primitive de vérification. Si une variable `X` est contrainte par `list(X)`, elle ne peut plus s'unifier avec un arbre qui n'est pas une liste.

Exemples :

```
>> list(A).
A ~ list.
>> list([1,2]).
true.
>> list(1).
false.
>> list([X|L]).
X ~ tree,
L ~ list.
>> leaf(X), list(X).
X = [].
```

Voir également : blist/2, bnlist/2, nlist/1.

ln/2 — Logarithme népérien

Définition : **ln/2** définit la relation $\{(a,b) \in \mathbf{R}^2 \mid b > 0 \text{ et } a = \ln b\}$.

Description : ln/2 est une relation sur les réels, qui est traitée par le solveur sur les intervalles. Elle établit une relation entre une valeur numérique et son logarithme naturel (de base e, où $e = 2.718281828459\ldots$).

Exemples :

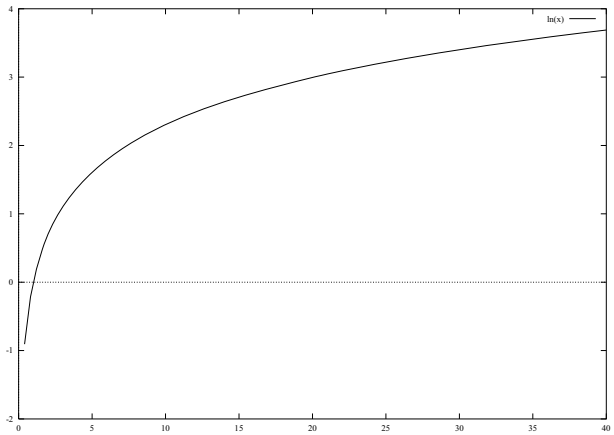
```
>> ln(A, B).
A ~ real,
B ~ gt(0).
>> ln(-1, B).
B ~ cc(‘>0.3678794‘,‘>0.36787945‘).
>> ln(0, B).
B = 1.
>> ln(1, B).
B ~ cc(‘>2.7182817‘,‘<2.718282‘).
>> ln(A, 1).
A = 0.
>> ln(A, 10).
A ~ cc(‘<2.302585‘,‘>2.302585‘).
>> ln(A, A).
A ~ gt(‘<4e38‘).
```

La contrainte $b > 0$ est appliquée au deuxième argument b de la contrainte **ln**(a,b). Ceci peut donc résulter en des échecs ou en des réductions de domaine :

```
>> ln(A, -2).
false.
>> B ~ cc(-2,2), A ~ ln(B).
A ~ le(‘<0.6931472‘),
B ~ oc(0,2).
```

Note : **ln/2** est ici défini comme une relation. Il n’y a donc pas de problème pour le traitement des logarithmes de nombres négatifs (voir exemples ci-dessus).

Graphe :



Voir également : exp/2, log/2, power/3, root/3, square/2, sqrt/2.

log/2 — Logarithme Décimal

Définition : **log/2** définit la relation $\{(a,b) \in \mathbf{R}^2 \mid b > 0 \text{ et } a = \log b\}$.

Description : log/2 est une relation sur les réels, qui est traitée par le solveur sur les intervalles. Elle établit une relation entre une valeur numérique et son logarithme de base 10.

Exemples :

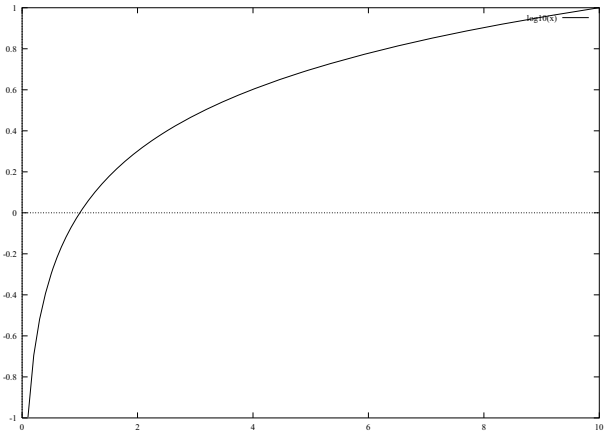
```
>> log(A, B).
A ~ real,
B ~ gt(0).
>> log(-1, B).
B = 1/10.
>> log(0, B).
B = 1.
>> log(1, B).
B = 10.
>> log(2, B).
B = 100.
>> log(A, 1).
A = 0.
>> log(A, 10).
A = 1.
>> log(A, 100).
A = 2.
>> log(A, A).
A ~ gt('<4e38').
```

La contrainte $b > 0$ est appliquée au deuxième argument b de la contrainte **log**(a,b). Ceci peut donc résulter en des échecs ou en des réductions de domaine :

```
>> log(A, -2).
false.
>> B ~ cc(-2,2), A ~ log(B).
A ~ le('>0.30103003'),
B ~ oc(0,2).
```

Note : **log/2** est ici défini comme une relation. Il n'y a donc pas de problème pour le traitement des logarithmes de nombres négatifs (voir exemples ci-dessus).

Graphe :



Voir également : exp/2, ln/2, power/3, root/3, square/2, sqrt/2.

lt/2 — Inférieur

Définition : **lt/2** définit la relation $\{(a,b) \in \mathbf{R}^2 \mid a < b\}$.

Description : **lt/2** est une relation sur les réels, qui est traitée par le solveur sur les intervalles. Elle établit une relation entre deux valeurs numériques telles que la première est strictement inférieure à la seconde.

Exemples :

```
>> lt(A, B).
B ~ real,
A ~ real.
>> lt(A, 0).
A ~ lt(0).
>> A ~ cc(0,4), B ~ cc(1,3), lt(A, B).
B ~ cc(1,3),
A ~ co(0,3).
>> A ~ cc(0,4), B ~ cc(1,3), lt(B,A).
B ~ cc(1,3),
A ~ oc(1,4).
```

La relation **lt/2** apparaît souvent dans les réponses de Prolog IV, dans la mesure où un pseudo-terme **lt**(v) désigne en fait l'intervalle $(-\infty, v)$.

Voir également : blt/3, ge/2, gt/2, le/2, ltlin/2.

ltlin/2 — Inférieur

Définition : **ltlin/2** définit la relation $\{(a,b) \in \mathbf{R}^2 \mid a < b\}$.

Description : **ltlin/2** est une relation sur les réels, qui est traitée par le solveur linéaire. Elle établit une relation entre deux valeurs numériques telles que la première est strictement inférieure à la seconde.

Exemples :

```
>> ltlin(A, B).
B ~ real,
A ~ real.
>> ltlin(A, B), ltlin(B, A).
false.
```

Voir également : gelin/2, gtlin/2, lelin/2, lt/2.

max/3 — Maximum

Définition : **max/3** définit la relation $\{(a,b,c) \in \mathbf{R}^3 \mid a = \max(b,c)\}$.

Description : **max/3** est une relation sur les réels, qui est traitée par le solveur sur les intervalles. Elle établit une relation entre deux valeurs et leur maximum.

Exemples :

```
>> max(A, B, C).
C ~ real,
B ~ real,
A ~ real.
>> A ~ max(cc(1,3), cc(2,4)).
A ~ cc(2,4).
>> A ~ max(0, C).
C ~ real,
A ~ ge(0).
>> max(0, B, C).
C ~ le(0),
B ~ le(0).
```

Voir également : min/3.

min/3 — Minimum

Définition : **min/3** définit la relation $\{(a,b,c) \in \mathbf{R}^3 \mid a = \min(b,c)\}$.

Description : **min/3** est une relation sur les réels, qui est traitée par le solveur sur les intervalles. Elle établit une relation entre deux valeurs et leur minimum.

Exemples :

```
>> min(A, B, C).
C ~ real,
B ~ real,
A ~ real.
>> min(0, B, C).
C ~ ge(0),
B ~ ge(0).
>> A ~ min(cc(1,3), cc(2,4)).
A ~ cc(1,3).
>> A ~ min(0, C).
C ~ real,
A ~ le(0).
```

Voir également : max/3.

minus/3

.-. Soustraction

Définition : **minus/3** définit la relation $\{(a,b,c) \in \mathbf{R}^3 \mid a = b - c\}$.

Description : **minus/3** est une relation sur les réels, qui est traitée par le solveur sur les intervalles. Elle établit une relation entre deux nombres et leur différence.

Autre notation : **minus**(a,b,c) s'écrit aussi $a = b$ **.-.** c.

Exemples :

```
>> minus(A, B, C).
C ~ real,
B ~ real,
A ~ real.
>> A ~ minus(3, 1).
A = 2.
>> A ~ cc(1,4) .-. cc(1,2).
A ~ cc(-1,3).
>> A ~ A .-. 1 .
A ~ real.
>> minus(0, B, C).
B = C,
C ~ real.
```

Note : Quand deux des arguments de **minus/3** sont réduits à des singletons, le troisième est calculé de manière exacte. Nous avons donc :

```
>> A ~ 2/3 .-. 1/3 .
A ~ 1/3.
```

Voir également : div/3, minuslin/3, plus/3, times/3, uminus/2, uplus/2.

minuslin/3

\- Soustraction

Définition : **minuslin/3** définit la relation $\{(a,b,c) \in \mathbf{R}^3 \mid a = b - c\}$.

Description : **minuslin/2** est une relation sur les réels, qui est traitée par le solveur linéaire. Elle établit une relation entre deux nombres et leur différence.

Autre notation : **minuslin**(a,b,c) s'écrit aussi $a = b$ **-** c.

Exemples :

```
>> minuslin(A, B, C).
C ~ real,
B ~ real,
A ~ real.
>> A ~ minuslin(3, 1).
A = 2.
>> A ~ A - 1 .
false.
>> minuslin(0, B, C).
B = C,
C ~ real.
```

Voir également : divlin/3, minus/3, pluslin/3, timeslin/3, uminuslin/2, upluslin/2.

mod/3 — Modulo

Définition : **mod/3** définit la relation $\{(a,b,c) \in \mathbf{R}^3 \mid 0 \leq a < c \text{ et } a = b \bmod c\}$.

Description : La relation **mod/3** n'est pas implantée dans la version actuelle de Prolog IV. Toutefois, étant donné qu'elle fait partie de la définition du langage, le symbole a été réservé.

Exemples :

```
>> mod(A, B, C).
Relation not yet implemented: mod
false.
```

Voir également : intdiv/3

n/3 — Intersection

Définition : **n/3** définit la relation $\{(a,b,c) \in \mathbf{A}^3 \mid a = b \text{ et } a = c\}$.

Description : **n/3** est une relation générale sur les arbres. Une contrainte $\mathbf{n}(a,b,c)$ est vérifiée si ses trois arguments sont égaux.

Le nom de **n/3** a été choisi car il ressemble au symbole de l'intersection. C'est en effet l'usage le plus courant de cette relation, au niveau des pseudo-termes. Se reporter à la note après les exemples pour plus d'explications à ce sujet.

Autre notation : $\mathbf{n}(a,b,c)$ s'écrit aussi $a = b \ \mathbf{n}\ c$.

Exemples :

```
>> n(A, B, C).
B = C,
A = C,
C ~ tree.
>> n(A, cc(1,3), cc(2,4)).
A ~ cc(2,3).
>> A ~ int n cc(1,5) .
A ~ cc(1,5).
>> set_prolog_flag(interval_mode,union).
true.
>> A ~ int n cc(1,5) .
A ~ 1 u 2 u 3 u 4 u 5.
>> A ~ int n cc(1,10) n 2 .*. nint .
A ~ 1 u 3 u 5 u 7 u 9.
>> set_prolog_flag(interval_mode,simple).
true.
```

Note : Attention, «n» n'est pas l'intersection, puisque Prolog IV ne raisonne que sur les nombres, et non sur les ensembles. La requête ci-dessous donne le résultat prévu par la définition de la relation, mais n'est bien sûr pas compatible avec ce qu'on attendrait d'une intersection :

```
>> cc(1,5) ~ cc(1,5) n A .
A ~ cc(1,5).
```

Considérons un autre exemple. Quand on utilise «n» en tant que pseudo-terme, on pense immédiatement à l'intersection, comme par exemple dans la requête suivante :

```
>> set_prolog_flag(interval_mode,union).
true.
>> X ~ cc(1,10) n int n dif(5).
X ~ 1 u 2 u 3 u 4 u 6 u 7 u 8 u 9 u 10.
```

En décomposant cette pseudo-expression, on obtient :

```
>> cc(A,1,10), n(C,A,B), int(B), dif(D,5), n(X,C,D).
D = X,
C = X,
B = X,
A = X,
X ~ cc(1,10).
```

Dans ce résultat, on voit bien les variables toutes égales.

Voir également : u/3.

nidentifier/1 — Ne pas être un identificateur

Définition : **nidentifier/1** définit la relation $\{a \in \mathbf{A} \mid a$ n'est pas un identificateur$\}$.

Description : **nidentifier/1** est une relation générale sur les arbres, qui est traitée par le solveur général. Une contrainte **nidentifier**(a) est vérifiée quand son argument a représente un arbre qui n'est pas un identificateur.

nidentifier/1 est une relation, et pas une primitive de vérification. Si une variable `X` est contrainte par `nidentifier(X)`, elle ne peut plus s'unifier avec un identificateur.

Exemples :

```
>> nidentifier(A).
A ~ nidentifier.
>> nidentifier(toto).
false.
>> nidentifier(cc(1,2)).
true.
>> list(X), nidentifier(X).
X ~ [tree|list].
```

Voir également : bidentifier/2, bnidentifier/2, identifier/1.

nint/1 — Ne pas être un entier

Définition : **nint/1** définit la relation $\{a \in \mathbf{R} \mid a$ n'est pas entier$\}$.

Description : **nint/1** est une relation sur les réels, qui est traitée par le solveur sur les intervalles. **nint/1** réussit quand son argument est un nombre non entier.

L'interprétation de **nint/1** est définie sur l'ensemble des réels. Cette relation échoue si son argument prend une valeur non numérique.

nint/1 est une relation, et pas une primitive de vérification. Si une variable `X` est contrainte par `nint(X)`, elle ne peut plus s'unifier avec un nombre entier.

Exemples :

```
>> nint(A).
A ~ real.
>> nint(1.5).
true.
>> A ~ cc(1,3) n nint .
A ~ oo(1,3).
>> set_prolog_flag(interval_mode,union).
true.
>> A ~ cc(1,3) n nint .
A ~ oo(1,2)u oo(2,3).
>> A ~ cc(1,10) n 3 .*. nint .
A ~ co(1,3)u oo(3,6)u oo(6,9)u oc(9,10).
>> A ~ cc(1,10) n int n 3 .*. nint .
A ~ 1 u 2 u 4 u 5 u 7 u 8 u 10.
>> set_prolog_flag(interval_mode,simple).
true.
```

Note : L'interprétation de **nint/1**, soit l'ensemble des réels non entiers, n'est pas un sous-domaine privilégié de Prolog IV. En conséquence, les contraintes **nint/1** ne sont pas reportées lors de l'écriture des solutions.

Voir également : bint/2, bnint/2, bnprime/2, bprime/2, int/1, nprime/1, prime/1.

nleaf/1 — Etre un arbre de plus d'un nœud

Définition : **nleaf/1** définit la relation $\{a \in \mathbf{A} \mid a$ est un arbre de plus d'un nœud$\}$.

Description : **nleaf/1** est une relation générale sur les arbres, qui est traitée par le solveur général. Une contrainte **nleaf**(a) est vérifiée quand son argument a représente un arbre de plus d'un nœud.

nleaf/1 est une relation, et pas une primitive de vérification comme **compound/1**. Si une variable `X` est contrainte par `nleaf(X)`, elle ne peut plus s'unifier avec un arbre qui n'est pas un identificateur.

Exemples :

```
>> nleaf(A).
A ~ nleaf.
>> nleaf(2).
false.
>> nleaf(f(X)).
X ~ tree.
>> list(L), nleaf(X).
L ~ list,
X ~ nleaf.
```

Voir également : bleaf/2, bnleaf/2, leaf/1

nlist/1 — Ne pas être une liste

Définition : **nlist/1** définit la relation $\{a \in \mathbf{A} \mid a$ n'est pas une liste$\}$.

Description : **nlist/1** est une relation générale sur les arbres, qui est traitée par le solveur général. Une contrainte **nlist**(a) est vérifiée quand son argument a ne représente pas une liste.

nlist/1 est une relation, et pas une primitive de vérification. Si une variable `X` est contrainte par `nlist(X)`, elle ne peut plus s'unifier avec une liste.

Exemples :

```
>> nlist(X).
X ~ nlist.
>> nlist([X|Y]).
X ~ tree,
Y ~ nlist.
>> nlist(cc(1,2)).
true.
>> nlist([1,2,3]).
false.
```

Voir également : blist/2, bnlist/2, list/1.

not/1 — Négation

Définition : **not/1** définit la relation $\{a \in \mathbf{B} \mid a = 0\}$.

Description : **not/1** est une relation sur les booléens, qui est traitée par le solveur sur les intervalles. Une contrainte **not**(a) réussit si son argument a représente la valeur faux (0).

Une telle contrainte est normalement utilisée pour construire une négation d'autres contraintes exprimées par des pseudo-opérations produisant des booléens.

Attention, **not/1** est ici une relation, et n'a pas de relation avec le prédicat **not/1** (ou $\backslash+$ fourni par certains systèmes Prolog (dont Prolog II+ et Prolog III), qui implémente généralement la négation par échec.

Exemples :

```
>> not(A).
A = 0.
>> not(bor(A,B)).
B = 0,
A = 0.
>> not(bnot(A)).
A = 1.
>> not(band(A,B)).
B ~ cc(0,1),
A ~ cc(0,1).
```

Voir également : and/2, bnot/2, equiv/2, impl/2, or/2, xor/2.

nprime/1 — Ne pas être un entier premier

Définition : **nprime/1** définit la relation $\{a \in \mathbf{R} \mid a$ n'est pas un entier premier$\}$.

Description : La relation **nprime/1** n'est pas implantée dans la version actuelle de Prolog IV. Toutefois, étant donné qu'elle fait partie de la définition du langage, le symbole a été réservé.

Exemples :

```
>> nprime(A).
Relation not yet implemented: nprime
false.
```

Voir également : bint/2, bnint/2, bnprime/2, bprime/2, int/1, nint/1, prime/1.

nreal/1 — Ne pas être un nombre réel

Définition : **nreal/1** définit la relation $\{a \in \mathbf{A} \mid a$ n'est pas un nombre réel$\}$.

Description : **nreal/1** est une relation générale sur les arbres, qui est traitée par le solveur général. Une contrainte **nreal**(a) est vérifiée quand son argument a ne représente pas un nombre réel.

nreal/1 est une relation, et pas une primitive de vérification. Si une variable `X` est contrainte par `nreal(X)`, elle ne peut plus s'unifier avec un arbre qui n'est pas un nombre réel.

Exemples :

```
>> nreal(A).
A ~ nreal.
>> nreal(toto).
true.
>> nreal(1).
false.
>> nreal(cc(1,2)).
false.
```

Voir également : breal/2, bnreal/2, real/1.

ntree/1 — Ne pas être un arbre

Définition : **ntree/1** définit la relation $\{a \in \mathbf{A} \mid a$ ne peut exister$\}$.

Description : **ntree/1** est une relation générale dont l'interprétation est l'ensemble vide. Elle ne réussit donc jamais, quelque soit l'argument fourni.

Exemples :

```
>> ntree(A).
false.
>> ntree(toto).
false.
>> ntree(1).
false.
```

Voir également : tree/1.

oc/3 — Appartenance à un intervalle

Définition : **oc/3** définit la relation $\{(a, b, c) \in \mathbf{R}^3 \mid a \in (b, c]\}$.

Description : **oc/3** est une relation sur les nombres réels, qui est traitée par le solveur sur les intervalles. Elle établit une relation entre un nombre et les bornes d'un intervalle ouvert à gauche et fermé à droite de la forme $(b, c]$ le contenant.

Exemples :

```
>> oc(A, B, C).
C ~ real,
B ~ real,
A ~ real.
>> oc(A, 1, 2).
A ~ oc(1,2).
>> oc(0, B, C).
C ~ ge(0),
B ~ lt(0).
>> oc(A, 0, 0).
false.
>> oc(A, 0, C).
C ~ gt(0),
A ~ gt(0).
>> A ~ cc(0,2), B ~ oc(1,3), C = A .+. B .
C ~ oc(1,5),
B ~ oc(1,3),
A ~ cc(0,2).
```

Note : Voir **cc/3** pour de plus amples explications sur les exemples ci-dessus.

Voir également : boc/4, boutoc/4, cc/3, co/3, oo/3, outoc/3.

oo/3 — Appartenance à un intervalle

Définition : **oo/3** définit la relation $\{(a, b, c) \in \mathbf{R}^3 \mid a \in (b, c)\}$.

Description : **oo/3** est une relation sur les nombres réels, qui est traitée par le solveur sur les intervalles. Elle établit une relation entre un nombre et les bornes d'un intervalle ouvert des deux côtés de la forme (b, c) le contenant.

Exemples :

```
>> oo(A, B, C).
C ~ real,
B ~ real,
A ~ real.
>> oo(A, 1, 2).
A ~ oo(1,2).
>> oo(0, B, C).
C ~ gt(0),
B ~ lt(0).
>> oo(A, 0, 0).
false.
>> oo(A, 0, C).
C ~ gt(0),
A ~ gt(0).
>> A ~ cc(0,2), B ~ oo(1,3), C = A .+. B .
C ~ oo(1,5),
B ~ oo(1,3),
A ~ cc(0,2).
```

Note : Voir **cc/3** pour de plus amples explications sur les exemples ci-dessus.

Voir également : boo/4, boutoo/4, cc/3, co/3, oc/3, outoo/3.

or/2 — Ou

Définition : **or/2** définit la relation $\{(a, b) \in \mathbf{B}^2 \mid a = 1 \text{ ou } b = 1\}$.

Description : **or/2** est une relation sur les booléens, qui est traitée par le solveur sur les intervalles. Une contrainte **or**(a, b) réussit si un de ses deux arguments vaut 1.

Cette contrainte est normalement utilisée pour construire une combinaison d'autres contraintes exprimées par des pseudo-opérations produisant des booléens.

Autre notation : **or**(a, b) s'écrit aussi a **or** b.

Exemples :

```
>> or(A, B).
B ~ cc(0,1),
A ~ cc(0,1).
>> or(0, B).
B = 1.
>> or(1, B).
B ~ cc(0,1).
```

Voir également : and/2, band/3, bequiv/3, bimpl/3, bnot/2, bor/3, bxor/3, equiv/2, impl/2, not/1, xor/2.

outcc/3 — Non-appartenance à un intervalle

Définition : **outcc/3** définit la relation $\{(a, b, c) \in \mathbf{R}^3 \mid a \notin [b, c]\}$.

Description : **outcc/3** est une relation sur les nombres réels, qui est traitée par le solveur sur les intervalles. Elle établit une relation entre un nombre et les bornes d'un intervalle fermé des deux côtés de la forme $[b, c]$ ne le contenant pas.

Exemples :

```
>> outcc(A, B, C).
C ~ real,
B ~ real,
A ~ real.
>> outcc(A, 1, 2).
A ~ real.
>> A ~ outcc(1,3) n cc(1,4).
A ~ oc(3,4).
>> set_prolog_flag(interval_mode,union).
true.
>> outcc(A, 1, 2).
A ~ lt(1)u gt(2).
>> A ~ outcc(1,3) n cc(1,4).
A ~ oc(3,4).
>> set_prolog_flag(interval_mode,simple).
true.
```

Note : Comme on l'a vu dans les exemples, il est courant en mode intervalles simples que la pose d'une contrainte **outcc**(a, b, c) n'effectue pas de réduction immédiate. Toutefois, ces contraintes, comme celles construites avec **dif/2** sont avant tout destinées à être dormantes jusqu'à ce que les domaines des variables soient suffisamment réduits.

Voir également : bcc/4, boutcc/4, cc/3, outco/3, outoc/3, outoo/3.

outco/3 — Non-appartenance à un intervalle

Définition : **outco/3** définit la relation $\{(a,b,c) \in \mathbf{R}^3 \mid a \notin [b,c)\}$.

Description : **outco/3** est une relation sur les nombres réels, qui est traitée par le solveur sur les intervalles. Elle établit une relation entre un nombre et les bornes d'un intervalle fermé à gauche et ouvert à droite de la forme $[b,c)$ ne le contenant pas.

Exemples :

```
>> outco(A, B, C).
C ~ real,
B ~ real,
A ~ real.
>> outco(A, 1, 2).
A ~ real.
>> A ~ outco(1,3) n cc(1,4) .
A ~ cc(3,4).
>> set_prolog_flag(interval_mode,union).
true.
>> outco(A, 1, 2).
A ~ lt(1)u ge(2).
>> A ~ outco(1,3) n cc(1,4) .
A ~ cc(3,4).
>> set_prolog_flag(interval_mode,simple).
true.
```

Voir également : bco/4, boutco/4, co/3, outcc/3, outoc/3, outoo/3.

outlist/2 — Non-appartenance à une liste

Définition : **outlist/2** définit la relation $\{(a,b) \in \mathbf{A} \times \mathbf{L} \mid a$ ne figure pas dans $b\}$.

Description : **outlist/2** est une relation sur les listes, qui est traitée par le solveur général. Une contrainte **outlist**(a,b) réussit si son premier argument a n'est pas un élément de son second argument b.

Exemples :

```
>> outlist(A, B).
A ~ tree,
B ~ list.
>> outlist(A, []).
A ~ tree.
>> outlist(A,[1,2,3]).
A ~ real.
```

Attention, ces contraintes sont évaluées par approximation. Avec des valeurs non numériques, il est donc indispensable que la liste soit entièrement connue pour avoir des résultats satisfaisants :

```
>> outlist(toto, [X,toto]).
X ~ tree.
```

Il en est de même si certaines des variables ne sont pas typées numériques :

```
>> outlist(1,[_,_,1,_]).
true.
```

Par contre on obtient le résultat attendu dès que les variables sont typées, même si leurs valeurs sont inconnues (ceci est dû au fait que les produits cartésiens d'intervalles sont des sous-domaines utilisés dans l'approximation de Prolog IV) :

```
>> outlist(1,[real,real,1,real]).
false.
>> set_prolog_flag(interval_mode,union).
true.
>> outlist(A,[1,2,3]).
A ~ lt(1)u oo(1,2)u oo(2,3)u gt(3).
>> set_prolog_flag(interval_mode,simple).
true.
```

Note : Voir **inlist/2** pour des remarques importantes concernant le fonctionnement opérationnel de cette contrainte.

Voir également : binlist/3, boutlist/3, intlist/2.

outoc/3 — Non-appartenance à un intervalle

Définition : **outoc/3** définit la relation $\{(a,b,c) \in \mathbf{R}^3 \mid a \notin (b,c]\}$.

Description : **outoc/3** est une relation sur les nombres réels, qui est traitée par le solveur sur les intervalles. Elle établit une relation entre un nombre et les bornes d'un intervalle ouvert à gauche et fermé à droite de la forme $(b,c]$ ne le contenant pas.

Exemples :

```
>> outoc(A, B, C).
C ~ real,
B ~ real,
A ~ real.
>> outoc(A, 1, 2).
A ~ real.
>> A ~ outoc(1,3) n cc(2,4) .
A ~ oc(3,4).
>> A ~ outoc(1,3) n cc(1,4) .
A ~ cc(1,4).
>> set_prolog_flag(interval_mode,union).
true.
>> outoc(A, 1, 2).
A ~ le(1)u gt(2).
>> A ~ outoc(1,3) n cc(1,4) .
A ~ 1 u oc(3,4).
>> set_prolog_flag(interval_mode,simple).
true.
```

Voir également : boc/4, boutoc/4, oc/3, outcc/3, outco/3, outoo/3.

outoo/3 — Non-appartenance à un intervalle

Définition : **outoo/3** définit la relation $\{(a,b,c) \in \mathbf{R}^3 \mid a \notin (b,c)\}$.

Description : **outoo/3** est une relation sur les nombres réels, qui est traitée par le solveur sur les intervalles. Elle établit une relation entre un nombre et les bornes d'un intervalle ouvert des deux côtés de la forme (b,c) ne le contenant pas.

Exemples :

```
>> outoo(A, B, C).
C ~ real,
B ~ real,
A ~ real.
>> outoo(A, 1, 2).
A ~ real.
>> A ~ outoo(1,3) n cc(2,4) .
A ~ cc(3,4).
>> A ~ outoo(1,3) n cc(1,4) .
A ~ cc(1,4).
>> set_prolog_flag(interval_mode,union).
true.
>> outoo(A, 1, 2).
A ~ le(1)u ge(2).
>> A ~ outoo(1,3) n cc(1,4) .
A ~ 1 u cc(3,4).
```

Voir également : boo/4, boutoo/4, oo/3, outcc/3, outco/3, outoc/3.

pi/1 — π

Définition : **pi/1** définit la relation $\{a \in \mathbf{R} \mid a = \pi\}$.

Description : **pi/1** est une relation sur les réels qui est vraie si son argument prend la valeur π.

Etant donné que π n'est pas représentable avec les rationnels IEEE disponibles en Prolog IV, la meilleure approximation possible est utilisée, c'est-à-dire l'intervalle ouvert des deux côtés dont les bornes inférieures et supérieures sont respectivement le plus grand rationnel IEEE inférieur à π et le plus petit rationnel IEEE supérieur à π.

Exemples :

```
>> pi(A).
A ~ oo(`>3.1415925`,`>3.1415927`).
>> A ~ sin(pi).
A ~ oo(-`>1.6292068e-7`,`<1.947072e-7`).
```

Il n'est pas possible d'unifier π avec une constante en Prolog IV, car le langage sait qu'il n'existe pas de représentation rationnelle de π ; or toutes les constantes du langage sont rationnelles :

```
>> pi(3.141592653589793238).
false.
```

Une erreur classique est d'utiliser `-pi`. Ceci ne fonctionne pas, car ici, `pi` est un pseudo-terme, utilisé pour poser une contrainte dans le solveur des intervalles. Etant donné que « – » est le raccourci pour la négation unaire du

solveur linéaire, la communication ne s'effectue pas entre les solveurs et nous obtenons la requête suivante (au lieu du résultat correct donné en-dessous) :

```
>> A = -pi.
A ~ real.
>> A ~ .-. pi .
A ~ oo(-'>3.1415927',-'>3.1415925').
```

Voir également : arccos/2, arcsin/2, arctan/2, cos/2, cot/2, sin/2, tan/2.

plus/3
.+. — Addition

Définition : **plus/3** définit la relation $\{(a,b,c) \in \mathbf{R}^3 \mid a = b + c\}$.

Description : **plus/3** est une relation sur les réels, qui est traitée par le solveur sur les intervalles. Elle établit une relation entre deux nombres et leur somme.

Autre notation : **plus**(a,b,c) s'écrit aussi $a = b$ **.+.** c.

Exemples :

```
>> plus(A, B, C).
C ~ real,
B ~ real,
A ~ real.
>> A ~ plus(3, 1).
A = 4.
>> A ~ cc(1,4) .+. cc(1,2).
A ~ cc(2,6).
>> A ~ A .+. 1 .
A ~ real.
>> plus(A, B, 0).
A = B,
B ~ real.
```

Note : Quand deux des arguments de **plus/3** sont réduits à des singletons, le troisième est calculé de manière exacte. Nous avons donc :

```
>> A ~ 1/3 .+. 1/3 .
A ~ 2/3.
```

Voir également : div/3 minus/3, pluslin/3, times/3, uminus/2, uplus/2.

pluslin/3
+ — Addition

Définition : **pluslin/3** définit la relation $\{(a,b,c) \in \mathbf{R}^3 \mid a = b + c\}$.

Description : **pluslin/2** est une relation sur les réels, qui est traitée par le solveur linéaire. Elle établit une relation entre deux nombres et leur somme.

Autre notation : **pluslin**(a,b,c) s'écrit aussi $a = b + c$.

Exemples :

```
>> pluslin(A, B, C).
C ~ real,
B ~ real,
A ~ real.
>> A ~ pluslin(3, 1).
A = 4.
>> A ~ A + 1 .
false.
>> pluslin(A, B, 0).
A = B,
B ~ real.
```

Voir également : divlin/3, minuslin/3, plus/3, timeslin/3, uminuslin/2, upluslin/2.

power/3

.^ Puissance nième

Définition : **power/3** définit la relation $\{(a,b,c) \in \mathbf{R}^2 \times \mathbf{Z} \mid c \geq 0 \text{ et } a = b^c\}$.

Description : L'implantation de **power/3** n'est pas totalement conforme à la définition ci-dessus. En fait, **power/3** est implantée comme une suite infinie de relations binaires **power**$_1$**/2**, **power**$_2$**/2**, ..., où l'interprétation de chacun des **power**$_i$**/2** est donnée par :

$\{(a,b) \in \mathbf{R}^2 \mid a = b^i\}$.

Dans la pratique, cela se ressent dans l'implantation de **power/3** par le fait qu'aucune réduction ne peut être effectuée sur le domaine du troisième argument. L'exploitation des contraintes **power**(a,b,c) est donc retardée jusqu'à ce que la valeur de ce troisième argument c soit connue.

Autre notation : **power/3**(a,b,c) s'écrit aussi $a = b$ **.^.** c.

Exemples :

```
>> power(A, B, C).
C ~ real,
B ~ real,
A ~ real.
>> A ~ power(2,3).
A = 8.
>> A ~ 1.01 .^. 36.
A ~ oo('>1.4307682','>1.4307744').
>> A ~ power(cc(1,3),2).
A ~ cc(1,9).
>> set_prolog_flag(interval_mode,union).
true.
>> A ~ power(cc(-2,-1) u cc(1,2),2).
A ~ cc(1,4).
>> A ~ power(cc(-2,-1) u cc(1,2),3).
A ~ cc(-8,-1)u cc(1,8).
```

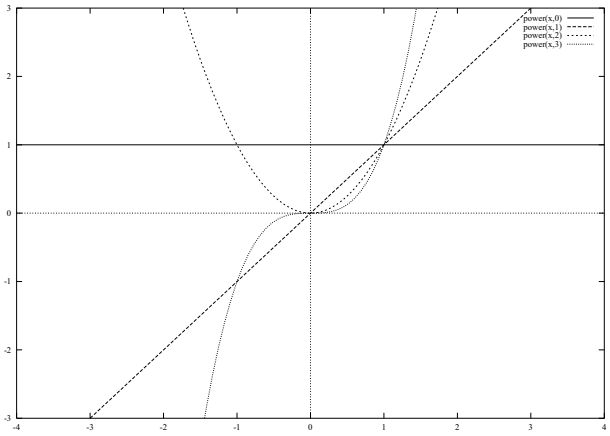
Les contraintes **power**(a,b,c) sont donc retardées jusqu'à ce que la valeur du dernier argument c soit connue :

```
>> A ~ power(cc(1,3),cc(2,3)).
A ~ real.
```

Par convention, $\forall x, x^0 = 1$, et donc en particulier, $0^0 = 1$.

```
>> power(A, B, 0).
A = 1,
B ~ real.
>> A ~ 0 .^. 0.
A = 1.
```

Graphe :



Voir également : exp/2, ln/2, log/2, root/3, square/2, sqrt/2.

prime/1 — Etre un entier premier

Définition : **prime/1** définit la relation $\{a \in \mathbf{R} \mid a \text{ est un entier premier}\}$.

Description : La relation **prime/1** n'est pas implantée dans la version actuelle de Prolog IV. Toutefois, étant donné qu'elle fait partie de la définition du langage, le symbole a été réservé.

Exemples :

```
>> prime(A).
Relation not yet implemented: prime
false.
```

Voir également : int/1, nint/1, nprime/1.

real/1 — Etre un nombre réel

Définition : **real/1** définit la relation $\{a \in \mathbf{A} \mid a \text{ est un nombre réel}\}$.

Description : **real/1** est une relation générale sur les arbres, qui est traitée par le solveur général. Une contrainte **real**(a) est vérifiée quand son argument représente un nombre réel.

real/1 est une relation, et pas une primitive de vérification comme **number/1**. Si une variable `X` est contrainte par `real(X)`, elle ne peut plus s'unifier avec un arbre qui n'est pas un nombre réel.

Attention, comme toutes les relations numériques de Prolog IV, la relation **real/1** n'est pas compatible avec les nombres flottants de la norme ISO.

Exemples :

```
>> real(A).
A ~ real.
>> real(a).
false.
>> real(1).
true.
>> real(f(X)).
false.
>> real(1 + X + Y).
Y ~ real,
X ~ real.
```

Voir également : breal/2, bnreal/2, nreal/1.

root/3 — Racine nième

Définition : **root/3** définit la relation $\{(a, b, c) \in \mathbf{R}^2 \times \mathbf{Z} \mid c > 0 \text{ et } a = \sqrt[c]{b}\}$.

Description : L'implantation de **root/3** n'est pas totalement conforme à la définition ci-dessus. En fait, **root/3** est implantée comme une suite infinie de relations binaires **root$_1$/2**, **root$_2$/2**, ..., où l'interprétation de chacun des **root$_i$/2** est donnée par :

$$\{(a, b) \in \mathbf{R}^2 \mid a = \sqrt[i]{b}\}.$$

Dans la pratique, cela se ressent dans l'implantation de **root/3** par le fait qu'aucune réduction ne peut être effectuée sur le domaine du troisième argument. L'exploitation des contraintes **root**(a, b, c) est donc retardée jusqu'à ce que la valeur de ce troisième argument soit connue.

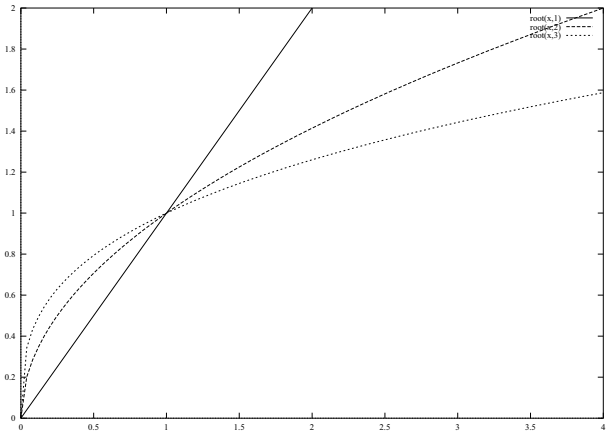
Exemples :

```
>> root(A, B, C).
C ~ real,
B ~ real,
A ~ real.
>> A ~ root(2,3).
A ~ cc('<1.259921','>1.259921').
>> A ~ root(0.12, 36).
A ~ oo('<0.9428046','>0.9428046').
>> A ~ root(cc(1,3),2).
A ~ cc(1,'<1.732051').
```

Par convention, $\forall x, x^0 = 1$. En transposant aux racines, nous obtenons donc :

```
>> root(A, B, 0).
B = 1,
A ~ real.
>> A ~ root(1,0).
A ~ real.
```

Graphe :



Voir également : exp/2, ln/2, log/2, power/3, square/2, sqrt/2.

sin/2 — Sinus

Définition : **sin/2** définit la relation $\{(a, b) \in \mathbf{R}^2 \mid a = \sin b\}$.

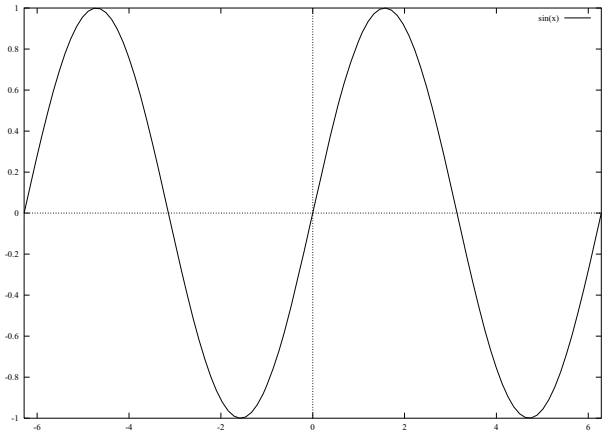
Description : **sin/2** est une relation sur les réels, qui est traitée par le solveur sur les intervalles. Elle établit une relation entre une valeur numérique et son sinus.

Exemples :

```
>> sin(A, B).
B ~ real,
A ~ cc(-1,1).
>> sin(A, 0).
A = 0.
>> A = 2.*.square(sin(pi./.4)).
A ~ oo(‘<0.9999984‘,‘>1.000001‘).
>> A = 4.*.square(sin(pi./.3)).
A ~ oo(‘<2.999997‘,‘>3.0000016‘).
>> sin(A, pi./.2).
A ~ oc(‘>0.9999999‘,1).
>> sin(A, A).
A ~ oo(-‘>0.017607333‘,‘<0.01740003‘).
```

Note : Voir **cos/2** pour des remarques importantes concernant l'utilisations de relations associées à des fonctions périodiques en mode unions d'intervalles.

Graphe :



Voir également : arccos/2, arcsin/2, arctan/2, cos/2, cot/2, pi/2, tan/2.

sinh/2 — Sinus hyperbolique

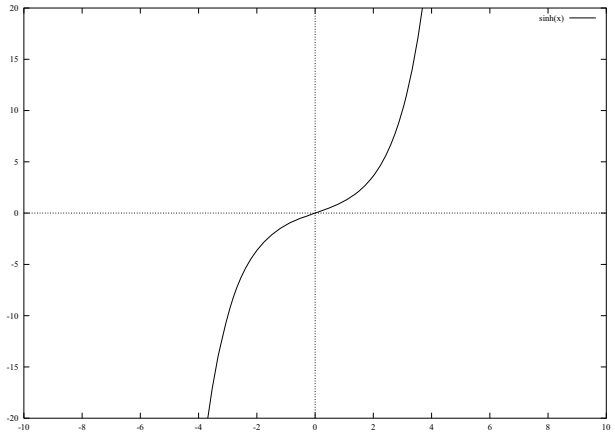
Définition : **sinh/2** définit la relation $\{(a, b) \in \mathbf{R}^2 \mid a = \sinh b\}$.

Description : **sinh/2** est une relation sur les réels, qui est traitée par le solveur sur les intervalles. Elle établit une relation entre une valeur numérique et son sinus hyperbolique.

Exemples :

```
>> sinh(A, B).
B ~ real,
A ~ real.
>> sinh(0, B).
B = 0.
>> sinh(1, B).
B ~ cc(‘<0.8813736‘,‘>0.8813736‘).
>> sinh(A, 0).
A = 0.
>> sinh(A, 1).
A ~ cc(‘>1.1752011‘,‘>1.1752012‘).
>> sinh(A, 10).
A ~ cc(‘>11013.232‘,‘>11013.233‘).
>> sinh(A, 100).
A ~ ge(‘<4e38‘).
>> sinh(A, A).
A ~ real.
```

Graphe :



Voir également : cosh/2, coth/2, tanh/2.

size/2 — Taille d'une liste

Définition : **size/2** définit la relation $\{(a,b) \in \mathbf{Z} \times \mathbf{L} \mid a = |b|\}$.

Description : **size/2** est une relation sur les listes, qui est traitée par le solveur général. Elle établit une relation entre une liste et une valeur numérique représentant sa longueur.

Exemples :

```
>> size(A, B).
B ~ list,
A ~ ge(0).
>> size(0, B).
B = [].
>> size(B) ~ 4 .
B ~ [tree,tree,tree,tree].
```

La contrainte **size**(a,b) ne fait des déductions sur la longueur a que quand celle-ci devient connue. Nous avons donc :

```
>> size(A, [1, 2, 3, 4 | B]).
B ~ list,
A ~ ge(0).
```

La variable `A` pourrait ici être contrainte à appartenir à `ge(4)`.

Voir également : conc/3, index/3.

sqrt/2 — Racine carré

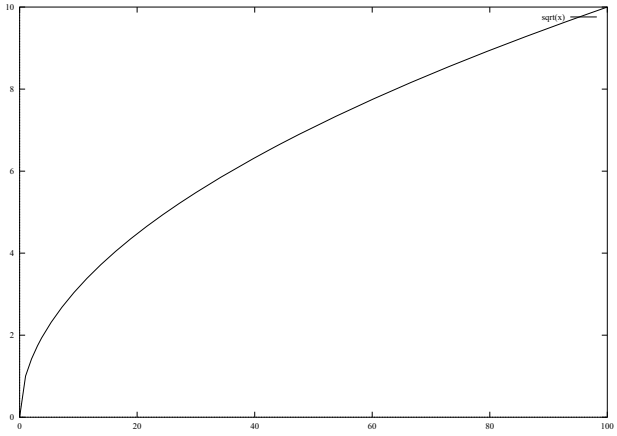
Définition : **sqrt/2** définit la relation $\{(a,b) \in \mathbf{R}^2 \mid b \geq 0 \text{ et } a = \sqrt{b}\}$.

Description : **sqrt/2** est une relation sur les réels, qui est traitée par le solveur sur les intervalles. Elle établit une relation entre une valeur numérique x et sa racine carrée $\sqrt{x}$.

Exemples :

```
>> sqrt(A, B).
B ~ ge(0),
A ~ ge(0).
>> A ~ sqrt(2).
A ~ cc(‘>1.4142135‘,‘>1.4142136‘).
>> A ~ cc(-2,2), B ~ sqrt(A).
B ~ cc(0,‘>1.4142136‘),
A ~ cc(0,2).
>> cc(2,3) ~ sqrt(B).
B ~ cc(4,9).
```

Graphe :



Voir également : exp/2, ln/2, log/2, power/3, root/3, square/2.

square/2 — Carré

Définition : **square/2** définit la relation $\{(a,b) \in \mathbf{R}^2 \mid a = b^2\}$.

Description : **square/2** est une relation sur les réels, qui est traitée par le solveur sur les intervalles. Elle établit une relation entre une valeur numérique x et son carré x^2.

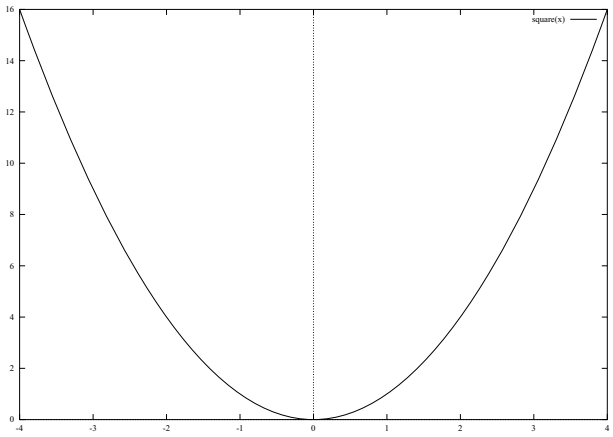
Exemples :

```
>> square(A,B).
B ~ real,
A ~ ge(0).
>> A = square(2).
A = 4.
>> A ~ square(cc(2,3)).
A ~ cc(4,9).
```

On note ici que la contrainte `sqrt(A,B)` n'est pas strictement équivalente à la contrainte `square(B,A)`, car la relation square est également définie pour les nombres négatifs.

```
>> cc(2,3) ~ square(B).
B ~ cc(-'<1.732051','<1.732051').
>> set_prolog_flag(interval_mode,union).
true.
>> cc(2,3) ~ square(B).
B ~ cc(-'<1.732051',-'>1.4142135')u cc('>1.4142135','<1.732051').
>> A ~ square(cc(2,3)) n int .
A ~ 4 u 5 u 6 u 7 u 8 u 9.
>> A ~ square(cc(2,3) n int).
A ~ 4 u 9.
>> set_prolog_flag(interval_mode,simple).
true.
```

Graphe :



Voir également : exp/2, ln/2, log/2, power/3, root/3, sqrt/2.

tan/2 — Tangente

Définition : **tan/2** définit la relation $\{(a, b) \in \mathbf{R}^2 \mid b \bmod \pi \neq \frac{\pi}{2} \text{ et } a = \tan b\}$.

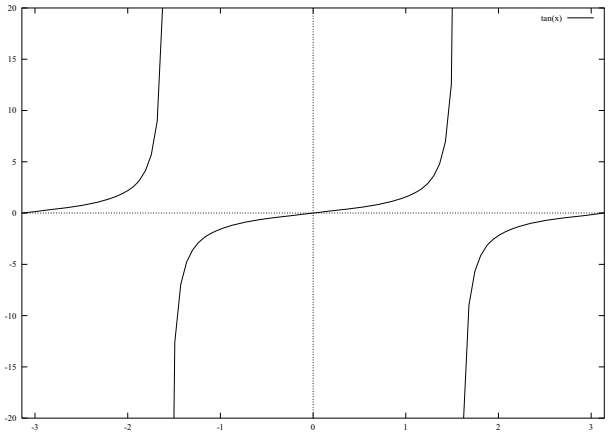
Description : **tan/2** est une relation sur les réels, qui est traitée par le solveur sur les intervalles. Elle établit une relation entre une valeur numérique et sa tangente.

Exemples :

```
>> tan(A, B).
B ~ real,
A ~ real.
>> tan(0, B).
B ~ real.
>> tan(1, B).
B ~ real.
>> tan(A, 0).
A = 0.
>> tan(A, pi./.4).
A ~ oo(`<0.9999999`,`>1.0000001`).
>> tan(A, pi./.2).
A ~ real.
>> tan(A, A).
A ~ real.
```

Note : Voir **cos/2** pour des remarques importantes concernant l'utilisations de relations associées à des fonctions périodiques en mode unions d'intervalles.

Graphe :



Voir également : arccos/2, arcsin/2, arctan/2, cos/2, cot/2, pi/2, sin/2.

tanh/2 — Tangente hyperbolique

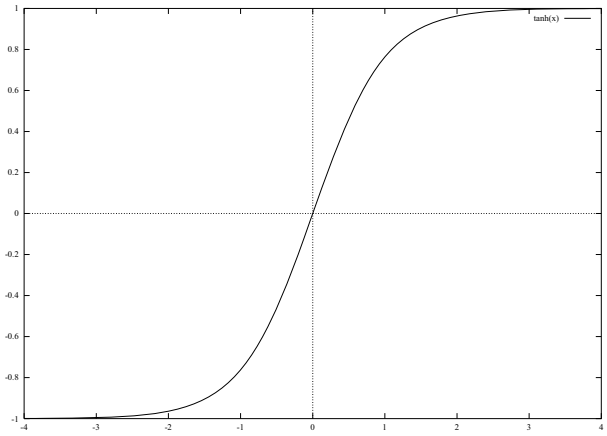
Définition : **tanh/2** définit la relation $\{(a, b) \in \mathbf{R}^2 \mid a = \tanh b\}$.

Description : **tanh/2** est une relation sur les réels, qui est traitée par le solveur sur les intervalles. Elle établit une relation entre une valeur numérique et sa tangente hyperbolique.

Exemples :

```
>> tanh(A, B).
B ~ real,
A ~ oo(-1,1).
>> tanh(0, B).
B = 0.
>> tanh(A, 0).
A = 0.
>> tanh(A, 1).
A ~ cc('>0.7615941','<0.7615942').
>> tanh(A, 2).
A ~ cc('>0.9640275','<0.9640276').
>> tanh(A, 10).
A ~ cc('>0.9999999',1).
```

Graphe :



Voir également : cosh/2, coth/2, sinh/2.

times/3
.*. — Multiplication

Définition : **times/3** définit la relation $\{(a, b, c) \in \mathbf{R}^3 \mid a = b \times c\}$.

Description : **times/3** est une relation sur les réels, qui est traitée par le solveur sur les intervalles. Elle établit une relation entre deux nombres et leur produit.

Autre notation : **times**(a, b, c) s'écrit aussi $a = b$ **.*.** c.

Exemples :

```
>> times(A, B, C).
C ~ real,
B ~ real,
A ~ real.
>> A ~ times(2, 3).
A = 6.
>> A ~ cc(1,2) .*. cc(1,3).
A ~ cc(1,6).
>> A ~ B .*. 0 .
A = 0,
B ~ real.
>> A ~ B .*. 1 .
A = B,
B ~ real.
>> times(cc(1,2), cc(-1,1), A) .
A ~ real.
>> set_prolog_flag(interval_mode,union).
true.
>> times(cc(1,2), cc(-1,1), A) .
A ~ le(-1)u ge(1).
>> set_prolog_flag(interval_mode,simple).
true.
```

Note : Quand deux des arguments de **times/3** sont réduits à des singletons, le troisième est calculé de manière exacte. Nous avons donc :

```
>> 1 ~ 3 .*. A .
A ~ 1/3.
```

Il existe deux exceptions notables à cette règle :

```
>> times(0,0,A).
A ~ real.
>> times(0,A,0).
A ~ real.
```

Voir également : div/3, minus/3, plus/3, timeslin/3, uminus/2, uplus/2.

timeslin/3

* Multiplication

Définition : **timeslin/3** définit la relation $\{(a, b, c) \in \mathbf{R}^3 \mid a = b \times c\}$.

Description : **timeslin/2** est une relation sur les réels, qui est traitée par le solveur linéaire. Elle établit une relation entre deux nombres et leur produit.

Autre notation : **timeslin**(a, b, c) s'écrit aussi $a = b * c$.

Exemples :

```
>> timeslin(A, B, C).
C ~ real,
B ~ real,
A ~ real.
>> A ~ timeslin(2, 3).
A = 6.
>> A ~ B * 0 .
A = 0,
B ~ real.
>> A ~ B * 1 .
A = B,
B ~ real.
```

Note : **timeslin/3** est une relation du solveur linéaire. Or, il est possible de l'utiliser pour poser des contraintes non linéaires, comme par exemple dans la requête suivante :

```
>> X * X = -1 .
X ~ real.
```

Comme on le voit, la contrainte n'est pas prise en compte. En fait, elle est retardée jusqu'à ce qu'elle devienne linéaire, c'est-à-dire jusqu'à ce qu'au moins un des arguments de la multiplication soit connu. Dans l'exemple suivant, on obtient un échec dans la deuxième requête dès que la valeur de `T` est connue :

```
>> X = Y*T, X = Y*T + 1 .
T ~ real,
Y ~ real,
X ~ real.
>> X = Y*T, X = Y*T + 1, T = 2 .
false.
```

Les mêmes remarques s'appliquent à la relation **divlin/2**.

Voir également : divlin/3, minuslin/3, pluslin/3, times/3, uminuslin/2, upluslin/2.

tree/1 — Etre un arbre

Définition : **tree/1** définit la relation $\{a \in \mathbf{A} \mid a$ n'est assujetti à aucune condition$\}$.

Description : **tree/1** est une relation générale sur les arbres, qui est vérifiée pour tout arbre. Cette contrainte n'échoue jamais.

tree/1 est également le pseudo-terme le plus général, qui est très souvent utilisé dans les réponses de Prolog IV pour indiquer qu'aucune restriction n'a été effectuée sur le domaine d'une variable.

Exemples :

```
>> tree(A).
A ~ tree.
>> tree(1).
true.
>> tree(f(X)).
X ~ tree.
```

Voir également : ntree/1.

u/3 — Union

Définition : **u/3** définit la relation $\{(a,b,c) \in \mathbf{A}^3 \mid a = b$ ou $a = c\}$.

Description : **u/3** est une relation générale sur les arbres. Une contrainte $\mathbf{u}(a,b,c)$ est vérifiée si son premier argument a est égal soit à son deuxième b, soit à son troisième c.

Le nom de **u/3** a été choisi en référence au symbole de l'union des ensembles. C'est en effet l'usage le plus courant de cette relation, au niveau des pseudo-termes. Se reporter à la note après les exemples pour plus d'explications à ce sujet.

Autre notation : $\mathbf{u}(a,b,c)$ s'écrit aussi $a = b$ **u** c.

Exemples :

```
>> u(A, B, C).
C ~ tree,
B ~ tree,
A ~ tree.
>> A ~ u(1,2).
A ~ cc(1,2).
>> u(A, cc(1,2), cc(3,4)).
A ~ cc(1,4).
>> X ~ identifier u real .
X ~ leaf.
>> set_prolog_flag(interval_mode,union).
true.
>> A ~ u(1,2).
A ~ 1 u 2.
>> u(A, cc(1,2), cc(3,4)).
A ~ cc(1,2)u cc(3,4).
>> set_prolog_flag(interval_mode,simple).
true.
```

Note : Attention, la relation «u» n'est pas une relation ensembliste, et n'effectue donc

pas toutes les propagations qu'on attendrait d'une union. C'est simplement un moyen élégant de noter des disjonctions :

```
>> A ~ cc(1,2), A ~ B u C .
A ~ cc(1,2),
C ~ tree,
B ~ tree.
```

La requête ci-dessus est équivalente au système d'équations suivant :

$$\begin{cases} 1 \leq a \leq 2 \\ (a = b) \vee (a = c) \end{cases}$$

On voit bien que ce système ne contraint pas les valeurs de b et c, à cause de la disjonction. A l'opposé, une telle contrainte peut faire des déductions qui n'ont rien à voir avec l'union des ensembles :

```
>> A ~ cc(1,5), B ~ cc(-10,10), A ~ B u cc(7,10).
A = B,
B ~ cc(1,5).
```

Voir également : n/3.

uminus/2

.—. Moins unaire

Définition : **uminus/2** définit la relation $\{(a,b) \in \mathbf{R}^2 \mid a = -b\}$.

Description : **uminus/2** est une relation sur les réels, qui est traitée par le solveur sur les intervalles. Elle établit une relation entre un nombre et son opposé.

Autre notation : **uminus**(a,b) s'écrit aussi $a =$ **.—.** b.

Exemples :

```
>> uminus(A, B).
B ~ real,
A ~ real.
>> A ~ uminus(-1) .
A = 1.
>> A ~ .-. 2.
A = -2.
>> A ~ .-. cc(1,2).
A ~ cc(-2,-1).
>> A ~ le(-3) n ( .-. cc(1,5)).
A ~ cc(-5,-3).
>> A ~ .-. A .
A ~ real.
```

Note : Quand un des arguments de **uminus/2** est réduit à un singleton, le deuxième est calculé de manière exacte. Nous avons donc :

```
>> A ~ .-. 1/3 .
A ~ -1/3.
```

Voir également : div/3, minus/3, plus/3, times/3, uminuslin/2, uplus/2.

uminuslin/2

− Moins unaire

Définition : **uminuslin/2** définit la relation $\{(a,b) \in \mathbf{R}^2 \mid a =$-$b\}$.

Description : **uminuslin/2** est une relation sur les réels, qui est traitée par le solveur linéaire. Elle établit une relation entre un nombre et son opposé.

Autre notation : **uminuslin**(a,b) s'écrit aussi $a = -b$.

Exemples :

```
>> uminuslin(A, B).
B ~ real,
A ~ real.
>> A ~ uminuslin(2).
A = -2.
>> A ~ -A .
A = 0.
```

Voir également : divlin/3, minuslin/3, pluslin/3, timeslin/3, uminus/2, upluslin/2.

uplus/2

.+. Plus unaire

Définition : **uplus/2** définit la relation $\{(a,b) \in \mathbf{R}^2 \mid a = +b\}$.

Description : **uplus/2** est une relation sur les réels, qui est traitée par le solveur sur les intervalles. Elle établit une relation entre deux nombres réels égaux. C'est donc l'identité sur les nombres réels.

Autre notation : **uplus**(a,b,c) s'écrit aussi $a = b$ **.+.** c.

Exemples :

```
>> uplus(A, B).
A = B,
B ~ real.
>> A ~ uplus(1).
A = 1.
>> A ~ cc(1,4), B ~ cc(3,5), A ~ .+. B .
A = B,
B ~ cc(3,4).
```

Note : Quand un des arguments de **uplus/2** est réduit à un singleton, le deuxième est calculé de manière exacte. Nous avons donc :

```
>> A ~ .+. 1/3 .
A ~ 1/3.
```

Voir également : div/3, minus/3, plus/3, times/3, uminus/2, upluslin/2.

upluslin/2

+ Plus unaire

Définition : **upluslin/2** définit la relation $\{(a,b) \in \mathbf{R}^2 \mid a = +b\}$.

Description : **upluslin/2** est une relation sur les réels, qui est traitée par le solveur linéaire. Elle établit une relation entre deux nombres réels égaux. C'est donc l'identité sur les nombres réels.

Autre notation : **upluslin**(a,b,c) s'écrit aussi $a = b + c$.

Exemples :

```
>> upluslin(A, B).
A = B,
B ~ real.
>> A ~ upluslin(1).
A = 1.
>> +A ~ -A .
A = 0.
```

Voir également : divlin/3, minuslin/3, pluslin/3, timeslin/3, uminuslin/2, uplus/2.

xor/2 Ou exclusif

Définition : **xor/2** définit la relation $\{(a,b) \in \mathbf{B}^2 \mid a \neq b\}$.

Description : **xor/2** est une relation sur les booléens, qui est traitée par le solveur sur les intervalles. Une contrainte **xor**(a,b) réussit si ses deux arguments sont des booléens distincts (ce qui revient à la définition du ou exclusif).

Cette contrainte est normalement utilisée pour construire une combinaison d'autres contraintes exprimées par des pseudo-opérations produisant des booléens.

Autre notation : **xor**(a,b) s'écrit aussi a **xor** b.

Exemples :

```
>> xor(A, B).
B ~ cc(0,1),
A ~ cc(0,1).
>> xor(0, B).
B = 1.
>> xor(1, B).
B = 0.
>> xor(A, 0).
A = 1.
>> xor(A, 1).
A = 0.
>> xor(A, A).
A ~ cc(0,1).
>> xor(A, A), boolsplit([A]).
false.
```

Voir également : and/2, bxor/3, equiv/2, impl/2, not/1, or/2.

4

Prédicats prédéfinis Prolog IV

DANS CE CHAPITRE sont décrits les prédicats prédéfinis et les prédicats d'énumération de Prolog IV. Après des préliminaires explicitant des généralités sur ces primitives et les concepts qui les entoure, une liste détaillée de ces primitives est donnée dans l'ordre alphabétique afin d'en faciliter les recherches.

4.1 Préliminaires

La présente section donne un aperçu des diverses catégories de prédicats prédéfinis en Prolog IV. On touvera quelques explications sur les primitives traitant les règles, les affectations statiques (variables globales et tableaux) ainsi que les primitives de méta-programmation sur les intervalles (dont les énumérations).

4.1.1 L'entrée des règles

Voici tout d'abord quelques notions élémentaires sur les règles prolog.

Paquets de règles

On appelle nom et arité d'une règle le nom et l'arité de la tête de cette règle.

On appelle paquet de règles un ensemble de règles de mêmes nom et arité. Un paquet est donc parfaitement repéré par ce couple (nom, $arité$). L'usage veut que les règles de mêmes nom et arité soient regroupées en un seul paquet, et non pas éparpillées dans un ou plusieurs fichiers.

La description d'un paquet sera souvent noté $nom/arité$.

Compilation, consultation

Il existe en Prolog IV trois façons d'entrer des règles :

- en compilant un fichier,
- en consultant un fichier (ou la console, qui n'est autre chose qu'un fichier spécial toujours ouvert),

– en effectuant des assertions de règles[1] (à partir de termes) pendant l'exécution d'un programme Prolog IV.

Il existe deux formes de codage de règle en Prolog IV, dues à la présence d'impératifs techniques contradictoires :

– la forme compilée (au moyen des primitives de la famille `compile`). Elle a pour avantage la vitesse d'exécution de ces règles, une fois appelées. Son inconvénient est l'impossibilité de traduire ces règles en termes, et de modifier dynamiquement les paquets de règles ainsi compilés (autrement qu'en les détruisant).

– la forme interprétée (ou assertée) (au moyen des primitives des familles `consult` ou `assert`). Son avantage est la possibilité de traduire ces règles en termes, et de modifier dynamiquement les paquets de règles (en ajoutant ou supprimant une règle donnée du paquet). Son inconvénient est une moins grande rapidité d'exécution, ainsi qu'une plus grande consommation mémoire. On ne peut pas suivre avec le débogueur le déroulement de l'exécution des programmes entrés sous cette forme.

Malgré ce qui a été montré tout au long du tutoriel, c'est le mode d'entrée de règles par compilation qui est fortement préconisé. Il est en effet beaucoup plus rare d'avoir besoin d'une gestion dynamique de règles.

Les primitives de lecture de règles

consult lit une suite de règles dans l'entrée courante. Si un paquet de règles ayant mêmes nom et arité fait déjà partie de la base de règles, une erreur indiquant une tentative de redéfinition survient. Si on souhaite vraiment redéfinir ce paquet, il faut utiliser la primitive `reconsult`.

consult(F**)** se comporte comme `consult` sans argument, mais les paquets de règles sont lus dans un fichier de nom F.

reconsult lit une suite de paquets de règles dans l'entrée courante. Ceux qui existaient déjà avec ces mêmes nom et arité sont détruits si ce sont des règles utilisateurs[2] seulement.

reconsult(F**)** se comporte comme `reconsult` sans argument, mais les paquets de règles sont lus dans un fichier de nom F.

compile(F**), compile(**$F, Options$**)** se comporte comme `consult`, mais utilise la compilation des règles comme forme de codage interne.

recompile(F**), compile(**$F, Options$**)** se comporte comme `reconsult`, mais utilise la compilation des règles comme forme de codage interne.

Quand une erreur survient pendant la (re)compilation, la base de règles est laissée dans l'état où elle se trouvait avant la compilation.

1. Par abus de langage, on dira «asserter des règles» et «règles assertées».

2. Ce sont toutes les règles qui ne font pas partie du système Prolog IV.

Dans tous les cas :

- La fin de fichier ou le terme «`end_of_file.`» arrêtent le lecteur de règles.
- Les règles sont codées en mémoire.
- Tenter de redéfinir des règles prédéfinies mène à une erreur.

Note : On ne peut pas compiler de règles dans la console, il faut passer par un fichier[3].

4.1.2 Affectation, variables globales et tableaux

On a en Prolog IV la possibilité de conserver des termes dans ce qu'on appelle variables globales. Ces variables globales sont nommées à l'aide d'identificateurs, ou à l'aide d'une notation fonctionnelle qui rappelle l'indiçage d'un tableau à une dimension. Les termes conservés sont des copies. Cette remarque prend toute son importance lorsque le terme contient une ou plusieurs variables, puisque celles-ci sont déconnectées du contexte où elles ont été créées. Retrouver un terme conservé crée également une copie. Les sous-domaines ne sont en général pas conservés.

```
>> record(toto, f(g(X), [3,5,7])).
X ~ tree.
>> recorded(toto,X).
X ~ f(g(tree),[3,5,7]).
```

Rappel : Le pseudo-terme $tree$ représente une variable muette.

record(I**,**T**)** associe une copie du terme T à l'identificateur I. Le terme T peut contenir des variables, peut être un arbre infini et peut même contenir des variables soumises à des contraintes linéaires.

recorded(I**,**T**)** crée une copie du terme associé à l'identificateur I et l'unifie avec T.

Il existe aussi la notion de tableau. On doit déclarer son nom et sa taille avant de pouvoir l'utiliser. On peut ensuite se servir de chacun de ses éléments (d'indice entre 1 et *taille*) au travers des primitives précédentes `record/2` et `recorded/2`. Utiliser un tableau non déclaré, ou un indice hors des bornes du tableau mène à une erreur.

def_array(I**,**N**)** définit le tableau de nom I (un identificateur) de taille N éléments.

```
>> def_array(tab,10).
true.
>> record(tab(1), f(g,100/3)).
true.
>> recorded(tab(1),X).
X = f(g,100/3).
```

3. On peut cependant, lorsqu'on est sous unix et dans un terminal `tty` (comme une fenêtre `xterm` ou `cmdtool`) utiliser le pseudo-fichier `/dev/tty`, comme dans `compile('/dev/tty')`, qui attendra des buts dans le terminal, et ce jusqu'à la fin de fichier (généralement obtenue en tapant `CTRL-D`)

redef_array(I**,**N**)** redimensionne le tableau de nom I à la taille N.

undef_array(I**)** détruit le tableau de nom I.

4.1.3 Retard

La fameuse primitive `freeze/2`.

freeze/2 retarde l'exécution d'un littéral tant qu'une variable est libre.

4.1.4 Divers

Mesure de temps

Ces primitives permettent la mesure de temps CPU lors de l'exécution d'un programme Prolog IV.

reset_cpu_time réinitialise le chronomètre.

cpu_time(T**)** unifie T avec le temps cpu en millisecondes écoulé depuis le dernier appel à `reset_cpu_time`.

Commandes systèmes

chdir(D**)** positionne le répertoire de travail de Prolog IV à la chaîne représentée par l'atome D.

command(C**)** effectue la commande système contenue dans la chaîne représentée par l'atome C.

4.1.5 Méta-prédicats et intervalles

L'utilisation des techniques de réduction de domaines pour résoudre des problèmes numériques est en général constituée deux phases :

1. La phase de pose des contraintes, qui utilise principalement les relations de Prolog IV.
2. La phase d'énumération qui permet de cerner les solutions du système de contraintes, ou de démontrer qu'il n'y en a pas.

Cette phase d'énumération est absolument nécessaire dans la plupart des problèmes car les informations qu'apporte le solveur approché seul sont en général bien insuffisantes pour être utiles.

Supposons que l'on cherche à résoudre le système de contraintes $\{X = -X\}$. Voici ce que l'on obtient :

```
>> X = .-. X.
X ~ real.
```

La réponse de Prolog IV n'est manifestement pas convainquante. On peut par contre noter que Prolog IV est beaucoup plus prolixe si on lui demande :

```
>> X = .-. X, le(X,0).
X = 0.

>> X = .-. X, gt(X,0).
false.
```

On peut alors noter que $\{X \leq 0\}$ et $\{X > 0\}$ forment une partition des nombres réels, et en déduire que les deux requêtes précédentes permettent de conclure quand aux solutions du système $\{X = -X\}$.

Ce petit exemple laisse apparaître clairement le principe de base d'une énumération : diviser un problème trop difficile à résoudre en sous-problèmes plus petits, donc, on l'espère, plus faciles à résoudre. Dans le cadre des intervalles, on scindera volontiers les problèmes en découpant les domaines associés aux variables, comme sur cet exemple.

Il existe bien sûr mille et une manières de concevoir et de programmer une énumération. Prolog IV propose, parmi ses prédicats, un certain nombre d'outils permettant :

1. De programmer ses propres énumérations : ce sont les méta-prédicats de bas niveau. Ils permettent de manipuler les domaines des variables, les rationnels IEEE 754, ou bien encore d'obtenir des informations statistiques sur la résolution des problèmes numériques par les intervalles.
2. De disposer d'énumérations préprogrammées : ce sont les prédicats de haut niveau. Ils permettent d'énumérer sans avoir à programmer sa propre énumération des systèmes de contraintes de nature booléens, entiers, ou réels.

Le principe des prédicats d'énumération prédéfinis que sont `boolsplit`, `intsplit`, et `realsplit` est le même. Ces énumérations reposent sur trois paramètres fondamentaux : la condition d'arrêt, le choix de la variable dont le domaine va être scindé en deux, et la manière dont ce découpage va être fait.

Voici le programme Prolog IV qui implante et décrit ces procédés d'énumération :

```
mon_enum(L):-
   condition_d_arret(L),
   !.
mon_enum(L):-
   choix_variable(L,X),
   milieu(X,M),
   decouper(X,M),
   mon_enum(L).

decouper(X,M):-
   le(X,M).
decouper(X,M):-
   gt(X,M).
```

Avec :

– `condition_d_arret(L)`, un prédicat dont la réussite signifie que la condition d'arrêt est atteinte.

– `choix_variable(L,X)`, un prédicat qui unifie `X` avec la variable de la liste `L` sur laquelle la dichotomie va être effectuée.

– `milieu(X,M)` qui calcule le milieu du domaine de `X`.

L'implantation effective des méta-prédicats d'énumération de Prolog IV est un peu plus complexe que le petit programme que l'on vient de présenter, mais celui-ci reflète bien la méthodologie utilisée.

4.1.6 Méta-prédicats sur les réels

De façon générale, les primitives dont le nom se terminent en `lin` travaillent exclusivement avec le solveur linéaire, et ne tiennent donc pas compte des contraintes de type «intervalle». Réciproquement, celles qui ne se terminent pas en `lin` ignorent les contraintes linéaires.

Enumération dans un domaine

boolsplit/1 /2 énumèrent les valeurs possibles booléennes des éléments d'une liste de variables.

intsplit/1 /2 /3 énumèrent les valeurs possibles entières d'une liste de variables.

realsplit/1 /2 /3 /4 énumèrent les valeurs possibles réelles d'une liste de variables.

enum/1 /3 énumèrent les valeurs possibles entières d'une variable.

enumlin/1 /3 énumèrent les valeurs possibles entières d'une variable (en ne tenant compte que des contraintes linéaires).

Bornes d'un domaine

bounds/3, glb/2, lub/2 récupèrent les bornes d'un domaine.

ebounds/3, eglb/2, elub/2 récupèrent les bornes d'un domaine, avec des conventions pour indiquer le type de celles-ci.

boundslin/3, glblin/2, lublin/2 calculent les bornes d'une variable (en ne tenant compte que des contraintes linéaires).

Taille d'un domaine basé sur les intervalles

float_size/2 rend la mesure d'un domaine en nombre d'intervalles IEEE (flottants) atomiques.

int_size/2 rend la mesure d'un domaine en nombre d'entiers relatifs.

real_size/2 rend la mesure d'un domaine dans **R**.

Rationnels IEEE *(flottants)*

max_float/1 rend le plus grand rationnel IEEE (le plus grand flottant simple).

min_positive_float/1 rend le plus petit rationnel IEEE positif (le plus petit flottant simple positif).

float_rank/2 associe un nombre flottant et son rang.

Optimisation sur le linéaire

minimizelin/1, maximizelin/1 effectuent une minimisation ou une maximisation d'une variable soumise à des contraintes linéaires.

Statistiques sur les intervalles

get_istats/2 /3 rendent des statistiques sur l'usage des intervalles.

reset_istats/0 initialise des compteurs statistiques sur l'usage des intervalles.

4.2 Liste alphabétique

boolsplit/1
boolsplit/2 — Enumération d'une liste de booléens

Description : **boolsplit**(a, b) enumère les booléens du domaine des éléments de la liste a et les unifie avec les éléments de a dans l'ordre de cette liste. La première valeur essayée est b. Si b est libre, elle est unifiée avec la valeur par défaut, soit 0.

boolsplit(a) enumère les booléens du domaine des éléments de la liste a et les unifie avec les éléments de a dans l'ordre de cette liste. La première valeur essayée est 0 (faux).

Dans les deux cas, une erreur est provoquée quand a n'est pas une liste de variables compatibles avec des nombres réels.

Exemples :

```
>> or(B,C), boolsplit([B,C]).
C = 1,
B = 0;
C = 0,
B = 1;
C = 1,
B = 1.
>> or(B,C), boolsplit([B,C],1).
C = 1,
B = 1;
C = 0,
B = 1;
C = 1,
B = 0.
>> A ~ band(B,C), boolsplit([A,B,C]).
C = 0,
B = 0,
A = 0;
C = 1,
B = 0,
A = 0;
C = 0,
B = 1,
A = 0;
C = 1,
B = 1,
A = 1.
```

Voir également : intsplit/3, realsplit/4.

bounds/3 — Bornes d'un domaine

Description : **bounds**(a, b, c) sert à récupérer les bornes du domaine de a. Les bornes inférieure et supérieure du domaine de a sont unifiées respectivement avec b et c.

Un appel à **bounds**(a, b, c) échoue dans les cas suivants :

– a n'est pas d'un type compatible avec un type numérique,

– un des côtés du domaine de a n'est pas borné.

Les deux conditions reviennent à dire que a doit être une variable numérique bornée des deux côtés.

Il est important de remarquer que les valeurs retournées par **bounds/3** sont des bornes inférieure et supérieure. Ces bornes ne font donc pas toujours partie du domaine de la variable, en particulier quand une variable est contrainte à appartenir à un intervalle ouvert.

Exemples :

```
>> bounds(1, B, C).
C = 1,
B = 1.
>> bounds(cc(1,2), B, C).
C = 2,
B = 1.
>> bounds(oo(1,2), B, C).
C = 2,
B = 1.
>> bounds(pi, B, C).
C = 13176795/4194304,
B = 6588397/2097152.
>> bounds(le(0), B, C).
false.
>> bounds(ge(0), B, C).
false.
```

Voir également : glb/2, lub/2.

boundslin/3 — Bornes d'un domaine linéaire

Description : **boundslin**(a, b, c) sert à récupérer les bornes du domaine de a. Les bornes inférieure et supérieure du domaine de a sont unifiées respectivement avec b et c. Seules les contraintes linéaires sont prises en compte.

Un appel à **boundslin**(a, b, c) échoue dans les cas suivants :

– a n'est pas d'un type compatible avec un type numérique,

– un des côtés du domaine de a n'est pas borné.

Les deux conditions reviennent à dire que a doit être une variable numérique bornée des deux côtés.

Il est important de remarquer que les valeurs retournées par **boundslin/3** sont des bornes inférieure et supérieure. Ces bornes ne font pas toujours partie du domaine de la variable, en particulier quand la variable a été contrainte au moyen d'inégalités strictes (ou avec `dif`).

Exemples :

```
>> boundslin(1, B, C).
C = 1,
B = 1.
>> gelin(X,1), lelin(X,2),  boundslin(X, B, C).
X ~ real,
C = 2,
B = 1.
>> gtlin(X,1), ltlin(X,2), boundslin(X, B, C).
X ~ real,
C = 2,
B = 1.
>> boundslin(lelin(0), B, C).
false.
>> boundslin(gelin(0), B, C).
false.
```

Voir également : glblin/2, lublin/2.

chdir/1 — Changement de répertoire courant

Types : chdir(+atome)

Description : **chdir**(a) effectue un changement de répertoire courant : après la commande, Prolog IV est positionné dans le répertoire dont la chaîne est contenu dans l'atome a. On fait appel au système d'exploitation pour l'exécution de cette commande.

- Elle génère une erreur si l'argument n'est pas un atome.
- Elle échoue si le système d'exploitation retourne une erreur en lançant la commande (si le répertoire n'existe pas ou n'est pas accessible).
- Elle réussit dans les autres cas.

Exemples : On suppose être sous Unix.

```
>> system(pwd).
/mon/repertoire/de/travail
true.
>> system('cd /mon/repertoire').
true.
>> system(pwd).
/mon/repertoire/de/travail
true.
>> chdir('/mon/repertoire').
true.
>> system(pwd).
/mon/repertoire
true.
>>
```

Notes : 1. Il faut quoter l'atome avec des apostrophes s'il contient des blancs ou des caractères autres que des lettres ou des chiffres.
2. Si le système d'exploitation exige l'emploi de caractères \ pour nommer les chemins (comme sous DOS ou Windows), il faut les doubler pour pouvoir les utiliser dans les atomes quotés (ils sont sinon interprétés comme des caractères d'échappement).

Voir également : system/1

compile/1
compile/2 — Compilation d'un fichier

Types : compile(+atome)
compile(+atome, @liste)

Description : **compile**(a) lit une suite de règles dans le fichier de nom a. Les règles lues sont compilées et entrées en mémoire. Si un paquet de règles de mêmes nom et arité figure déjà dans la base de règles, une erreur indiquant une tentative de redéfinition survient. Si on souhaite vraiment redéfinir ce paquet, il faut utiliser la primitive `recompile`.

Dans la version avec un second argument, on donne au compilateur une liste d'options. Celle-ci peut être constituée des options suivantes :

- `debug(off)`, `debug(on)` pour compiler le programme en conservant (ou pas) des informations pour un éventuel usage par le débogueur. Par défaut, on a `debug(off)`.
- `syntax(prolog4)`, `syntax(iso)` pour indiquer dans quel mode le programme est lu. Par défaut, les règles sont lues en tenant compte du mode syntaxique courant de Prolog IV, c.à.d. au moment où la primitive **compile** est appelée.

Le mode syntaxique de compilation (`iso` ou `prolog4`) dépend des facteurs suivants, par priorité croissante :

- Le mode courant (indiqué par le prompt courant).
- L'option passée dans la commande de compilation (`syntax(`x`)`).
- Les directives lues dans le fichier compilé comme `:- syntax(iso)` ou `:- syntax(prolog4)`. Ces directives n'ont pour portée que la commande de compilation, et ce à partir de l'endroit où elles sont trouvées.

Une directive présente dans le fichier outrepasse l'option correspondante à partir de ce point.

En cas d'erreur pendant la compilation, la base de règles reste dans l'état où elle était avant l'appel à **compile**.

Notes :

- L'option `debug(on)` évite l'emploi de la directive `debug` (qui impose une modification du fichier source par le programmeur).
- L'option `syntax` permet d'être indépendant du mode (prompt) courant.

Exemples :

```
>> compile('menu.p4').
true.
>> compile('menu.p4', [debug(on)]).
### error (procedure lightMeal/3) : illegal redefinition
    of a static procedure.
error: normal_error
>> recompile('menu.p4', [debug(on)]).
true.
```

Voir également : recompile/n, consult/n, debug/0.

consult/0
consult/1 — Consultation d'un fichier

Types : consult
consult(+atome)

Description : **consult**(a) lit une suite de règles dans le fichier de nom a. Les règles lues sont traduites puis entrées en mémoire. Si un paquet de règles de mêmes nom et arité figure déjà dans la base de règles, une erreur indiquant une tentative de redéfinition survient. Si on souhaite vraiment redéfinir ce paquet, il faut utiliser la primitive `reconsult`.

Les règles lues ont le statut *dynamique*, ce qui veut dire que les primitives de gestion de règles (comme `retract, assert, clause`) peuvent être utilisées avec elles.

Les règles lues ne sont pas compilées, c'est là la différence avec les primitives `compile/`n. Il est fait appel à la primitive `assert` pour le codage des règles. L'exécution de ces règles s'effectue moins vite, et on ne peut pas utiliser le débogueur pour la suivre.

Dans la version sans argument, le fichier est l'entrée courante.

En cas d'erreur pendant la consultation, la base de règles reste dans l'état où elle était avant l'appel à **consult**.

On ne peut redéfinir de paquet ayant été entré avec les primitives `compile`.

Exemples :

```
>> consult('menu.p4').
true.
>> consult('menu.p4').
Consulting ...
error: consult_error(error(permission_error(modify,
static_procedure,lightMeal(_857,_858,_859)),assert/2))
>> reconsult('menu.p4').
true.
```

Voir également : reconsult/n, compile/n.

cpu_time/1 — Mesure du temps CPU

Types : cpu_time(?terme)

Description : **cpu_time(T)** unifie T avec le temps cpu (en millisecondes) écoulé depuis le dernier appel à `reset_cpu_time`.

Exemples :

```
>> reset_cpu_time, travail, cpu_time(T).
T = 16500. % soit 16,5 seconde CPU
>>
```

Voir également : reset_cpu_time/0.

debug/0 — Passage en mode debug

Description : **debug** rend actif le débogueur. Celui-ci va collecter des informations et effectuer des surveillances qui n'ont habituellement pas lieu lors de l'exécution normale des programmes. Ces derniers peuvent donc s'exécuter moins vite dans ce mode.

Le débogueur ne peut montrer des informations que sur les règles qui ont été compilées avec les primitives de la famille `compile`.

Voir le chapitre « Environnement » pour de plus amples détails sur le débogueur.

Voir également : no_debug/0, compile/n.

def_array/2 — Définition d'un tableau

Types : def_array(+atome, +entier)

Description : **def_array**(a, b) définit dynamiquement un tableau de constantes PROLOG de nom a et de longueur b. Ce tableau se comportera comme une variable «globale» (ultérieurement accessible pendant l'effacement de n'importe quel but) et «statique» (résistante au backtracking). Les valeurs légales de l'indice sont inclues dans $\{1, \ldots, b\}$.

Si un tableau de même nom existe déjà :

- s'il s'agit d'un tableau de même taille, il ne se passe rien,
- si les tailles diffèrent, alors il se produit une erreur.

L'accès et l'affectation sont analogues à ceux des autres langages de programmation.

Exemples :

```
>> def_array(tab, 10).
true.
>> N ~ cc(1, 10), enum(N), record(tab(N), N.*.N).
N = 1;
N = 2;
N = 3;
N = 4;
N = 5;
N = 6;
N = 7;
N = 8;
N = 9;
N = 10.
>> recorded(tab(6), X).
X = 36.
>> undef_array(tab).
true.
```

Voir également : record/2, recorded/2, redef_array/2, undef_array/1.

ebounds/3 — Bornes d'un domaine

Description : **ebounds**(a, b, c) est utilisé pour récupérer les bornes inférieures et supérieures du domaine de a, dans l'extension des réels. Les bornes inférieures et supérieures du domaine de a sont unifiées respectivement avec b et c.

Un appel à **bounds**(a, b, c) échoue si a n'est pas d'un type compatible avec un type numérique.

Un réel étendu peut prendre les valeurs suivantes :

- un nombre réel x,
- $+\infty$, noté `pinfinity` en Prolog IV,
- $-\infty$, noté `minfinity` en Prolog IV,
- pour chaque nombre réel x, l'objet x^+, qui est strictement plus grand que x, et strictement plus petit que tous les réels strictement plus grands que x. On note x^+ `up(`x`)` en Prolog IV.
- pour chaque nombre réel x, l'objet x^-, qui est strictement plus petit que x, et strictement plus grand que tous les réels strictement plus petits que x. On note x^- `dn(`x`)` en Prolog IV.

D'un point de vue pratique, l'extension des réels est utilisée dans deux cas :

- pour pouvoir faire une analyse des bornes qui fonctionne même quand le domaine d'une variable est non-borné ;
- pour savoir si un intervalle est ouvert ou fermé.

En fait, on peut reprendre tous les types possibles d'intervalles, et voir les

réponses effectuées par Prolog IV dans ces différents cas :

Intervalle	*Borne inf.*	*Borne sup.*
$[a,b]$	a	b
$[a,b)$	a	dn(b)
$(a,b]$	up(a)	b
(a,b)	up(a)	dn(b)
$[a,+\infty)$	a	pinfinity
$(a,+\infty)$	up(a)	pinfinity
$(-\infty,b]$	minfinity	b
$(-\infty,b)$	minfinity	dn(b)
$(-\infty,+\infty)$	minfinity	pinfinity

Les quatre informations à retenir sont les suivantes :

- **ebounds**/4 renvoie `minfinity` quand une variable n'est pas bornée inférieurement,
- **ebounds**/4 renvoie `pinfinity` quand une variable n'est pas bornée supérieurement,
- **ebounds**/4 renvoie `up(`a`)` quand une variable a une borne inférieure qu'elle ne peut pas atteindre,
- **ebounds**/4 renvoie `dn(`a`)` quand une variable a une borne supérieure qu'elle ne peut pas atteindre.

Exemples :

```
>> ebounds(1,B,C).
C = 1,
B = 1.
>> ebounds(cc(1,2),B,C).
C = 2,
B = 1.
>> ebounds(oo(1,2), B, C).
C = dn(2),
B = up(1).
>> ebounds(pi, B, C).
C = dn(13176795/4194304),
B = up(6588397/2097152).
>> ebounds(ge(0), B, C).
C = pinfinity,
B = 0.
>> ebounds(ge(0), B, C).
C = pinfinity,
B = 0.
```

Voir également : eglb/2, elub/2.

eglb/2 — Borne inférieure

Description : **eglb**(a, b) sert à récupérer la borne inférieure (en anglais, *greatest lower bound*) du domaine de a dans l'extension des réels. Le résultat est stocké dans la variable b.

Un appel à **eglb**(a, b) échoue si a n'est pas d'un type compatible avec un type numérique.

Voir `ebounds/3` pour une définition accompagnée de remarques importantes concernant les réels étendus.

Exemples :

```
>> eglb(1, B).
B = 1.
>> eglb(cc(1,2), B).
B = 1.
>> eglb(oo(1,2), B).
B = up(1).
>> eglb(pi, B).
B = up(6588397/2097152).
>> eglb(ge(0), B).
B = 0.
>> eglb(le(0), B).
B = minfinity.
```

Voir également : ebounds/3, elub/2.

elub/2 — Borne supérieure

Description : **elub**(a, b) sert à récupérer la borne supérieure (en anglais, *lowest upper bound* du domaine de a dans l'extension des réels. Le résultat est stocké dans la variable b.

Un appel à **elub**(a, b) échoue si a n'est pas d'un type compatible avec un type numérique.

Voir ebounds/3 pour une définition accompagnée de remarques importantes concernant les réels étendus.

Exemples :

```
>> elub(1, B).
B = 1.
>> elub(cc(1,2), B).
B = 2.
>> elub(oo(1,2), B).
B = dn(2).
>> elub(pi, B).
B = dn(13176795/4194304).
>> elub(ge(0), B).
B = pinfinity.
>> elub(le(0), B).
B = 0.
```

Voir également : ebounds/3, eglb/2.

enum/1
enum/3 — Enumération d'entiers

Description : **enum**(a, b, c) enumère les entiers du domaine de a compris entre b et c et les unifie avec a. L'ordre dans lequel l'énumération est effectuée n'est en aucun cas garanti.

enum(a) enumère les entiers du domaine de a et les unifie avec a.

enum/3 est définie en fonction de **enum/1** de la manière suivante :

```
enum(X,L,U) :-
  cc(X, L, U),
  enum(X).
```

Pour les deux prédicats, une erreur est déclenchée quand des arguments ne sont pas compatibles avec des nombres réels.

Exemples :

```
>> X ~ cc(0,5), enum(X).
X = 0;
X = 1;
X = 2;
X = 3;
X = 4;
X = 5.
>> enum(X, .-. pi, pi).
X = -3;
X = -2;
X = -1;
X = 0;
X = 1;
X = 2;
X = 3.
>> enum(toto).
error: error('Real variable expected',enum)
```

Voir également : enumlin/3, intsplit/3.

enumlin/1
enumlin/3 — Enumération d'entiers

Description : **enumlin**(a, b, c) enumère les entiers du domaine de a compris entre b et c et les unifie avec a. L'ordre dans lequel l'énumération est effectuée n'est en aucun cas garanti. Ici, ce sont les contraintes linéaires qui sont prises en compte pour contrôler l'énumération.

enumlin(a) enumère les entiers du domaine de a et les unifie avec a.

enumlin/3 est définie en fonction de **enumlin/1** de la manière suivante :

```
enumlin(X,L,U) :-
  gelin(X, L),
  lelin(X, U),
  enumlin(X).
```

Pour les deux prédicats, une erreur est déclenchée quand des arguments ne sont pas compatibles avec des nombres réels.

Exemples :

```
>> X ~ gelin(0) n lelin(5), enumlin(X).
X = 0;
X = 1;
X = 2;
X = 3;
X = 4;
X = 5.
>> enumlin(X, -3, 3).
X = -3;
X = -2;
X = -1;
X = 0;
X = 1;
X = 2;
X = 3.
>> enumlin(toto).
error: error('Real variable expected',enumlin)
```

Voir également : enumlin/3, intsplit/3.

float_rank/2 — Rang d'un nombre flottant

Description : **float_rank**(a, b) établit une correspondance entre un flottant a et son rang b. A l'appel de ce prédicat, au moins un des arguments doit avoir une valeur connue. Par ailleurs, si a a une valeur qui n'est pas un nombre flottant IEEE ou si b a une valeur qui n'est pas le rang d'un flottant, une erreur de type est déclenchée.

Le rang d'un flottant est une numérotation des flottants, tels que le rang de 0 est 0. Le plus petit flottant positif est donc de rang 1, et le plus grand flottant négatif[4] est de rang -1.

Cette numérotation sert par exemple à déterminer le nombre de flottants entre deux nombres.

Exemples :

```
>> float_rank(0, B).
B = 0.
>> float_rank(B, 1), min_positive_float(B).
B = 1/71362384635297994052914298472474756819137 3312.
>> float_rank(1,X).
X = 1065353216.
>> float_rank(1,X1), float_rank(2, X2), Diff = X2 - X1 .
Diff = 8388608,
X2 = 1073741824,
X1 = 1065353216.
>> float_rank(A, B).
error: error(instantiation_error,float_rank/2)
>> float_rank(1/3, B).
error: error(type_error('ieee754(single)',1/3),float_rank/2)
```

4. Donc le plus proche de 0

Voir également : max_float/1, min_positive_float/1.

float_size/2 — Mesure dans les flottants d'un domaine

Description : **float_size**(a, b) retourne dans b une mesure du domaine de a, qui représente le nombre d'intervalles IEEE atomiques (c'est-à-dire les intervalles dont les bornes sont deux rationnels IEEE identiques ou successifs) inclus dans le domaine de a.

float_size/2 sert à mesurer la taille du domaine d'une variable, et est en particulier utile lors de la phase de sélection d'une variable sur laquelle énumérer.

En mode inervalles simples, si L est le rang de la borne inférieure du domaine de a, que U le rang de la borne supérieure de a, et que N est le nombre de ces bornes que a ne peut atteindre, alors la mesure b peut être calculée par :

$$b = (U - L) \times 2 - N + 1$$

En mode unions d'intervalles, la mesure du domaine d'une variable est la somme des mesures de chaque intervalle le composant.

Exemples :

```
>> float_size(1,S).
S = 1.
>> float_size(pi,S).
S = 1.
>> float_size(real,S).
S = 8556380159.
>> float_size(oo(1,2), S).
S = 16777215.
>> float_size(cc(1,2), S).
S = 16777217.
>> cc(X,1,2), float_size(X,S),
    float_rank(1,S1), float_rank(2,S2),
    S = (S2 - S1)*2 + 1 .
S2 = 1073741824,
S1 = 1065353216,
S = 16777217,
X ~ cc(1,2).
>> set_prolog_flag(interval_mode,union).
true.
>> float_size( pi u 2 .*. pi, S).
S = 2.
>> float_size(cc(1,1000) n int, S).
S = 1000.
>> float_size(cc(1,2) u cc(4,7), S).
S = 29360130.
>> set_prolog_flag(interval_mode,simple).
true.
```

Voir également : int_size/2, real_size/2, float_rank/2.

freeze/2 — Retardement de l'exécution d'un but

Types : freeze(@terme, @terme_exécutable)

Description : **freeze**(a, b) retarde l'exécution du but b tant que la valeur de l'étiquette de a n'est pas connue. Par connue, on entend le fait que cette étiquette ne peut prendre qu'une seule valeur. Cette notion est la même que celle utilisée dans `nonvar/1`.

Note : Lorsque a devient connu, l'ensemble des buts gelés sur a vient s'insérer en tête de la résolvante. Pour des raisons d'efficacité, l'ordre dans lequel ils sont exécutés n'est pas spécifié.

Exemples :

```
>> freeze(X, write(toto)).
X ~ tree.
>> freeze(X, write(toto)), X = 1 .
totoX = 1.
```

Un appel gelé ne se dégèle pas quand une variable est contrainte mais que son étiquette n'est pas connue :

```
>> freeze(X, dif(X,1)), X ~ cc(1,3).
X ~ cc(2,3).
```

Par contre, il se dégèle dès que cette valeur est connue :

```
>> freeze(X, dif(X,1)), X = 1 .
false.
```

La valeur n'a cependant pas besoin d'êntre entièrement connue. On note de plus dans cet exemple l'usage des parenthèses pour inscrire une suite de buts dans le second argument de **freeze/2** :

```
>> freeze(X, (write(toto),nl)), X = toto(Y).
toto
X = toto(Y),
Y ~ tree.
```

Voir également : nonvar/1.

get_istats/3
get_istats/2 — Lecture des statistiques sur les intervalles

Description : **get_istats**(a, b) unifie respectivement a et b avec les valeurs des compteurs suivants :

- Le nombre de points fixes exécutés,
- Le nombre de réévaluations de relations primaires effectuées dans le point fixe.

get_istats(a, b, c) unifie respectivement a, b et c avec les valeurs des compteurs suivants :

- Le nombre de points fixes exécutés,

– Le nombre de réévaluations de relations primaires effectuées dans le point fixe.

– La taille de la plus grande union d'intervalles.

Exemples :

```
>> reset_istats, intsplit([cc(1,4), cc(3, 6)]), false.
false.
>> get_istats(A, B).
B = 253,
A = 93.
>> get_istats(A, B, C).
C = 0,
B = 253,
A = 93.
>> set_prolog_flag(interval_mode,union).
true.
>> X ~ square(int n cc(1,10)).
X ~ 1 u 4 u 9 u 16 u 25 u 36 u 49 u 64 u 81 u 100.
>> get_istats(A,B,C).
C = 10,
B = 268,
A = 99.
>> set_prolog_flag(interval_mode,simple).
true.
```

Voir également : get_istats/2, reset_istats/0.

glb/2 — Borne inférieure

Description : **glb**(a, b) sert à récupérer la borne inférieure (en anglais, *greatest lower bound* du domaine de a dans la variable b.

Un appel à **glb**(a, b) échoue dans les cas suivants :

– a n'est pas d'un type compatible avec un type numérique,

– le domaine de a n'a pas de borne inférieure.

Les deux conditions reviennent à dire que a doit être une variable numérique avec une borne inférieure.

Il est important de remarquer que la valeur retournée par **glb/2** est une borne inférieure. Cette borne ne fait donc pas toujours partie du domaine de la variable, et en particulier quand la variable est contrainte à appartenir à un intervalle ouvert.

Exemples :

```
>> glb(1,B).
B = 1.
>> glb(cc(1,2), B).
B = 1.
>> glb(oo(1,2), B).
B = 1.
>> glb(pi, B).
B = 6588397/2097152.
>> glb(le(0), B).
false.
>> glb(ge(0), B).
B = 0.
```

Voir également : bounds/3, lub/2.

glblin/2 — Borne inférieure (linéaire)

Description : **glblin**(a, b) sert à récupérer la borne inférieure (en anglais, *greatest lower bound*) du domaine de a dans la variable b. Seules les contraintes linéaires sont prises en compte.

Un appel à **glblin**(a, b) échoue dans les cas suivants :

- a n'est pas d'un type compatible avec un type numérique,
- le domaine de a n'a pas de borne inférieure.

Les deux conditions reviennent à dire que a doit être une variable numérique avec une borne inférieure.

Il est important de remarquer que la valeur retournée par **glblin/2** est une borne inférieure. Cette borne ne fait donc pas toujours partie du domaine de la variable, en particulier quand la variable a été contrainte au moyen d'inégalités strictes (ou avec `dif`).

Exemples :

```
>> glblin(1,B).
B = 1.
>> glblin(lelin(0), B).
false.
>> glblin(gelin(7), B).
B = 7.
```

Voir également : boundslin/3, lublin/2.

int_size/2 — Mesure dans **Z** d'un domaine

Description : **int_size**(a, b) unifie b avec une mesure du domaine de a, qui est le nombre d'entiers contenus dans a. Si le domaine n'est pas borné, alors b est unifié avec `pinfinity`.

int_size/2 sert à mesurer la taille du domaine d'une variable, et est en particulier utile lors de la phase de sélection d'une variable sur laquelle énumérer.

Exemples :

```
>> int_size(1,S).
S = 1.
>> int_size(pi,S).
S = 0.
>> int_size(real,S).
S = pinfinity.
>> int_size(cc(1,2), S).
S = 2.
>> int_size(oo(1,2), S).
S = 0.
>> set_prolog_flag(interval_mode,union).
true.
>> int_size(cc(1,1000) n int, S).
S = 1000.
>> int_size(cc(1,2) u cc(4,7), S).
S = 6.
>> set_prolog_flag(interval_mode,simple).
true.
```

Voir également : float_size/2, real_size/2.

intsplit/1
intsplit/2
intsplit/3 — Enumération d'une liste d'entiers

Description : **intsplit**(a, b, c) enumère les entiers du domaine des éléments de la liste a en coupant en deux à chaque étape le domaine de la variable choisie suivant la valeur de b :

- $b =$ `smallest_domain` : Choix de la variable ayant le plus petit domaine,
- $b =$ `greatest_domain` : Choix de la variable ayant le plus grand domaine,
- $b =$ `list_order` : Choix de la première variable de la liste qui ne remplit pas la condition d'arrêt,
- $b =$ *my_choice* : Utilisation du prédicat *my_choice*(a, x) – défini par l'utilisateur –, qui unifie x avec l'une des variables de a.

Cette énumération s'arrête suivant la valeur de c :

- $c =$ `depth(`n`)` : Arrêt quand l'arbre d'énumération a atteint une pro-

fondeur de n, c'est-à-dire quand n choix ont été effectués.

- $c = \texttt{prec}(p)$, ou $c = p$: Arrêt quand toutes les variables de a ont un domaine de mesure inférieure à p, c'est-à-dire quand leur domaine contient au plus p entiers.
- $c = \textit{my\_stop}$: Utilisation du prédicat $\textit{my\_stop}(a)$ – défini par l'utilisateur –, un succès de ce prédicat signifie que la condition d'arrêt est atteinte.

Quand des prédicats définis par l'utilisateur sont utilisés, que ce soit pour la sélection de la variable ou pour le test d'arrêt, la liste a passée en argument contient l'état actuel de la liste a passée en paramètre de **intsplit/3**. L'utilisateur peut donc pendant son traitement faire toutes les opérations désirées sur cette liste. Toutefois, les algorithmes de sélection de variables et de test d'arrêt doivent être prudemment écrits, de manière à éviter des bouclages.

intsplit(a, b) enumère les entiers du domaine des éléments de la liste a en coupant en deux à chaque étape le domaine de l'entier choisi suivant la valeur de b, qui peut prendre les mêmes valeurs que pour **intsplit/3**. L'algorithme s'arrête quand chaque élément de la liste a ne contient plus qu'un seul entier. **intsplit/2** peut donc se définir à partir de **intsplit/3** par :

```
intsplit(A, B) :-
  intsplit(A, B, prec(1)).
```

intsplit(a) enumère les entiers du domaine des éléments de la liste a en coupant en deux à chaque étape le domaine de l'entier dont le domaine est le plus petit, jusqu'à ce que chaque élément de la liste a ne contienne plus qu'un seul entier. **intsplit/1** peut donc se définir à partir de **intsplit/3** par :

```
intsplit(A) :-
  intsplit(A, smallest_domain, prec(1)).
```

Exemples :

```
>> A ~ cc(1,20), intsplit([A], smallest_domain, prec(5)).
A ~ cc(1,5);
A ~ cc(6,10);
A ~ cc(11,15);
A ~ cc(16,20).
>> A ~ cc(1,20), B ~ cc(1,20), B ~ square(A),
    intsplit([A,B]).
B = 1,
A = 1;
B = 4,
A = 2;
B = 9,
A = 3;
B = 16,
A = 4.
>> A ~ cc(1,20), B ~ cc(1,20), B ~ square(A),
    intsplit([A,B], greatest_domain, depth(1)).
B ~ cc(1,4),
A ~ cc(1,2);
B ~ cc(9,16),
A ~ cc(3,4).
```

Voir également : boolsplit/2, realsplit/4.

iso/0 — Passage en mode iso

Description : **iso** permet le passage vers le mode `iso`, c.à.d. le mode syntaxique de la norme. Dans ce mode, les foncteurs prédéfinis sont laissés inchangés dans les règles et les requêtes (ils sont interprétés en mode `prolog4`).

Exemples :

```
>> X+2 = 2*X-4.
X = 6.
>> iso.
true.
```

```
?- X+2 = 2*X-4.
false.
```

Voir également : prolog4/0, :-iso/0 (directive).

lub/2 — Borne supérieure

Description : **lub**(a, b) sert à récupérer la borne supérieure (en anglais, *lowest upper bound*) du domaine de a dans la variable b.

Un appel à **lub**(a, b) échoue dans les cas suivants :

- a n'est pas d'un type compatible avec un type numérique,
- le domaine de a n'a pas de borne supérieure.

Les deux conditions reviennent à dire que a doit être une variable numérique avec une borne supérieure.

Il est important de remarquer que la valeur retournée par lub/3 est une borne supérieure. Cette borne ne fait donc pas toujours partie du domaine de la variable, en particulier quand elle est contrainte à appartenir à un intervalle ouvert.

Exemples :

```
>> lub(1, B).
B = 1.
>> lub(cc(1,2), B).
B = 2.
>> lub(oo(1,2), B).
B = 2.
>> lub(pi, B).
B = 13176795/4194304.
>> lub(le(0), B).
B = 0.
>> lub(ge(0), B).
false.
```

Voir également : bounds/3, glb/2.

lublin/2 — Borne supérieure (linéaire)

Description : **lublin**(a, b) sert à récupérer la borne supérieure (en anglais, *lowest upper bound*) du domaine de a dans la variable b. Seules les contraintes linéaires sont prises en compte.

Un appel à **lublin**(a, b) échoue dans les cas suivants :

- a n'est pas d'un type compatible avec un type numérique,
- le domaine de a n'a pas de borne supérieure.

Les deux conditions reviennent à dire que a doit être une variable numérique avec une borne supérieure.

Il est important de remarquer que la valeur retournée par **lublin/2** est une borne supérieure. Cette borne ne fait donc pas toujours partie du domaine de la variable, en particulier quand la variable a été contrainte au moyen d'inégalités strictes (ou avec `dif`).

Exemples :

```
>> lublin(1, B).
B = 1.
>> lublin(lelin(3), B).
B = 3.
>> lublin(ltlin(3), B).
B = 3.
>> lublin(gelin(3), B).
false.
```

Voir également : boundslin/3, glblin/2.

maximizelin/1 — Borne supérieure (linéaire)

Description : **maximizelin**(a) impose à la variable a d'atteindre sa borne supérieure. Les contraintes linéaires sont seules prises en compte.

Un appel à **maximizelin**(a) échoue dans les cas suivants :

- a n'est pas d'un type compatible avec un type numérique,
- le domaine de a n'a pas de borne supérieure.
- cette borne supérieure ne peut être atteinte par a.

Les deux conditions reviennent à dire que a doit être une variable numérique avec une borne supérieure atteignable.

Cette borne ne fait pas toujours partie du domaine de la variable, en particulier quand la variable a été contrainte au moyen d'inégalités strictes (ou avec `dif`).

Exemples :

```
>> maximizelin(1).
B = 1.
>> X ~ lelin(3), maximizelin(X).
X = 3.
>> dif(X, 3), X ~ lelin(3), maximizelin(X).
false.
>> X ~ ltlin(3), maximizelin(X).
false.
>> maximizelin(gelin(3)).
false.
```

Voir également : boundslin/3, lublin/2, minimizelin/1.

minimizelin/1 — Borne inférieure (linéaire)

Description : **minimizelin**(a) impose à la variable a d'atteindre sa borne inférieure. Les contraintes linéaires sont seules prises en compte.

Un appel à **minimizelin**(a) échoue dans les cas suivants :

- a n'est pas d'un type compatible avec un type numérique,
- le domaine de a n'a pas de borne inférieure.
- cette borne inférieure ne peut être atteinte par a.

Les deux conditions reviennent à dire que a doit être une variable numérique avec une borne inférieure atteignable.

Cette borne ne fait donc pas toujours partie du domaine de la variable, en particulier quand la variable a été contrainte au moyen d'inégalités strictes (ou avec `dif`).

Exemples :

```
>> minimizelin(1).
B = 1.
>> X ~ gelin(3), minimizelin(X).
X = 3.
>> dif(X, 3), X ~ gelin(3), minimizelin(X).
false.
>> X ~ gtlin(3), minimizelin(X).
false.
>> minimizelin(lelin(3)).
false.
```

Voir également : boundslin/3, glblin/2, maximizelin/1.

max_float/1 — Plus grand rationnel IEEE

Description : **max_float**(a) unifie a avec le plus grand rationnel IEEE, qui est l'entier 340282346638528859811704183484516925440.

Exemples :
```
>> max_float(A).
A = 340282346638528859811704183484516925440.
```

Voir également : float_rank/2, min_positive_float/1.

min_positive_float/1 Plus petit nombre flottant positif représentable

Description : **min_float**(a) unifie a avec le plus petit rationnel IEEE positif, soit $\frac{1}{713623846352979940529142984724747568191373312}$.

Exemples :
```
>> min_positive_float(A).
A = 1/713623846352979940529142984724747568191373312.
```

Voir également : float_rank/2, max_float/1.

no_debug/0 — Sortie du mode debug

Description : **no_debug** rend inactif le débogueur. Les affichages du débogueur ainsi que ses surveillances sont désactivées.

On rappelle que le débogueur ne peut montrer des informations que sur les règles qui ont été compilées avec les primitives de la famille `compile`.

Voir le chapitre « Environnement » pour de plus amples détails sur le débogueur.

Voir également : debug/0, compile/n.

prolog4/0 — Passage en mode prolog4

Description : **prolog4** permet le passage vers le mode `prolog4`, c.à.d. le mode syntaxique naturel de Prolog IV. Dans ce mode, certains foncteurs prédéfinis sont interprétés dans les règles et les requêtes.

Exemples :
```
?- 1+2 = 7-4.
false.
?- prolog4.
true.
```

```
>> 1+2 = 7-4.
true.
```

Voir également : iso/0, :-prolog4/0 (directive).

real_size/2 — Mesure dans **R** d'un domaine

Description : **real_size**(a, b) unifie b avec une mesure du domaine de a, qui est la taille du domaine de a dans **R**. Si le domaine n'est pas borné, alors b est unifié avec `pinfinity`.

real_size/2 sert à mesurer la taille du domaine d'une variable, et est en particulier utile lors de la phase de sélection d'une variable sur laquelle énumérer.

En mode intervalles simples, si L est le rang de la borne inférieure du domaine de a et que U le rang de la borne supérieure de a, alors la mesure b peut être calculée par :

- $b = U - L$ si le domaine de a est borné des deux côtés,
- `pinfinity` sinon.

En mode unions d'intervalles, la mesure du domaine d'une variable est la somme des mesures de chaque intervalle le composant.

Exemples :

```
>> real_size(1,S).
S = 0.
>> real_size(pi,S).
S = 1/4194304.
>> real_size(real,S).
S = pinfinity.
>> real_size(cc(1,2),S).
S = 1.
>> real_size(oo(1,2),S).
S = 1.
>> set_prolog_flag(interval_mode,union).
true.
```

Dans l'exemple ci-dessous, on voit que les intervalles réduits à un singleton ont une taille de 0, et que $1000 \times 0 = 0$.

```
>> real_size(cc(1,1000) n int, S).
S = 0.
>> real_size(cc(1,2) u cc(4,7), S).
S = 4.
>> set_prolog_flag(interval_mode,simple).
true.
```

Voir également : float_size/2, int_size/2.

realsplit/1
realsplit/2
realsplit/3
realsplit/4 — Enumération d'une liste de réels

Description : **realsplit**(a, b, c, d) enumère les réels du domaine des éléments de la liste a en coupant en deux à chaque étape le domaine de la variable choisie suivant la valeur de b :

- $b =$ smallest_domain : Choix de la variable ayant le plus petit domaine,
- $b =$ greatest_domain : Choix de la variable ayant le plus grand domaine,
- $b =$ list_order : Choix de la première variable de la liste dont la mesure du domaine est supérieure à la précision requise,
- $b =$ *my_choice* : Utilisation du prédicat *my_choice*(a, x) – défini par l'utilisateur –, qui unifie x avec l'une des variables de a.

Cette énumération s'arrête suivant la valeur de c :

- $c =$ depth(n) : Arrêt quand l'arbre d'énumération a atteint une profondeur de n, c'est-à-dire quand n choix ont été effectués.
- $c =$ prec(p) ou $c = p$: Arrêt quand tous les réels de a sont dans un domaine de mesure inférieure à p,
- $c =$ my_stop(a) : Utilisation du prédicat *my_stop*(a) – défini par l'utilisateur –, une réussite de ce prédicat signifie que la condition d'arrêt est atteinte.

La méthode de mesure des domaines est fonction de d :

- $d =$ `real_mode` : Mesure sur **R** (voir real_size/2),
- $d =$ `float_mode` : Mesure sur les flottants (voir float_size/2).

La méthode de mesure est utilisée à la fois pour sélectionner les variables sur lesquelles porte l'énumération et pour effectuer les tests d'arrêt.

Si certains des arguments b, c et d ne sont pas instanciés lors de l'appel à **realsplit/4**, ils sont unifiés avec la valeur par défaut sélectionnée.

realsplit(a, b, c) enumère les réels du domaine des éléments de la liste a en coupant en deux à chaque étape le domaine du réel choisi suivant la valeur de b. L'énumération s'arrête quand une condition indiquée par c est atteinte. Les valeurs possibles de b et c sont les mêmes que ci-dessus. Toutes les mesures sont effectuées sur **R**, ce qui correspond au `real_mode` de **realsplit/4**.

realsplit(a, b) enumère les réels du domaine des éléments de la liste a en coupant en deux à chaque étape le domaine du réel choisi suivant la valeur de b. L'énumération s'arrête quand La précision des calculs est de 10^{-4}. Toutes les mesures sont effectuées sur **R**, ce qui correspond au `real_mode` de **realsplit/4**.

realsplit(a) enumère les réels du domaine des éléments de la liste a en coupant en deux à chaque étape le domaine du réel dont le domaine est le plus petit, ce qui correspond à l'option `smallest_domain` de **realsplit/4**. L'énumération s'arrête quand La précision des calculs est de 10^{-4}. Toutes les mesures sont effectuées sur **R**, ce qui correspond qu `real_mode` de **realsplit/4**.

Ces trois versions peuvent donc aisément être définies à partir de **realsplit/4** de la manière suivante :

```
realsplit(A, B, C) :-
    realsplit(A, B, C, real_mode).

realsplit(A, B) :-
    realsplit(A, B, prec(1/10000), real_mode).

realsplit(A) :-
    realsplit(A, smallest_domain, prec(1/10000), real_mode).
```

[large]

Exemples :

```
>> A ~ cc(1,pi), realsplit([A], B, prec(1)).
B = smallest_domain,
A ~ cc(1,'>1.5353982');
B = smallest_domain,
A ~ oc('>1.5353982','>2.0707964');
B = smallest_domain,
A ~ oo('>2.0707964','>2.6061944');
B = smallest_domain,
A ~ co('>2.6061944','>3.1415927').
```

Voir également : boolsplit/2, intsplit/3.

recompile/1
recompile/2 — Recompilation d'un fichier

Types : recompile(+atome)
recompile(+atome, @liste)

Description : Très similaire à la primitive `compile/`n, **recompile**(a) lit une suite de règles dans le fichier de nom a. Les règles lues sont compilées et entrées en mémoire. Si un paquet de règles de mêmes nom et arité figure déjà dans la base de règles, il serait simplement redéfini sans qu'une erreur ne soit générée.

recompile/2 est le pendant le la primitive `compile/2`.

Voir les primitives `compile/`n pour la description des options et les exemples.

Voir également : compile/n, consult/n, debug/0.

reconsult/0
reconsult/1 — Reconsultation d'un fichier

Types : reconsult
reconsult(+atome)

Description : **reconsult**(a) lit une suite de règles dans le fichier de nom a. Les règles lues sont traduites puis entrées en mémoire. Si un paquet dynamique de règles de mêmes nom et arité figure déjà dans la base de règles, il serait simplement redéfini sans qu'une erreur ne soit générée.

reconsult/0 est le pendant le la primitive `consult/0`.

Voir la description et les exemples donnés pour `consult/`n pour compléments d'information.

Voir également : consult/n, compile/n.

record/2 — Affectation de variable globale

Types : record(+atome, @terme)

Description : **record**(a, b) associe le terme b à l'atome donné dans a. Tout se passe comme si a était le nom d'une variable «globale» prennant b comme valeur. Il s'agit donc bien de l'affectation classique, comme elle se pratique en FORTRAN, PASCAL, etc...

Exemples :

```
>> record(age, 21).
true.
>> write(age), nl.
age
true.
>> recorded(age, X).
X = 21.
```

Voir également : recorded/2.

record/2 — Affectation d'un élément de tableau

Types : record(+élément_de_tableau, @terme)

Description : **record**$(a(i), b)$ associe le terme b à l'élément de rang i du tableau a.

Exemples :

```
>> def_array(tab, 10).
true.
>> N ~ cc(1, 10), enum(N), record(tab(N), N.*.N).
N = 1;
N = 2;
N = 3;
N = 4;
N = 5;
N = 6;
N = 7;
N = 8;
N = 9;
N = 10.
>> recorded(tab(6), X).
X = 36.
>> undef_array(tab).
true.
```

Voir également : def_array/2, recorded/2, undef_array/1.

recorded/2 — Evaluation d'une variable globale

Types : recorded(+atome, ?terme)
recorded(+élément_de_tableau, ?terme)

Description : **recorded**(a, b) produit le résultat b, en fonction de a qui doit être un identificateur ou un élément de tableau, en respectant les règles suivantes :

- la valeur associée à un tableau indicé est égale à la valeur associée à l'élément correspondant de ce tableau,
- la valeur associée à un identificateur i est définie comme suit :
  - si un terme t a été associé à i (au moyen de la primitive `record/2`), alors la valeur associée est t,
  - sinon, i n'a pas fait l'objet d'une association préalable, et la valeur associée à i est lui-même.

Exemples :

```
>> record(age, 21).
true.
>> write(age), nl.
age
true.
>> recorded(age, X).
X = 21.
>> def_array(tab, 10).
true.
>> N ~ cc(1, 10), enum(N), record(tab(N), N.*.N).
N = 1;
N = 2;
N = 3;
N = 4;
N = 5;
N = 6;
N = 7;
N = 8;
N = 9;
N = 10.
>> recorded(tab(6), X).
X = 36.
>> undef_array(tab).
true.
```

Voir également : def_array/2, record/2, undef_array/1.

redef_array/2 — Redéfinition d'un tableau

Types : redef_array(+atome, +entier)

Description : **redef_array**(a, b) (re)définit dynamiquement un tableau de termes, de nom a et de longueur b.

Si un tableau de même nom existe déjà, aucune erreur n'est produite (à moins que la taille à réserver ne soit trop grande). Dans ce cas, soit le tableau est redéfini de plus petite taille, et les éléments de rang supérieur à la nouvelle taille sont perdus, soit le tableau est redéfini de plus grande taille, et les nouveaux éléments sont initialisés à zéro.

Voir la primitive `def_array/2` pour d'autres informations.

Exemples :

```
>> def_array(tab, 10).
true.
>> N ~ cc(1, 10), enum(N), record(tab(N), N.*.N).
N = 1;
N = 2;
N = 3;
N = 4;
N = 5;
N = 6;
N = 7;
N = 8;
N = 9;
N = 10.
>> recorded(tab(5), X).
X = 25.
>> recorded(tab(6), X).
X = 36.
>> redef_array(tab,5).
true.
>> recorded(tab(5), X).
X = 25.
>> recorded(tab(6), X).
error: error(prologIV_error(array,6),recorded/2)

>> redef_array(tab,10).
true.
>> recorded(tab(5), X).
X = 25.
>> recorded(tab(6), X).
X = 0.
```

Voir également : def_array/2, record/2, recorded/2, undef_array/1.

reset_istats/0 — Réinitialisation des statistiques sur les intervalles

Description : **reset_istats** remet trois compteurs à zéro :

- Le nombre de points fixes exécutés,
- Le nombre de réévaluations de relations primaires effectuées dans le point fixe,
- La taille de la plus grande union d'intervalles.

Cette primitive est utilisée en conjonction avec `get_istats/2` et `get_istats/3`.

Exemples :

```
>> reset_istats.
true.
```

Voir également : get_istats/2, get_istats/3.

reset_cpu_time/0 — Réinitialisation du chronomètre

Description : **reset_cpu_time** remet le chronomètre à zéro. Cette primitive est utilisée en conjonction avec `cpu_time/1`.

Exemples :

```
>> reset_cpu_time, cpu_time(T).
T = 0.
```

Voir également : cpu_time/1.

system/1 — Appel système

Types : system(+atome)

Description : **system**(a) effectue la commande contenue dans la chaîne repésentée par l'atome a. On fait appel au système d'exploitation pour l'exécution de cette commande. Les commandes effectuées par le biais de cette primitive sont en général locales à celle-ci ; en particulier, on ne peut changer le répertoire de travail courant de façon durable avec cette primitive : il faut utiliser pour cela `chdir/1`. Le comportement précis de cette primitive, notamment en ce qui concerne ses effets de bord, est bien sûr non-portable. Toutefois :

- Elle génère une erreur si l'argument n'est pas un atome.
- Elle échoue si le système d'exploitation retourne une erreur en lançant la commande.
- Elle réussit dans les autres cas.

Exemples :

```
>> system(pwd).
/mon/repertoire/de/travail
true.
>> system('cd /mon/repertoire').
true.
>> system(pwd).
/mon/repertoire/de/travail
true.
>>
```

Note : Il faut entourer l'atome avec des apostrophes s'il contient des blancs ou des caractères autres que des lettres ou des chiffres, ou s'il commence par une majuscule.

Voir également : chdir/1

undef_array/1 — Destruction de tableau

Types : undef_array(+atome)

Description : **undef_array**(a) libère le tableau de nom a créé par la primitive `def_array/2`. Le tableau a ne peut plus être utilisé sans une nouvelle définition au moyen de `def_array/2`.

Exemples :

```
>> def_array(tab, 10).
true.
>> N ~ cc(1, 10), enum(N), record(tab(N), N.*.N).
N = 1;
N = 2;
N = 3;
N = 4;
N = 5;
N = 6;
N = 7;
N = 8;
N = 9;
N = 10.
>> recorded(tab(6), X).
X = 36.
>> undef_array(tab).
true.
>> recorded(tab(6), X).
error: error(prologIV_error(undefined_array,tab(6)),
 recorded/2)
```

Voir également : def_array/2, record/2, recorded/2.

5

Prédicats prédéfinis ISO

CE CHAPITRE comporte la liste exhaustive des prédicats prédéfinis et des fonctions évaluables définis par la norme Prolog ISO IEC DIS 13211-1 .

5.1 Introduction

Ces prédicats et fonctions sont classés par catégorie, et suivent la classification introduite par la norme. Les catégories sont les suivantes :

1. égalité et inégalité,
2. tests de types,
3. comparaison de termes,
4. création et décomposition de termes,
5. évaluation arithmétique,
6. comparaisons arithmétiques,
7. recherche de clauses,
8. création et suppression de clauses,
9. calcul de toutes les solutions,
10. sélection et contrôle des flux d'entrée et de sortie,
11. entrées-sorties de caractères,
12. entrées-sorties de mots machine,
13. entrées-sorties de termes,
14. contrôle,
15. traitement de termes atomiques,
16. prédicats divers,
17. fonctions évaluables.

5.1.1 Liste des primitives ISO par catégories

Egalité et inégalité

=/2 égalité (unification).

\=/2 non-égalité (non-unification).

unify_with_occurs_check/2 égalité avec test d'occurrence.

Tests de types

atom/1 test d'identificateur (d'atome).

atomic/1 test de constante (nombres, atomes).

compound/1 test de terme composé.

float/1 test de flottant.

integer/1 test d'entier.

nonvar/1 test de non-variable.

number/1 test de nombre.

rational/1 test de nombre rationnel.

var/1 test de variable.

Comparaison de termes

==/2 termes identiques (égalité formelle).

\==/2 termes non-identiques.

@</2 terme inférieur à.

@=</2 terme inférieur ou égal à.

@>/2 terme supérieur à.

@>=/2 terme supérieur ou égal à.

Création et décomposition de termes

arg/3 sélection d'un argument dans un terme.

functor/3 (dé)compose un terme en étiquette et arité.

=../2 (dé)compose un terme en étiquette et liste de fils.

copy_term/2 copie de terme (avec création de nouvelles variables).

Evaluation arithmétique

is/2 évaluation d'une formule numérique. Voir la section « Fonctions évaluables ».

Comparaisons arithmétiques

Voir la section « Fonctions évaluables » pour l'évaluation des arguments.

</2, =</2, >=/2, >/2 évaluent leurs arguments, puis les comparent.

=:= évalue ses arguments, puis détermine si leurs valeurs sont égales.

=\= évalue ses arguments, puis détermine si leurs valeurs sont différentes.

Recherche de clauses

clause/2 recherche de clauses.

current_predicate/1 recherche de procédure.

Création et suppression de clauses

abolish/1 suppression d'une procédure (paquet de règles).

asserta/1 ajout d'une clause en tête de sa procédure.

assertz/1 ajout d'une clause à la fin de sa procédure.

retract/1 suppression de clauses.

Calcul de toutes les solutions

^/2 notation de variables existentielles (notation pour d'autres primitives de la famille).

bagof/3 liste de solutions.

findall/3 liste de solutions.

setof/3 liste de solutions (triée).

Sélection et contrôle des flux d'entrée et de sortie

at_end_of_stream/0 /1 test de la fin d'un flux.

close/1 /2 fermeture d'un flux.

current_input/1 flux d'entrée courant.

current_output/1 flux de sortie courant.

flush_output/1 /2 vidange du tampon d'écriture.

open/3 /4 ouverture d'un flux d'entrée ou de sortie.

set_input/1 sélection du flux d'entrée courant.

set_output/1 sélection du flux de sortie courant.

set_stream_position/2 repositionnement d'un flux.

stream_property/2 propriétés d'un flux.

Entrées-sorties de caractères

get_char/1 /2 lecture d'un caractère.

get_code/1 /2 lecture du code d'un caractère.

nl/0 /1 écriture d'une fin de ligne.

peek_char/1 /2 prochain caractère à lire.

peek_code/1 /2 code du prochain caractère à lire.

put_char/1 /2 écriture d'un caractère.

put_code/1 /2 écriture du code d'un caractère.

Entrées-sorties de mots machine

get_byte/1 /2 lecture d'un octet.

peek_byte/1 /2 prochain octet à lire.

put_byte/1 /2 écriture d'un octet.

Entrées-sorties de termes

read_term/2 /3 lecture d'un terme.

read/1 /2 lecture d'un terme.

op/3 déclaration d'opérateurs.

current_op/3 obtenir les caractéristiques des opérateurs déclarés.

write/1 /2 Ecriture d'un terme (pas forcément lisible par `read`).

writeq/1 /2 Ecriture d'un terme. Ce qui a été écrit peut être relu par les primitives `read`.

write_term/3 /2 Ecriture d'un terme. C'est la primitive principale, avec paramètres.

write_canonical/1 /2 Ecriture d'un terme sans l'emploi d'opérateurs (lisible par `read`).

Contrôle

!/0 (cut) coupure des points de choix.

\+ **/1** appel non-effaçable .

,/2 conjonction de littéraux.

;/2 disjonction de littéraux.

->/2 si-alors de littéraux.

->; /3 si-alors-sinon de littéraux.

call/1 méta-appel d'un littéral.

catch/3 gestion d'erreur (rattrapage).

fail/0 échec.

once/1 méta-appel d'un littéral (exécution unique).

repeat/0 multiples exécutions.

throw/1 gestion d'erreur (génération).

true/0 succès.

Traitement de termes atomiques

atom_chars/2 éclatement d'un atome.

atom_codes/2 éclatement d'un atome.

atom_concat/3 concaténation d'atomes.

atom_length/2 nombre de caractères d'un atome.

char_code/2 correspondance caractère/code.

number_chars/2 éclatement d'un nombre.

number_codes/2 éclatement d'un nombre.

sub_atom/5 découpage d'un atome.

Prédicats divers

current_prolog_flag/2 interrogation de paramètres.

halt/0 /1 sortie de prolog.

set_prolog_flag/2 mise à jour de paramètres.

Fonctions évaluables

+ /2	**- /2**	*** /2**	**/ /2**	**- /1**
// /2	**rem/2**	**mod/2**		
**** /2**	**sign/1**	**abs/1**	**sqrt/1**	
sin/1	**cos/1**	**atan/1**	**exp/1**	**log/1**
float/1	**round/1**	**floor/1**	**ceiling/1**	**truncate/1**
float_integer_part		**float_fractional_part**		

Ces fonctions sont reconnues par les primitives de la famille `is/2`. Voir *Evaluation arithmétique* et *Comparaisons arithmétiques*.

5.1.2 Types

Le type de chacun des arguments d'un prédicat prédéfini sera spécifié par l'un des identifiants suivants :

`atome` : un atome (identificateur).

`liste_d'atomes` : une liste d'atomes.

`atomique` : un terme atomique (atome ou nombre).

`octet` : un entier entre 0 et 255.

`caractère` : un atome de longueur 1.

`code_caractère` : un entier correspondant au code d'un caracère (selon le jeu de caractère, cet entier peut être plus grand que 255).

`liste_de_codes_caractères` : une liste de `code_caractère`

`liste_de_caractères` : une liste de `caractère`.

`clause` : une règle (ou un fait).

`options_pour_close` : liste d'options reconnues par les primitives `close`.

`évaluable` : une expression composée de termes dont les foncteurs figurent dans la liste des fonctions évaluables, ainsi que de nombres et de variables.

`paramètre` : un atome associé à un paramètre Prolog IV.

`terme_composé` : un terme dont le nœud principal a un ou plusieurs fils.

`tête` : le premier littéral d'une règle.

`entier` : un entier.

`octet_entré` : un `octet` ou l'entier -1.

`caractère_entré` : un octet ou l'atome `end_of_file`.

`code_caractère_entré` : un `code_caractère`.

`mode_es` : un mode d'entré/sortie (`read`, `write` ou `append`).

`liste` : une liste (avec `[]` à la fin).

`nonvar` : un terme atomique ou un terme composé.

`nombre` : un entier, rationnel ou un flottant.

`spécificateur` : l'un des atomes `xf`, `yf`, `xfx`, `xfy`, `yfx`, `fx` ou `fy`. Utilisé pour spécifier la classe (postfixe, préfixe ou infixe) et l'associativité des opérateurs.

`indicateur_de_prédicat` : un terme composé de la forme A/N, où A est un atome et N un entier, et qui désigne une procédure.

`options_pour_read` : une liste d'options reconnues par les primitives de la famille `read`.

`source_ou_puits` : un endroit (comme un fichier) ou des données peuvent être lues ou écrites.

`flux` : la connexion avec un `source_ou_puits`.

`options_pour_open` : liste de termes qui spécifient des caractéristiques d'ouverture d'un flux.

`flux_ou_alias` : un flux ou un alias.

`position_dans_flux` : un nombre décrivant la position courante dans le `source_ou_puits` associé au flux.

`propriété_de_flux` : un terme représentant une des caractéristiques d'un flux.

`terme` : un terme atomique, un terme composé ou une variable.

`options_pour_write` : liste d'options reconnues par les primitives `write_term`

`terme_exécutable` : un atome ou un terme composé.

5.1.3 Modes

Les différents modes possibles pour les arguments seront exprimés par les caractères suivants qui précèdent le descripteur de type :

`+` : L'argument doit être instancié.

`?` : L'argument doit être instancié ou doit être une variable.

`@` : Il n'y a pas de modification de l'argument.

`-` : L'argument doit être une variable qui sera instanciée si et seulemement si le prédicat réussit.

5.1.4 Contraintes

Les ensembles de contraintes mentionnés dans la description des prédicats prédéfinis seront généralement exprimés en notation mathématique standard. Les variables seront notamment indicées si nécessaire. Nous noterons S le système courant de contraintes.

5.2 Préalables

5.2.1 A propos des comparaisons de termes

Prédicats portant sur des termes

Rigoureusement conformes à la norme ISO du langage Prolog, les six prédicats prédéfinis décrits dans cette section prennent en charge la comparaison des termes qui sont leurs arguments. Il faut remarquer qu'il s'agit là d'un comportement original. En effet, un terme, associé à un système de contraintes, représente en Prolog IV un ensemble d'arbres (les arbres sont les éléments du domaine de Prolog). Par conséquent, aussi bien les prédicats prédéfinis que les prédicats écrits par l'utilisateur expriment en règle générale des propriétés et des traitements qui portent sur de tels ensembles d'arbres, non sur les termes qui les représentent.

On a donc du mal à expliquer simplement ce que font les prédicats de comparaison de termes, le plus difficile étant d'exprimer quels sont les termes effectivement pris en compte. Par exemple, le but

```
?- X == Y.
```

échoue, car les variables `X` et `Y` sont des termes distincts, alors que le but

```
?- X = Y, X == Y.
```

réussit, révélant une certaine confusion entre l'aspect formel de `X` et `Y` et les ensembles d'arbres que ces variables représentent.

Une manière d'expliquer les choses, du moins en l'absence d'arbres infinis, consiste à dire que si *T* est un argument d'un des prédicats qui nous intéressent ici, alors le terme effectivement pris en compte par un tel prédicat est le terme *T'* que l'on obtient en remplaçant dans *T* chaque variable *V* par le terme *W*, si

- le système de contraintes courantes implique *V* = *W*,
- *W* n'est pas une variable, ou *W* précède *V*.

Dans les explications suivantes, nous dirons que *T'* est le *terme complètement instancié* représenté par *T*, compte tenu du système de contraintes courantes. On peut noter que, à peu de choses près, *T'* est le terme défini par l'expression écrite de *T* que donnerait Prolog.

Ordre des termes

La relation d'ordre sur les termes est définie de la manière suivante :

- Si T_1 et T_2 sont des termes identiques, alors ni T_1 ne précède T_2, ni T_2 ne précède T_1.
- Tout terme de type variable précède tout terme de type nombre flottant ; tout terme de type nombre flottant précède tout terme de type nombre rationnel ; tout terme de type nombre rationnel précède tout atome ; enfin, tout atome précède tout terme composé.

  Rappelons que les nombres flottants ne peuvent être entrés avec la notation décimale habituelle que si Prolog IV se trouve dans le mode `iso`. Dans le mode `prolog4`, tous les nombres sont des rationnels (des nombres exacts).

- Si T_1 et T_2 sont deux variables distinctes, leur position relativement à l'ordre des termes dépend de l'implémentation. En Prolog IV, une variable est supérieure à une autre si elle a été créée plus tard.
- Si T_1 et T_2 sont de type numérique, leur position relativement à l'ordre des termes est déterminée par la relation d'ordre des nombres. T_1 @=< T_2 équivaut donc dans ce cas à $T_1 =< T_2$.
- Si T_1 et T_2 sont des atomes, alors T_1 précède T_2 si et seulement si la chaîne de caractères associée à T_1 est inférieure, pour l'ordre lexicographique, à la chaîne de caractères associée à T_2.
- Si T_1 et T_2 sont des termes composés, alors T_1 précède T_2 si et seulement si :
  - l'arité de T_1 est inférieure à celle de T_2, ou bien
  - T_1 et T_2 ont la même arité et le nom du foncteur de T_1 précède le nom du foncteur de T_2, ou bien
  - T_1 et T_2 ont le même foncteur (même arité et même nom), et il existe un entier k tel que pour chaque i compris entre 1 et $k-1$, le i^{eme} argument de T_1 et le i^{eme} argument de T_2 sont des termes identiques, et le k^{eme} argument de T_1 précède le k^{eme} argument de T_2.

5.2.2 A propos des entrées-sorties

Sources et puits

Les entités extérieures à un programme Prolog IV avec lesquelles ce dernier échange des données sont les *sources* et les *puits*. Une source est susceptible de fournir des données à un programme Prolog, à la demande de ce dernier. Un puits est capable de consommer les données qu'un programme Prolog lui transmet. Des exemples de sources et de puits sont un fichier, une console, un tube de communication entre processus, etc.

Les sources et les puits sont toujours vus comme des suites d'octets ou de caractères. Ces suites sont finies ou infinies. Elles ont toujours un début, parfois elles n'ont pas de fin.

Chaque système d'exploitation spécifie une manière de donner des noms aux sources et aux puits. Ces noms apparaissent dans les programmes Prolog uniquement comme valeurs du premier argument des prédicats `open`/3 et `open`/4.

Modes

Les sources et les puits peuvent être utilisés au moins selon les trois modes suivants :

Lecture : l'entité extérieure au programme est une source. Si elle correspond à un fichier, celui-ci doit exister et son contenu sera transmis au programme à partir du début du fichier.

Ecriture : l'entité extérieure au programme est un puits. Il peut exister ou non. S'il n'existe pas il sera créé. S'il s'agit d'un fichier existant, il sera remis à zéro (i.e. entièrement vidé) avant son utilisation par le programme.

Allongement : l'entité extérieure au programme est un puits. Il peut exister ou non. S'il s'agit d'un fichier existant, il ne sera pas remis à zéro ; à la place de cela, les données transmises depuis le programme seront placées à la fin du fichier, à la suite des données existantes.

Flux

La vue logique que l'on a d'une source ou d'un puits depuis un programme Prolog s'appelle un *flux*. Durant l'exécution d'un programme, chaque flux est représenté par un *descripteur de flux*, crée par l'exécution du prédicat `open` et détruit par celle du prédicat `close`.

Le programmeur Prolog ne doit pas faire d'hypothèses sur la nature précise d'un descripteur de flux, sauf pour ce qui est des propriétés suivantes :

- un descripteur de flux est un terme sans variable,
- ce n'est pas un atome,
- un descripteur de flux ne fait référence à une source ou un puits que pendant que la source ou le puit en question est ouvert.

Un flux peut aussi être représenté dans un programme par un *alias*, qui est un atome choisi par le programmeur, associé à une source ou à un puits lors de l'ouverture du flux. Plusieurs alias peuvent représenter le même flux.

Flux standard et flux courants

Deux flux particuliers sont prédéfinis et ouverts durant toute session Prolog : le *flux standard d'entrée* et le *flux standard de sortie*. Ils sont respectivement représentés par les alias `user_input` et `user_output`.

Parmi les flux ouverts, deux sont distingués à tout instant durant une session Prolog : le *flux d'entrée courant* et le *flux de sortie courant*. Lorsqu'un prédicat prédéfini correspondant à une opération de lecture [resp. d'écriture] ne comporte pas un argument indiquant le flux concerné, c'est que l'opération doit concerner le flux d'entrée [resp. de sortie] courant.

Par défaut, les flux d'entrée et de sortie courants sont respectivement le flux standard d'entrée et le flux standard de sortie, i.e. `user_input` et `user_output`.

Flux de texte, flux binaires

Un flux de texte est vu par un programme Prolog IV comme formé d'une suite de lignes. Une ligne est une suite, éventuellement vide, de caractères suivis d'un caractère particulier qui en indique la fin ; en Prolog IV, il s'agit du caractère '`\n`'. Le programmeur Prolog n'a pas à faire d'hypothèse sur la manière dont le système d'exploitation représente effectivement les lignes dans les sources et les puits : les fonctions de lecture feront en sorte que chaque fin de ligne se traduise par exactement un caractère '`\n`'. (Il en résulte que les caractères lus ne sont pas toujours exactement ceux qui se trouvent dans la source.)

Un flux binaire est une suite d'octets auxquels les prédicats d'entrée et sortie n'attribuent aucune interprétation particulière. Des lectures successives d'octets depuis un flux binaire (par le prédicat `get_code`/1) fournissent exacte-

ment les valeurs qui composent la source. Si ces octets sont successivement écrits dans un deuxième flux binaire (prédicat `put_code`/1) alors le puits créé est identique à la source lue.

Fin d'un flux

Lorsque tous les caractères disponibles dans un flux ouvert en lecture ont été lus avec succès, le flux se trouve dans la position « sur la fin du flux ». Si une opération de lecture a lieu à ce moment-là, la donnée acquise est une valeur conventionnelle qui indique l'épuisement du flux, et ce dernier passe dans la position « au-delà de la fin du flux ». Le comportement que doit adopter le flux si d'autres lectures sont tentées ensuite est défini par le programmeur (cf. prédicat `stream_property`/2).

5.2.3 A propos des règles

Mini-glossaire

clause : une règle (ou un fait).

règle prédéfinie : une fonctionnalité fournie avec le système Prolog IV.

règle utilisateur : toute règle entrée par consult, compile, assert et leurs familles.

primitive : règle prédéfinie.

prédicat : un paquet de règles.

procédure : un paquet de règles.

Règles statiques

Un paquet de règles statiques ne peut être modifié (par suppression ou ajout de règles), et ses règles ne peuvent être récupérées sous forme de termes.

Règles prédéfinies Elles font partie du système Prolog IV. On ne peut ni les voir, ni les modifier, ni les enlever, ni même les désactiver. Elles restent donc toujours disponibles pour utilisation.

Certaines des règles prédéfinies sont dûment documentées (on les trouvent dans les chapitres « Primitives. . . » et « Relations. . . ». Ces primitives sont bien sûr destinées à être utilisés par le programmeur ; on les appelera « règles publiques ». Mais il existe également des règles auxiliaires qui ne sont là que pour des besoins internes au système Prolog IV. Les règles « publiques » les utilisent (citons comme exemple le compilateur). Pour éviter que les noms de règles du programmeur n'interfèrent avec les noms des règles auxiliaires (dont la liste n'est jamais fournie), ces dernières respectent la convention qui consiste à utiliser quelque part dans leur nom la suite d'exactement deux *underscores* (_ _).

Règles de l'utilisateur Elles sont données à Prolog IV sous forme de texte dans des fichiers ou à la console.

La convention que doit utiliser le programmeur est de **ne pas** utiliser dans le nom des règles qu'il définit une séquence de deux underscores consécutifs (_ _). (il peut utiliser des suites de un, trois, quatre,. . . underscores mais pas deux !)

Règles dynamiques

Lorsqu'on utilise un paquet de règles dynamiques, on a la possibilité de le modifier pendant l'exécution d'un programme, que ce soit par l'ajout ou la suppression de règles. L'entrée des règles dynamiques peut se faire par le biais des primitives de type `assert` ou par l'utilisation des primitives de la famille `consult`.

dynamic(PI**)** le paquet peut être déclaré comme dynamique au moyen de cette primitive. On donne à celle-ci en argument un indicateur de prédicat, qui est de la forme *nom/arité* en mode `iso`. Si l'on est en mode `prolog4`, il faut utiliser la forme *^/(nom,arité)* qui construit exactement le même arbre[1].

Note : Il n'est pas utile de déclarer comme dynamiques les paquets de règles insérés par les primitives de la famille `consult` : ceci est fait implicitement.

```
>> dynamic(^/(toto,2)).
true.
```

asserta(T**), assertz(**T**)** permettent d'ajouter des règles :

```
>> asserta( (toto(1,X) :- write(X), nl) ).
X ~ tree.
>> assertz( (toto(2,X) :- write(X), write(X)) ).
Y ~ tree,
X ~ tree.
```

Notes :

- `assert/1` est un synonyme de `assertz/1`.
- Il faut parenthéser tout terme de priorité supérieure à 1000 donné en argument : il y a sinon une erreur de syntaxe. Les opérateurs «`:-`», «`,`» et «`;`» sont de priorité supérieure à 1000.

clause(T**,**Q**)** permet de retrouver sous forme de terme une règle dont la tête s'unifie avec T. Le nom et l'arité doivent être implicitement donnés par T. Le terme Q représentent la queue de la règle. Cette primitive donne les règles du paquet par backtracking :

```
>> clause(toto(1,X), Queue).
Queue = (write(X),nl),
X ~ tree.
```

retract(T**)** permet de supprimer les règles dont la description sous forme de terme s'unifie avec le patron donné en argument. Cette primitive donne les règles du paquet par backtracking :

```
>> retract( (toto(2,X) :- Queue) ).
Queue = (write(X),write(X)),
X ~ tree.
```

Si des règles à supprimer (ou à ajouter) sont en cours d'utilisation, leur suppression (insertion) du paquet est différée, mais elles deviennent immédiatement inaccessibles pour de nouvelles invocations du paquet.

1. Le caractère d'échappement «`^`» permet de donner un identificateur sans que celui-ci puisse être interprété comme un nom de relation réservée. Ici la relation de nom «`/`» est réservée, c'est la division dans le solveur linéaire.

La vue logique est implantée (tout paquet entamé restera du point de vue de l'appel inchangé pour toute la durée de l'appel, backtracking compris). Tout ce passe comme si une copie de l'état courant du paquet de règles était effectuée à chaque appel et conservée jusqu'à épuisement des choix pour cet appel.

5.2.4 A propos du paramétrage

Un paramètre[2] est un atome auquel est associé une valeur. Chacun des paramètres a un domaine de valeurs possibles prédéfini. Certains paramètres sont modifiables, d'autres sont au contraire fixés pour toute la session Prolog IV. Entre parenthèses se trouvent les valeurs possibles du paramètre. La première d'entre-elles est la valeur par défaut.

Liste des paramètres

bounded : (`false`) indique que l'arithmétique des entiers de Prolog IV travaille avec des entiers en précision infinie. Non modifiable.

integer_rounding_function : (`toward_zero`) indique le fonctionnement de la division entière (`//2`) et du reste (`rem/2`) dans les primitives `is/2` et assimilées. Non modifiable.

char_conversion : (`off on`) non-implanté. Non-modifiable.

debug : (`off on`) indique le mode courant de fonctionnement vis à vis du débogueur. Modifiable.

max_arity : (`1000`) indique l'arité maximale autorisée pour la construction des termes (nombre maximal de fils). Non-modifiable.

unknown : (`error fail warning`) indique l'action que doit effectuer la machine Prolog IV lorsqu'on tente d'exécuter une procédure qui ne fait pas partie de la base de règles. Modifiable[3].

double_quotes : (`atom chars codes`) indique par quoi est traduite l'entité syntaxique « chaîne à guillemet », lors de la lecture d'un terme par `read_term` ou la lecture d'un programme. Modifiable.

interval_mode : (`simple union`) indique le mode d'approximation utilisé par le solveur sur les réels. Il est déconseillé de modifier ce paramètre pendant l'exécution d'un programme qui est en train de manipuler des contraintes sur les réels. Modifiable.

'Q-calculator' : (`on off`) indique si la calculatrice rationnelle est active ou pas. Cette calculatrice est la partie numérique d'un solveur plus général. Elle complète la résolution des contraintes numériques dans les cas où «suffisamment» d'arguments sont connus, en effectuant des calculs en précision infinie pour déterminer la valeur des arguments manquants. Modifiable.

Primitives de gestion de paramétrage

set_prolog_flag(P,V) Affecte la valeur V au paramètre P. La liste des paramètres est donnée plus haut. Il est indiqué pour chacun s'il est possible de le modifier et l'ensemble des valeurs qu'il peut prendre.

2. *flag* dans la norme ISO.

3. Il n'est pas possible de le modifier dans cette version de Prolog IV.

current_prolog_flag(P,V) Permet de retrouver la valeur courante des paramètres qui figurent dans la liste donnée plus haut. Si P est inconnu à l'appel, égraine par backtracking les différents couples paramètre-valeur.

5.2.5 A propos des expressions arithmétiques

La norme ISO du langage Prolog reconnaît certains termes comme représentant des expressions arithmétiques. Ces termes peuvent se trouver soit en second argument du prédicat `is/2`, soit comme arguments des prédicats de comparaison arithmétiques (=:=/2, =\=/2, </2, >/2, =</2, >=/2).

Les nombres de la norme ISO sont subdivisés en deux sous-ensembles : les entiers relatifs en précision parfaite (I) et les nombres flottants IEEE (F) en double précision.

Les nombres rationnels (en précision parfaite eux aussi), peuvent être utilisés à peu près dans n'importe quelle fonction évaluable.

La liste des foncteurs arithmétiques ISO reconnus par Prolog IV peut être trouvée dans la table 5.1.

L'évaluation du ou des expressions arithmétiques doit avoir pour résultat un nombre ; dans le cas contraire, il y a une erreur.

instantiation_error Une variable non instanciée figure dans l'expression à évaluer.

type_error(evaluable, A) Un atome figure dans l'expression à évaluer.

type_error(evaluable, F/N) Le foncteur F d'arité N qui n'est pas une un foncteur arithmétique figure dans l'expression à évaluer.

type_error(number, V) Dans l'expression à évaluer, il y a une sous-expression de la forme `atan(X) cos(X)`, `exp(X)`, `log(X)`, `sin(X)` ou `sqrt(X)` dont l'argument `X` n'est pas une variable mais dont le résultat de l'évaluation n'est pas un nombre.

evaluation_error(float_overflow) La valeur produite par l'évaluation d'une sous-expression est trop grande pour être représentée en double précision.

evaluation_error(underflow) La valeur produite par l'évaluation d'une sous-expression est trop petite pour être représentée en double précision.

evaluation_error(zero_divisor) Tentative de division par zéro lors de l'évaluation d'une sous-expression.

evaluation_error(undefined) La valeur d'une sous-expression est non définie.

system_error Il n'y a plus les ressources nécessaires pour effectuer l'évalution.

Opérations sur les entiers

$$
\begin{array}{lcl}
add_I & I \times I & \longrightarrow I \\
sub_I & I \times I & \longrightarrow I \\
mul_I & I \times I & \longrightarrow I \\
intdiv_I & I \times I & \longrightarrow I \cup \{\texttt{zero\_divisor}\} \\
rem_I & I \times I & \longrightarrow I \cup \{\texttt{zero\_divisor}\} \\
mod_I & I \times I & \longrightarrow I \cup \{\texttt{zero\_divisor}, \texttt{undefined}\} \\
neg_I & I \times I & \longrightarrow I \\
abs_I & I \times I & \longrightarrow I \\
sign_I & I \times I & \longrightarrow I
\end{array}
$$

Les opérations add_I, sub_I, mul_I et neg_I ont leur sens habituel et on a les définitions suivantes :

$$
\begin{array}{lcll}
intdiv_I(x,y) & = & rnd_I(x/y) & \text{si } y \neq 0 \\
 & = & \texttt{zero\_divisor} & \text{si } y = 0 \\
 & & & \\
rem_I(x,y) & = & x - (rnd_I(x/y) * y) & \text{si } y \neq 0 \\
 & = & \texttt{zero\_divisor} & \text{si } y = 0 \\
 & & & \\
mod_I(x,y) & = & x - (rnd_I(\lfloor x/y \rfloor) * y) & \text{si } y \neq 0 \\
 & = & \texttt{zero\_divisor} & \text{si } y = 0
\end{array}
$$

$$
\begin{array}{ll}
sign_I(x) & = 1 \quad \text{si } x \geq 0 \\
 & = 0 \quad \text{si } x < 0
\end{array}
$$

Prolog IV utilise la définition de la partie entière suivante (« round toward zero ») :

$$|rnd_I(x)| \leq |x|$$

Opérations sur les flottants

$$
\begin{array}{lcl}
add_F & F \times F & \longrightarrow F \cup \{\texttt{overflow}, \texttt{underflow}\} \\
sub_F & F \times F & \longrightarrow F \cup \{\texttt{overflow}, \texttt{underflow}\} \\
mul_F & F \times F & \longrightarrow F \cup \{\texttt{overflow}, \texttt{underflow}\} \\
div_F & F \times F & \longrightarrow F \cup \{\texttt{overflow}, \texttt{underflow}, \texttt{zero\_divisor}\} \\
neg_F & F \times F & \longrightarrow F \\
abs_F & F \times F & \longrightarrow F \\
sqrt_F & F \times F & \longrightarrow F \cup \{\texttt{undefined}\} \\
sign_F & F \times F & \longrightarrow F \\
intpart_F & F \times F & \longrightarrow F \\
fractpart_F & F \times F & \longrightarrow F
\end{array}
$$

Les opérations add_F, sub_F, mul_F, div_F, neg_F, abs_F et $sqrt_F$ ont leur sens habituel et on a les définitions suivantes pour le reste :

$$
\begin{array}{ll}
sign_F(x) & = 1 \quad \text{si } x \geq 0 \\
 & = 0 \quad \text{si } x < 0
\end{array}
$$

$$
\begin{array}{lcl}
intpart_F(x,y) & = & sign_F(x) * \lfloor |x| \rfloor \\
 & & \\
fractpart_F(x,y) & = & x - intpart_F(x)
\end{array}
$$

Opérations mixtes

Ces opérations convertissent le(s) entier(s) en nombre(s) flottant(s) et effectuent l'opération correspondante sur les nombres flottants.

$$
\begin{array}{lcl}
add_{FI}(x,y) & = & add_F(x, float_{I\to F}(y)) \\
add_{IF}(x,y) & = & add_F(float_{I\to F}(x), y) \\
sub_{FI}(x,y) & = & sub_F(x, float_{I\to F}(y)) \\
sub_{IF}(x,y) & = & sub_F(float_{I\to F}(x), y) \\
mul_{FI}(x,y) & = & mul_F(x, float_{I\to F}(y)) \\
mul_{IF}(x,y) & = & mul_F(float_{I\to F}(x), y) \\
div_{FI}(x,y) & = & div_F(x, float_{I\to F}(y)) \\
div_{IF}(x,y) & = & div_F(float_{I\to F}(x), y) \\
div_{II}(x,y) & = & div_F(float_{I\to F}(x), float_{I\to F}(y)) \\
sqrt_I(x) & = & sqrt_F(float_{I\to F}(x)) \\
exponent_I(x) & = & exponent_F(float_{I\to F}(x))
\end{array}
$$

Autres opérations

$$
\begin{array}{lcll}
floor_{T\to I}(x) & = & \lfloor x \rfloor & \\
truncate_{T\to I}(x) & = & \lfloor x \rfloor & \text{si } x \geq 0 \\
 & = & -\lfloor |x| \rfloor & \text{si } x < 0 \\
round_{T\to I}(x) & = & \lfloor x + 1/2 \rfloor & \\
ceiling_{T\to I}(x) & = & -\lfloor -x \rfloor &
\end{array}
$$

Opérations trigonométriques Les opérations $sin(x)$, $cos(x)$ et $atan(x)$ ont pour argument un nombre quelconque ($I \cup F$ en radian) et donnent comme résultat un nombre flottant (F).

Puissance, exponentielle et logarithme Les opérations `'**'`(x), $exp(x)$ et $log(x)$ ont pour argument un nombre quelconque ($I \cup F$) et donnent comme résultat un nombre flottant (F).

Exemples

```
?- X is 7 + 3.
X = 10.
?- X is 7777777777777777 * 3333333333333333333.
X = 25925925925925923330740740740740741.
?- X is 7 + 3.0.
X = 1.0e+01.
?- X is 7.0 + 3.0.
X = 1.0e+01.
?- X is 7.0 + 3.0 + 5.
X = 15.0.
?- X is 7.0 + 3.0 + Y.
error: error(instantiation_error,(is)/2)
?- X is a +2.
error: error(type_error(number,a),(is)/2)
```

```
?- X is -7.
X = -7.
?- X is 3 - 7
X = -4.
?- X is 3.0 - 7.
X = -4.0.
?- X is - a.
error: error(type_error(number,a),(is)/2)
```

```
?- X is 3 / 4.
X = 0.75.
?- X is 3 // 4.
X = 0.
?- X is 3 / 0.
error: error(evaluation_error(zero_division,0),(is)/2)
?- X is 3 // 0.
error: error(evaluation_error(zero_division,0),(is)/2)
?- X is mod(7, -2).
X = -1.
```

```
?- X is floor(7.2).
X = 7.
?- X is floor(-7.2).
X = -8.
?- X is round(7.2).
X = 7.
?- X is round(-7.2).
X = -7.
?- X is ceiling(7.2).
X = 8.
?- X is ceiling(-7.2).
X = -7.
?- X is truncate(7.2).
X = 7.
?- X is truncate(-7.2).
X = -7.
?- X is float(-7.2).
X = -7.2.
?- X is float(11111111111111111111111).
X = 1.1111111111111e+23.
```

```
?- X is abs(-7).
X = 7.
?- X is abs(-7.2).
X = 7.2.
?- X is abs(N).
error: error(instantiation_error,(is)/2)
```

```
?- X is sqrt(4.0).
X = 2.
?- X is sqrt(4).
X = 2.
?- X is sqrt(-4).
false.
?- X is sqrt(a).
false.
```

```
?- X is sin(3.1416).
X = -7.346410206643591e-06.
?- X is sin(3.1416/2).
X = 0.9999999999932537.
```

```
?- X is 2 ** 3.
X = 8.0.
?- is(PI, atan(1.0) * 4),  is(X, sin(PI / 2.0)).
PI = 3.141592653589793, X = 1.0
```

```
?- is(X, cos(0.0)).
X = 1.0
?- is(X, cos(N)).
error: error(instantiation_error,(is)/2)
?- is(X, cos(0)).
X = 1.0
?- is(X, cos(foo)).
error: error(type_error(number,foo),(is)/2)
```

```
?- is(PI, atan(1.0) * 4), is(X, cos(PI / 2.0)).
X = 6.123031769111886e-17,
PI = 3.141592653589793.
?- is(X, atan(0.0)).
X = 0.0
?- is(PI, atan(1.0) * 4).
PI = 3.141592653589793
?- is(X, atan(N)).
error: error(instantiation_error,(is)/2)
?- is(X, atan(0)).
X = 0.0
?- is(X, atan(foo)).
error: error(type_error(number,foo),(is)/2)
```

```
?- is(X, exp(0.0)).
X = 1.0
?- is(X, exp(1.0)).
X = 2.718281828459045
?- is(X, exp(N)).
error: error(instantiation_error,(is)/2)
?- is(X, exp(0)).
X = 1.0
```

```
?- is(X, log(1.0)).
X = 0.0
?- is(X, log(2.7818)).
X = 1.0230982001908928
?- is(X, log(0.0)).
error: error(evaluation_error(undefined,0.0),(is)/2)
```

```
?- is(X, sign(-2387477234.239847239874)).
X = 0
?- is(X, sign(0)).
X = 1
?- is(X, sign(0.0)).
X = 1
?- is(X, sign(43982578435.398475345)).
X = 1
?- is(X, sign(87923)).
X = 1
?- is(X, sign(-239847823)).
X = 0
```

```
?- is(X, 0.0 ** -1).
error: error(evaluation_error(undefined,0.0),(is)/2)
?- is(X, log(-1)).
error: error(evaluation_error(undefined,0.0),(is)/2)
?- is(X, 2.0e+300 * 1.0e+308).
error: evaluation_error(overflow,2.0e+300)
?- is(X, 2.0e-300 * 1.0e-308).
error: evaluation_error(underflow,2.0e-300)
?- is(X, 1.0e+308 + 1.0e+308).
error: evaluation_error(overflow,1.0e+308)
?- is(X, 1.0e-300 / 1.0e+308).
error: evaluation_error(underflow,1.0e-300)
?- is(X, toto(8324739824.3495)).
error: type_error(evaluable,toto(8324739824.3495))
```

Comparaisons arithmétiques

Outre la simple évaluation des expressions arithmétiques de la norme avec `is/2`, PrologIV dispose des primitives de comparaison arithmétiques suivantes (les arguments sont préalablement évalués) :

Primitive	Description	Opération
`'=:='/2`	égal	eq_I, eq_F, eq_{IF}, eq_{FI}
`'=\\='/2`	différent	neq_I, neq_F, neq_{IF}, neq_{FI}
`'>'/2`	strictement supérieur	$greater_I$, $greater_F$, $greater_{IF}$, $greater_{FI}$
`'>='/2`	supérieur ou égal	geq_I, geq_F, geq_{IF}, geq_{FI}
`'<'/2`	strictement inférieur	$less_I$, $less_F$, $less_{IF}$, $less_{FI}$
`'=<'/2`	inférieur ou égal	leq_I, leq_F, leq_{IF}, leq_{FI}

Description		Opération
$eq_I(x,y)$	=	true $\Longleftrightarrow x = y$
$eq_F(x,y)$	=	true $\Longleftrightarrow x = y$
$eq_{FI}(x,y)$	=	true $\Longleftrightarrow eq_F(x, float_{I\to F}(y))$
$eq_{IF}(x,y)$	=	true $\Longleftrightarrow eq_F(float_{I\to F}(x), y)$
$neq_I(x,y)$	=	true $\Longleftrightarrow x \neq y$
$neq_F(x,y)$	=	true $\Longleftrightarrow x \neq y$
$neq_{IF}(x,y)$	=	true $\Longleftrightarrow neq_F(x, float_{I\to F}(y))$
$neq_{FI}(x,y)$	=	true $\Longleftrightarrow neq_F(float_{I\to F}(x), y)$
$greater_I(x,y)$	=	true $\Longleftrightarrow x > y$
$greater_F(x,y)$	=	true $\Longleftrightarrow x > y$
$greater_{IF}(x,y)$	=	true $\Longleftrightarrow greater_F(x, float_{I\to F}(y))$
$greater_{FI}(x,y)$	=	true $\Longleftrightarrow greater_F(float_{I\to F}(x), y)$
$geq_I(x,y)$	=	true $\Longleftrightarrow x >= y$
$geq_F(x,y)$	=	true $\Longleftrightarrow x >= y$
$geq_{IF}(x,y)$	=	true $\Longleftrightarrow geq_F(x, float_{I\to F}(y))$
$geq_{FI}(x,y)$	=	true $\Longleftrightarrow geq_F(float_{I\to F}(x), y)$
$less_I(x,y)$	=	true $\Longleftrightarrow x < y$
$less_F(x,y)$	=	true $\Longleftrightarrow x < y$
$less_{IF}(x,y)$	=	true $\Longleftrightarrow less_F(x, float_{I\to F}(y))$
$less_{FI}(x,y)$	=	true $\Longleftrightarrow less_F(float_{I\to F}(x), y)$
$leq_I(x,y)$	=	true $\Longleftrightarrow x \leq y$
$leq_F(x,y)$	=	true $\Longleftrightarrow x \leq y$
$leq_{IF}(x,y)$	=	true $\Longleftrightarrow leq_F(x, float_{I\to F}(y))$
$leq_{FI}(x,y)$	=	true $\Longleftrightarrow leq_F(float_{I\to F}(x), y)$

5.3 Liste alphabétique

!/0 — Cut

Types : !

Description : `!` s'exécute en coupant tous les choix possibles le précédant dans la règle où il est présent. Ces choix sont : les autres têtes de règles du paquet, et les autres manières d'exécuter les appels précédant le cut dans cette règle.

Note : `!/0` réussit toujours.

Exemples :

```
?- ami(X).
X = pierre;
X = paul;
X = marie.
?- ami(X), !.
X = pierre.
?- consult.
Consulting ...
echec(R) :- call(R), !, fail.
echec(R).
end_of_file.
true.
?- echec(ami(X)).
false.
```

=/2 — Egalité

Types : '='(?terme,?terme),?terme =?terme

Description : `'='(T1,T2)` ajoute la contrainte `T1 = T2` au système de contraintes courant.

Note : = est un opérateur infixé prédéfini de priorité 700 et non associatif (`xfx`).

Exemples :

```
?- a = a.
true.
?- 120 = "Cent vingt".
false.
?- f(X) = f(1).
X = 1.
?- _ = _.
true.
?- f(_) = g(_).
false.
?- X = [1,2,3,X].
X = [1,2,3,X].
?- X = Y, X = abc.
Y = abc,
X = abc.
?- X = Y.
X = Y,
Y ~ tree.
?- f(X,Y) = f(Z,Z), X=2, Y=3.
false.
```

Voir également : ==/2, \=/2, dif/2, eq/2, is/2.

==/2 — Termes identiques

Types : ==(@terme, @terme)

Description : L'exécution de $T_1 == T_2$ réussit si et seulement si les termes complètement instanciés représentés par T_1 et T_2 sont identiques. On appelle aussi cette comparaison *formellement égal*.

Exemples :

```
?- f(a, [1, 2, 3], b) == f(a, [1, 2, 3], b).
true.
?- f(a, [1, 2, 3], X) == f(a, [1, 2, 3], X).
X ~ tree.
?- X == X.
X ~ tree.
?- X == Y.
false.
?- X = Y, X == Y.
X = Y,
Y ~ tree.
?- 1 == 1.0.
false.
?- prolog4.
true.
>> 1 == 1.0.
true.
>> 1.5 == 3 / 2.
true.
>> iso.
true.
```

Note : == est un opérateur infixé prédéfini.

Voir également : \==/2, @</2, @=</2, @>/2, @>=/2.

=../2 — Eclatement de terme

Types : =..(+nonvar,?liste)
=..(-nonvar, +liste)

Description : L'exécution de T =.. L réussit si et seulement si :

- T est un terme atomique (nombre ou atome) et L est une liste dont le seul élément est T, ou bien si
- T est un terme composé et L est une liste dont la tête est le nom du foncteur de T et dont la queue est la liste des arguments de T.

Erreurs :

1. T est une variable et L une liste partielle (qui a la structure d'une liste, mais sans `[]` à la fin).
2. T est une variable et L n'est ni une liste ni une liste partielle.

3. T est une variable et L est une liste dont la tête est une variable.
4. L est une liste dont la tête n'est ni un atome ni une variable, et dont la queue n'est pas la liste vide.
5. L est une liste dont la tête est un terme composé, et dont la queue est la liste vide.
6. T n'est pas une variable et L n'est ni une variable ni une liste.
7. T est une variable et L est la liste vide.
8. T est une variable et la queue de L a une longueur plus grande que le paramètre d'implantation `max_arity`.

Exemples :

```
?- =..(foo(a,b), [foo,a,b]).
true.
?- =..(X, [foo, a,b]).
X = foo(a,b).
?- foo(a,b) =.. L.
L = [foo,a,b].
?- =..(foo(X,b), foo(a,Y)).
Y = b,
X = a.
?- =..(1, [1]).
true.
?- =..(foo(a,b), [foo,b,a]).
fail.
?- =..(X,Y).
error: error(instantiation_error,(=..)/2)
?- =..(X, [foo, a|Y]).
error: error(instantiation_error,(=..)/2)
?- =..(X, 4).
error: error(type_error(list,4),(=..)/2)
?- =..(X, [3,1]).
error: error(type_error(atom,3),(=..)/2)
```

Voir également : functor/3, arg/3.

=:=/2, =\=/2, >/2, >=/2, </2, =</2 — Comparaison arithmétique

Types :
'=:='(@évaluable, @évaluable), @évaluable =:= @évaluable
'=\\='(@évaluable, @évaluable), @évaluable =\= @évaluable
'>='(@évaluable, @évaluable), @évaluable >= @évaluable
'>='(@évaluable, @évaluable), @évaluable >= @évaluable
'=<'(@évaluable, @évaluable), @évaluable =< @évaluable
'>='(@évaluable, @évaluable), @évaluable >= @évaluable

Description : Après évaluation de `T1` donnant `V1` et celle de `T2` donnant `V2`, on a les fonctionnalités :
`'=:='(T1,T2)` réussit ssi `V1` égale `V2`.
`'=\\='(T1,T2)` réussit ssi `V1` diffère de `V2`.
`'=<'(T1,T2)` réussit ssi `V1` est inférieure ou égale à `V2`.
`'<'(T1,T2)` réussit ssi `V1` est inférieure à `V2`.

`'>='(T1,T2)` réussit ssi `V1` est supérieure ou égale à `V2`.
`'>'(T1,T2)` réussit ssi `V1` est supérieure à `V2`.

Note : Tous ces opérateurs infixés sont prédéfinis, de priorité 700 et non associatif (`xfx`).

Exemples :

```
?- 1 =:= 2.
false.
?- 1 =:= 1.
true.
?- 1 =:= 1.0.
true.
?- 3 * 2.0 =:= 12.0 / 2.
true.
?- exp(1-2) =< exp(1).
true.
?- X > 3.
error: error(instantiation_error,is/2)
```

Voir également : is/2 .

@</2 — Terme inférieur

Types : @<(@terme, @terme)

Description : L'exécution de T_1 @< T_2 réussit si et seulement si le terme complètement instancié représenté par T_1 précède, dans l'ordre sur les termes, le terme complètement instancié représenté par T_2.

Exemples :

```
?- f(1, 2, 3) @< f(1, 2, 3).
false.
?- X @< 1.5.
X ~ tree.
?- _ @< 1.
true.
?- 2.5 @< 1.
true.
?- 1 @< deux.
true.
?- simple @< f(1, 2, 3).
true.
?- X = X, Y = Y, X @< Y.
Y ~ tree,
X ~ tree.
?- Y = Y, X = X, X @< Y.
false.
?- _ @< _.
true.
?- 1 @< 2.
true.
?- 1.5 @< 1.6.
true.
?- abczef @< abcdef.
false.
?- abc @< abcde.
true.
?- abc(1, 2, 3) @< abcde(1, 2, 3).
true.
?- abc(1, 2, 3) @< abcde(1, 2).
false.
?- f(2, 3, 5) @< f(2, 7, 4).
true.
```

Note : @< est un opérateur infixé prédéfini.

Voir également : ==/2, \==/2, @=</2, @>/2, @>=/2.

@=</2 — Terme inférieur ou égal

Types : @=<(@terme, @terme)

Description : L'exécution de T_1 @=< T_2 réussit si et seulement si les termes complètement instanciés représentés par T_1 et T_2 sont identiques ou bien si le terme complètement instancié représenté par T_1 précède, dans l'ordre sur les termes, le terme complètement instancié représenté par T_2.

Exemples :

```
?- f(1, 2, 3) @=< f(1, 2, 3).
true.
?- f(1, 2, 3) @=< f(1, 4, 3).
true.
?- X = Y, X @=< Y.
X = Y,
Y ~ tree.
```

Note : @=< est un opérateur infixé prédéfini.

Voir également : ==/2, \==/2, @</2, @>/2, @>=/2.

@>/2 — Terme supérieur

Types : @>(@terme, @terme)

Description : L'exécution de T_1 @< T_2 réussit si et seulement si le terme complètement instancié représenté par T_1 précède, dans l'ordre sur les termes, le terme complètement instancié représenté par T_2.

Exemples :

```
?- f(1, 2, 3, 4) @> g(1, 2, 3).
true.
?- X = Y, X @> Y.
false.
```

Note : @> est un opérateur infixé prédéfini.

Voir également : ==/2, \==/2, @</2, @=</2, @>=/2.

@>=/2 — Terme supérieur ou égal

Types : @>=(@terme, @terme)

Description : L'exécution de T_1 @=< T_2 réussit si et seulement si les termes complètement instanciés représentés par T_1 et T_2 sont identiques ou bien si le terme complètement instancié représenté par T_1 précède, dans l'ordre sur les termes, le terme complètement instancié représenté par T_2.

Exemples :

```
?- f(1, 2, 3, 4) @>= g(1, 2, 3).
true.
?- X = Y, X @>= Y.
X = Y,
Y ~ tree.
```

Note : @>= est un opérateur infixé prédéfini.

Voir également : ==/2, \==/2, @</2, @=</2, @>/2.

\+/1 — Non effaçable

Types : `'\\+'(@terme_exécutable)`

Description : `'\\+'(T)` échoue si `call(T)` réussit, et réussit dans le cas contraire. Un *cut* (!) présent dans T reste local à `call` pendant l'exécution (et ne coupe donc aucun des choix de la règle contenant ce `call`.

Erreurs :

1. `T` est une variable ou un terme non-exécutable
   `instantiation_error`

Exemples :

```
?- ami(X).
X = pierre;
X = paul;
X = marie.
?- \+ ami(paul).
false.
?- \+ ami(X).
false.
?- \+ ami(anne).
true.
?- \+ ami(X, Y).
error: undefined_call(ami/2)
?- \+ X.
error: error(instantiation_error,call/1)
?- \+ 1.
error: error(type_error(callable,1),'$Control_construct')
```

Voir également : call/2.

\=/2 — Non-égalité

Types : '\\='(@terme, @terme), @terme \= @terme

Description : `\=(T1,T2)` réussit si la contrainte `T1 = T2` est insoluble. Echoue dans le cas contraire. Ce n'est pas une contrainte, juste une vérification.

Note : Noter la différence avec `dif/2`.

Exemples :

```
?- a \= a.
false.
?- a \= 1.
true.
?- f(X) \= g(Y).
Y ~ tree,
X ~ tree.
?- f(X,Y) \= f(Z,Z), X=2, Y=3.
false.
?- X=2, Y=3, f(X,Y) \= f(Z,Z).
Y = 3,
X = 2,
Z ~ tree.
```

Voir également : =/2, ==/2, eq/2, dif/2.

\==/2 — Termes non identiques

Types : \==(@terme, @terme), @terme \== @terme

Description : L'exécution de T_1 \== T_2 réussit si et seulement si les termes complètement instanciés représentés par T_1 et T_2 ne sont pas identiques.

Exemples :

```
?- f(a, [1, 2, 3], b) \== f(a, [1, 2, 3], b).
false.
?- f(a, [1, 2, 3], X) \== f(a, [1, 2, 3], X).
false.
?- f(a, [1, 2, 3], X) \== f(a, [1, 2, 3], Y).
Y ~ tree,
X ~ tree.
?- X \== Y.
Y ~ tree,
X ~ tree.
?- X = Y, X \== Y.
false.
?- _ \== _.
true.
```

Note : \== est un opérateur infixé prédéfini.

Voir également : ==/2, @</2, @=</2, @>/2, @>=/2.

^ /2 — Variables existentielles

Types : ^(@terme, @terme)

Description : Ce prédicat, qui est aussi un opérateur prédéfini, sert à distinguer certaines variables d'un terme, appelées ses variables existentielles. L'effacement de $X\hat{}T$ est équivalent à celui de T. L'utilisation de $X\hat{}T$ n'a de sens qu'en association avec `bagof`/3 et `setof`/3.

L'ensemble des *variables existentielles* d'un terme T est défini de la manière suivante :

- si T est de la forme $T_1\hat{}T_2$, l'ensemble des variables existentielles de T est la réunion de l'ensemble des variables de T_1 et de l'ensemble des variables existentielles de T_2 ;
- sinon, l'ensemble des variables existentielles de T est vide.

Par exemple, $\{X, Y\}$ est l'ensemble des variables existentielles de chacun des trois termes suivants : $X\hat{}Y\hat{}f(X, Y, Z)$, $(X, Y)\hat{}g(Z, Y, X)$, $(X + Y)\hat{}3$.

Note : Nous appellerons *terme déquantifié* correspondant à un terme T le terme T' défini de la manière suivante :

- si T est de la forme $T_1\hat{}T_2$ alors T' est le terme déquantifié de T_2 ;
- sinon, T' est T.

Voir également : bagof/3, findall/3, setof/3.

,/2 — Conjonction

Types : ','(+terme_exécutable, +terme_exécutable)

Description : `(T1,T2)` exécute `T1`. Si c'est un succès, exécute alors `T2`. C'est le mécanisme principal d'exécution d'une queue de règle.
« , » est un opérateur infixé prédéfini.

Exemples :

```
?- write(a), write(b).
abtrue.
?- write(a), write(b), nl.
ab
true.
```

Voir également : ;/2, -> /2.

;/2 — Disjonction

Types : ';'(+terme_exécutable, +terme_exécutable)

Description : `(T1;T2)` exécute alternativement les deux appels à `T1` et `T2`. On crée donc un point de choix en cet endroit. « ; » est un opérateur infixé prédéfini.

Exemples :

```
?- true; true.
true;
true.
?- write(a) ; write(b) ; write(c) ; write(d) .
atrueb;
truec;
trued;
true.
```

Voir également : ,/2, -> /2.

->/2 — Si-alors

Types : '-> '(+terme_exécutable, +terme_exécutable)

Description : `(T1 -> T2)` exécute `T1` une fois. Si c'est un succès, exécute alors `T2` de toutes les façons possibles. Le symbole `->` est un opérateur infixé prédéfini.

Exemples :

```
?- 1 = 1 -> ami(X).
X = pierre;
X = paul;
X = marie.
?- 1 = 2 -> ami(X).
false.
```

Voir également : ,/2, ;/2.

->;/3 — Si-alors-sinon

Types : +terme_exécutable -> +terme_exécutable ; +terme_exécutable

Description : (`T -> T1 ; T2`) s'exécute de la manière suivante. Exécute `T`. Si c'est un succès, exécute `T1` et ignore `T2`. Si c'est un échec, exécute `T2`. «`;` » est un opérateur infixé prédéfini. « `->` » est un opérateur infixé prédéfini. Un littéral de la forme `(Cond -> Alors ; Sinon` est compris comme étant `';'('->'(Cond, Alors), Sinon)`. Un *cut* (!) dans `Alors` ou `Sinon` coupe les choix de cette règle, comme si le *cut* était au même niveau syntaxique.

Exemples :

```
?- consult.
Consulting ...
test(P) :- ( call(P) -> write("oui") ; write("non") ), nl.
end_of_file.
true.
?- test(1 = 2).
non
true.
?- test(1 = 1).
oui
true.
```

Voir également : ,/2, ;/2, ->/2.

abolish/1 — Suppression d'une procédure

Types : abolish(@indicateur_de_prédicat)

Description : Sauf cas d'erreur, l'effacement de `abolish(`P`)` réussit toujours, avec pour effet la suppression de toutes les clauses dont l'indicateur de prédicat N/A s'unifie avec P.

Contrairement à `retract`/1, `abolish`/1 supprime les clauses et leurs procédures. Après l'effacement de `abolish(`A/N`)` l'état du programme est le même que si la procédure A/N n'avait jamais existé.

Erreurs :

1. L'argument est une variable.
2. L'argument est de la forme N/A, mais N et A sont tous les deux des variables.
3. L'argument n'est pas une variable et n'est pas de la forme N/A.
4. L'argument est de la forme N/A mais A n'est ni une variable ni un entier.
5. L'argument est de la forme N/A mais N n'est ni une variable ni un atome.
6. L'argument est de la forme N/A mais A est un entier négatif.
7. L'argument est de la forme N/A mais A est un entier supérieur à la valeur prédéfinie `max_arity`.

8. L'argument représente le prédicat correspondant à une procédure statique.

Exemples :

```
?- consult.
Consulting ...

pattes(X, 4) :- quadrupede(X).
pattes(pieuvre, 8).
pattes(X, 2) :- bipede(X).
pattes(X, 6) :- insecte(X).

tetes(X, 1).

quadrupede(cheval).
quadrupede(chien).

bipede(autruche).
bipede(canard).

insecte(mouche).

end_of_file.
true.
?- pattes(X, 4).
X = cheval;
X = chien;
X = chat.
?- abolish(quadrupede / 1).
true.
?- pattes(X, 4).
false.
?- abolish(bipede / N).

error: error(instantiation_error,abolish/1)
```

Note : Seules les procédures dynamiques peuvent être supprimées par des exécutions de `abolish`/1.

Voir également : asserta/1, assertz/1, retract/1.

arg/3 — Sélection d'un argument

Types : arg(+entier, +terme_composé,?terme)

Description : Cette primitive calcule le N-ième argument d'un terme. Plus précisément `arg(N,T1,T2)` pose la contrainte $\{$`T2` $=$ `M`$\}$, où `M` est le N-ième fils du terme `T1`. Echoue si `N` est nul ou plus grand que le nombre de fils de `T1`.

Erreurs : `N` est inconnu ou négatif.

Exemples :

```
?- arg(2, f(a,b,c), X).
X = b.
?- arg(0, f(a,b,c), X).
false.
?- arg(5, f(a,b,c), X).
false.
?- arg(2, f(X, Y, Z), g(Y)).
Y = g(Y),
Z ~ tree,
X ~ tree.
?- arg(X, f(a, b, c), b).
error: error(instantiation_error,arg/3)
?- arg(1, [1, 2, 3], X).
X = 1.
?- arg(2, [1, 2, 3], X).
X = [2,3].
?- arg(0, [1, 2, 3], X).
false.
```

Voir également : =../2, functor/3.

asserta/1 — Ajout d'une clause en tête de sa procédure

Types : asserta(@clause)

Description : Sauf cas d'erreur, l'effacement de `asserta(`T`)` réussit toujours, avec pour effet de bord l'ajout au programme courant de la clause déterminée par T. Cet ajout se fait au début de la procédure correspondante, c'est-à-dire devant les autres clauses qui ont le même prédicat.

La clause ajoutée par l'effacement de `asserta(`T`)` est :

- la règle $Tete$:- $Corps$, si T s'unifie avec le terme $Tete$:- $Corps$,
- la clause T :- $true$., c'est-à-dire le fait T, sinon.

Le prédicat (nom et arité) de la clause doit être connu au moment de l'exécution de `asserta(`T`)`. Autrement dit, $Tete$ (dans le premier cas) ou T (dans le second) doit représenter un arbre dont l'étiquette initiale est un atome connu et dont le nombre de fils est connu.

Erreurs :

1. L'argument est une variable.
2. Le terme $Tete$ (dans le cas 1) ou T tout entier (dans le cas 2) ne peut pas être converti en une tête de clause.

3. On se trouve dans le cas 1, et le terme $Corps$ ne peut pas être converti en un corps de clause.

4. Le prédicat de la clause déterminée par T correspond à une procédure statique existante (exemple : un prédicat prédéfini).

Exemples :

```
?- consult.
Consulting ...

pattes(X, 4) :- quadrupede(X).
pattes(pieuvre, 8).
pattes(X, 2) :- bipede(X).

tetes(X, 1).

quadrupede(cheval).
quadrupede(chien).

bipede(autruche).
bipede(canard).

insecte(mouche).

end_of_file.
true.
?- pattes(X, 4).
X = cheval;
X = chien.
?- asserta(quadrupede(chat)).
true.
?- pattes(X, 4).
X = chat;
X = cheval;
X = chien.
?- asserta((pattes(X, 6) :- insecte(X))).
X ~ tree.
?- pattes(X, N), N >= 6.
N = 6,
X = mouche;
N = 8,
X = pieuvre.
```

Notes :

1. La norme ISO du langage Prolog préconise le « point de vue logique » pour les modifications du programme courant. Cela signifie que si l'exécution d'un appel a pour effet d'ajouter ou de soustraire des clauses à la procédure correspondant à cet appel, alors les modifications ne sont effectives que lors des exécutions ultérieures de cette procédure ; les clauses ajoutées ou supprimées n'affectent pas l'exécution en cours.

2. Comme l'indique le dernier des cas d'erreur énumérés plus haut, seules les clauses des procédures dynamiques peuvent être créées par des exécutions de `asserta`/2.

3. La primitive `dynamic/1` permet de déclarer des procédures comme étant dynamiques (par exemple `dynamic(pattes/2)`). Les primitives `consult` le font automatiquement.

Voir également : assertz/1, retract/1, abolish/1, dynamic/1.

assertz/1 — Ajout d'une clause à la fin de sa procédure

Types : assertz(@clause)

Description : Sauf cas d'erreur, l'effacement de `assertz(`T`)` réussit toujours, avec pour effet de bord l'ajout au programme courant de la clause déterminée par T. Cet ajout se fait à la fin de la procédure correspondante, c'est-à-dire derrière les autres clauses qui ont le même prédicat.

La clause ajoutée par l'effacement de `assertz(`T`)` est :

- la règle $Tete$:- $Corps$, si T s'unifie avec le terme $Tete$:- $Corps$,
- la clause T :- $true$., c'est-à-dire le fait T, sinon.

Le prédicat (nom et arité) de la clause doit être connu au moment de l'exécution de `assertz(`T`)`. Autrement dit, $Tete$ (dans le premier cas) ou T (dans le second) doit représenter un arbre dont l'étiquette initiale est un atome connu et dont le nombre de fils est connu.

Erreurs :

1. L'argument est une variable.
2. Le terme $Tete$ (dans le cas 1) ou T tout entier (dans le cas 2) ne peut pas être converti en une tête de clause.
3. On se trouve dans le cas 1, et le terme $Corps$ ne peut pas être converti en un corps de clause.
4. Le prédicat de la clause déterminée par T correspond à une procédure statique existante (exemple : un prédicat prédéfini).

Exemples :

```
?- consult.
Consulting ...

pattes(X, 4) :- quadrupede(X).
pattes(pieuvre, 8).
pattes(X, 2) :- bipede(X).

tetes(X, 1).

quadrupede(cheval).
quadrupede(chien).

bipede(autruche).
bipede(canard).

insecte(mouche).

end_of_file.
true.
```

```
?- pattes(X, 4).
X = cheval;
X = chien.
?- assertz(quadrupede(chat)).
true.
?- pattes(X, 4).
X = cheval;
X = chien;
X = chat.
?- assertz((pattes(X, 6) :- insecte(X))).
X ~ tree.
?- pattes(X, N), N >= 6.
N = 8,
X = pieuvre;
N = 6,
X = mouche.
```

Voir également : asserta/1, retract/1, abolish/1.

atom/1 — Test d'identificateur

Types : atom(@terme)

Description : `atom(T)` reussit si `T` est un atome (identificateur) connu, échoue sinon.

Exemples :

```
?- atom(abc).
true.
?- atom(X).
false.
?- X=abc, atom(X).
X = abc.
?- atom(X), X = abc.
false.
?- atom(123).
false.
?- atom('123').
true.
?- atom([]).
true.
```

Voir également : atomic/1

atomic/1 — Test de feuille

Types : atomic(@terme)

Description : `atomic(A)` réussit si le terme `A` représente un arbre sans fils, c.à.d. un atome ou un nombre connu, échoue sinon.

Exemples :

```
?- atomic(abc).
true.
?- atomic(123).
true.
?- atomic([]).
true.
?- atomic(X).
false.
?- atomic(f(1,2,3)).
false.
?- atomic(2 + 3).
false.
?- prolog4.
true.
>> atomic(2 + 3).
true.
>> iso.
true.
?-
```

Voir également : atom/1, compound/1, number/1.

atom_chars/2 — Eclatement d'atome

Types : atom_chars(+atome,?liste_de_caractères)
atom_chars(-atome, +liste_de_caractères)

Description : `atom_chars(A,L)` réussit si `L` est la liste dont les éléments sont les caractères correspondants aux caractères successifs formant l'atome `A`.

Erreurs :

1. Si `A` est une variable et `L` est une liste partielle ou une liste dont un élément au moins est une variable,
2. Si `A` n'est ni une variable ni un atome,
3. Si `A` est une variable et `L` n'est ni une liste ni une une liste partielle,
4. Si `A` est une variable et il existe un élément `E` de `L` qui n'est ni un caractère ni une variable.

Exemples :

```
?- atom_chars(bonjour, X).
X = [b,o,n,j,o,u,r].
?- atom_chars(abcde, [a, b, X, d, Y]).
Y = e,
X = c.
?- atom_chars(X, [a, b, c, d, e]).
X = abcde.
?- atom_chars('1 + x = 5', X).
X = ['1',' ',+,' ',x,' ',=,' ','5'].
?- atom_chars(X, [a, ' ', +, ' ', c]).
X = 'a + c'.
?- atom_chars( », X).
X = [].
```

Voir également : atom_concat/3, atom_length/2, sub_atom/5.

atom_codes/2 — Eclatement d'atome

Types : atom_codes(+atome,?liste_de_codes_caractères)
atom_codes(-atome, +liste_de_codes_caractères)

Description : `atom_codes(A,L)` réussit si `L` est la liste dont les éléments sont les codes des caractères correspondants aux caractères successifs formant l'atome `A`.

Erreurs :

1. Si `A` est une variable et `L` est une liste partielle ou une liste dont un élément au moins est une variable,
2. Si `A` n'est ni une variable ni un atome,
3. Si `A` est une variable et `L` n'est ni une liste ni une une liste partielle,
4. Si `A` est une variable et il existe un élément `E` de `L` qui n'est ni un code de caractère ni une variable.

Exemples :

```
?- atom_codes(abcde, X).
X = [97,98,99,100,101].
?- atom_codes(X, [65, 66, 67, 68, 69]).
X = 'ABCDE'.
```

Voir également : atom_concat/3, atom_length/2, sub_atom/5.

atom_concat/3 — Concaténation d'atomes

Types : atom_concat(?atome,?atome, +atome)
atom_concat(+atome, +atome, -atome)

Description : `atom_concat(A1,A2,A12)` réussit si l'atome `A12` est formé de la concaténation des caractères des atomes `A1` et `A2`. Se comporte principalement comme le classique `append/3`. Notamment, si ni `A1` ni `A2` ne sont connus, essaie de manière non-déterministe toutes les affectations possibles de `A1` et `A2` qui rendent le prédicat vrai.

Erreurs :

1. Si `A1` et `A12` sont des variables,
2. Si `A2` et `A12` sont des variables,
3. Si `A1` n'est ni une variable ni un atome,
4. Si `A2` n'est ni une variable ni un atome,
5. Si `A12` n'est ni une variable ni un atome,

Exemples :

```
?- atom_concat(abc, def, Z).
Z = abcdef.
?- atom_concat(abc, Y, abcdef).
Z = def.
?- atom_concat(X, def, abcdef).
Z = abc.
?- atom_concat(X, Y, abcd).
Y = abcd,
X = '';
Y = bcd,
X = a;
Y = cd,
X = ab;
Y = d,
X = abc;
Y = '',
X = abcd.
```

Voir également : atom_length/2.

atom_length/2 — Nombre de caractères d'un atome

Types : atom_length(+atome,?entier)

Description : `atom_length(A,I)` réussit si le nombre de caractères de l'atome `A` est égal à `I`. `A` doit être un atome connu au moment de l'appel. Si `I` n'est pas un nombre connu, la valeur de `I` est rendue égale au nombre de caractères de `A`.

Erreurs :

1. Si `A` est une variable,
2. Si `A` n'est ni une variable ni un atome,

3. Si `I` n'est pas une variable ni un entier,
4. Si `I` est un entier strictement négatif.

Exemples :

```
?- atom_length(abcde, X).
X = 5.
?- atom_length('', X).
X = 0.
?- atom_length(X, 0).
error: error(instantiation_error,atom_length/2)
```

at_end_of_stream/1
at_end_of_stream/0 — Test de la fin d'un flux

Types : at_end_of_stream(@flux_ou_alias)
at_end_of_stream

Description : L'exécution de `at_end_of_stream(`F`)` [resp. `at_end_of_stream`] réussit si et seulement si la fin du flux a été atteinte sur le flux indiqué [resp. le flux d'entrée courant].

Erreurs :
1. F est une variable.
2. F n'est ni une variable, ni un descripteur de flux, ni un alias.
3. F n'est pas associé à un flux ouvert.

Exemples : Soit le fichier `data.in` contenant les trois lignes suivantes :

```
111. 222. 333.
444. 555. 666.
777.
```

Calculons la somme des nombres qui le constituent :

```
?- consult.
Consulting ...

somme(NomFichier, Total) :-
    open(NomFichier, read, In, []),
    current_input(PrevIn),
    set_input(In),
    read(N),
    suiteSomme(N, 0, Total),
    set_input(PrevIn).

suiteSomme(N, S, S) :- at_end_of_stream, !.
suiteSomme(N0, S0, S2) :-
    S1 is S0 + N0, read(N1), suiteSomme(N1, S1, S2).

end_of_file.
true.
?- somme("data.in", T).
T = 3108.
```

Fait autrement :

```
?- consult.
Consulting ...

somme(NomFichier, Total) :-
    open(NomFichier, read, In, []),
    read(In, N),
    suiteSomme(In, N, 0, Total).

suiteSomme(In, N, S, S) :- at_end_of_stream(In), !.
suiteSomme(In, N0, S0, S2) :-
    S1 is S0 + N0, read(In, N1), suiteSomme(In, N1, S1, S2).

end_of_file.
true.
?- somme("data.in", T).
T = 3108.
```

Voir également : stream_property/2.

bagof/3 — Liste de solutions

Types : bagof(@terme, +terme_exécutable,?liste)

Description : Sauf cas d'erreur, l'exécution de `bagof` ($Terme$, But, $Liste$) unifie $Liste$ avec la liste L construite de la manière suivante :

S étant le système de contraintes courantes, soit I une affectation des variables non existentielles de But n'apparaissant pas dans $Terme$, telle que But s'efface sous les contraintes $S \cup I$

Dans ces conditions, X étant une variable qui n'apparaît ni dans $Terme$ ni dans But, L est la liste des valeurs successivement prises par X lorsque, avec le système de contraintes $S \cup I$, on effectue de toutes les manières possibles les appels : $But, X = Terme$.

L'ordre des éléments de la liste L est l'ordre dans lequel sont trouvées les solutions du but ci-dessus.

Cet effacement produit la création d'un point de choix dont les alternatives correspondent aux autres choix possibles pour I (i.e. les autres affectations possibles des variables libres non existentielles de But).

Erreurs :

1. Le terme déquantifié correspondant à But est une variable.
2. Le terme déquantifié correspondant à But n'est ni une variable ni un terme exécutable.
3. $Liste$ n'est ni une variable ni une liste.

Exemples :

```
?- consult.
Consulting ...

pattes(X, 4) :- quadrupede(X).
pattes(pieuvre, 8).
pattes(X, 2) :- bipede(X).
pattes(X, 6) :- insecte(X).

tetes(X, 1).

quadrupede(cheval).
quadrupede(chien).
quadrupede(chat).

bipede(autruche).
bipede(canard).

insecte(mouche).

end_of_file.
true.
?- pattes(X, N).
N = 4,
X = cheval;
N = 4,
X = chien;
N = 4,
X = chat;
N = 8,
X = pieuvre;
N = 2,
X = autruche;
N = 2,
X = canard;
N = 6,
X = mouche.
?- bagof(X, pattes(X, N), L).
L = [mouche],
N = 6,
X ~ tree;
L = [autruche,canard],
N = 2,
X ~ tree;
L = [pieuvre],
N = 8,
X ~ tree;
L = [cheval,chien,chat],
N = 4,
X ~ tree.
?- bagof(X, N ^ pattes(X, N), L).
L = [cheval,chien,chat,pieuvre,autruche,canard,mouche],
N ~ tree,
X ~ tree.
```

Voir également : findall/3, setof/3.

call/1 — Méta-appel

Types : call(+terme_exécutable)

Description : `call(T)` exécute l'appel représenté par le terme `T`.

Notes :

1. Le prédicat `call/1` est réexécutable.
2. L'effet d'un cut figurant à l'intérieur du terme `T` est limité à cet appel. Il n'a pas d'effet à l'extérieur de `call/1`.

Erreurs :

1. `T` est une variable,
   `instantiation_error`
2. `T` n'est ni une variable, ni un terme exécutable,
   `type_error(callable)`

Exemples :

```
?- call( write("Hello") ).
Hellotrue.
?- call( ( write("Hello"), nl ) ).
Hello
true.
?- call(X).
error: error(instantiation_error,call/1)
?- call(123).
error: error(type_error(callable,123),'$Control_construct')
?- call(bonjour).
error: undefined_call(bonjour/0)
?- call(ami(X)).
X = pierre;
X = paul;
X = marie.
```

catch/3 — Gestion d'erreur

Types : catch(+terme_exécutable,?terme,?terme)

Description : `catch(T,T1,T2)` exécute l'appel à `T`. Si une erreur est générée durant l'exécution, la contrainte `T1` = *terme-message-d'erreur* est ajoutée à l'ensemble de contraintes courant, et l'appel `T2` est exécuté.

Erreurs :

1. `T` est une variable,
   `instantiation_error`
2. `T` n'est ni une variable, ni un terme exécutable,
   `type_error(callable)`

Exemples :

```
?- call(X).
error: error(instantiation_error,call/1)
?- catch(call(X), error(X,Y), true).
Y = call/1,
X = instantiation_error.
?- consult.
Consulting ...
job(R) :- catch( call(R), _,
          ( write("l'execution de "), write(f),
            write(" s'est mal passee"), nl, false ) ).
end_of_file.
true.
?- job(X).
l'execution de f s'est mal passee
false.
?- job(machin(1,2)).
l'execution de f s'est mal passee
false.
```

Voir également : throw/1.

char_code/2 — Correspondance caractère/code

Types : char_code(+caractère,?code_caractère)
char_code(-caractère, +code_caractère)

Description : `char_code(C,N)` réussit si `N` est le code correspondant au caractère `C`.

Erreurs :

1. Si `C` et `N` sont des variables,
2. Si `C` n'est ni une variable ni un caractère,
3. Si `N` n'est ni une variable ni un entier,
4. Si `N` n'est ni une variable ni un code de caractère.

Exemples :

```
?- char_code(a, N).
N = 97.
?- char_code(X, 97).
X = a.
?- char_code(X, Y).
error: error(instantiation_error,char_code/2)
?- char_code(X, 1000).
error: error(representation_error(character_code),char_code/2)
```

Voir également : atom_concat/3, atom_length/2, sub_atom/5.

clause/2 — Recherche de clauses

Types : clause(+tête,?terme_exécutable)

Description : Sauf cas d'erreur, `clause(`T`,`C`)` s'efface en unifiant respectivement T et C avec $Tete$ et $Corps$, où $Tete$:- $Corps$. est une des clauses du programme courant pour lesquelles cette unification est possible.

Cet effacement produit la création d'un point de choix, dont les alternatives correspondent aux autres clauses $Tete$:- $Corps$ ayant la même propriété. En Prolog IV, ces clauses sont prises en considération dans l'ordre où elles figurent dans le programme.

S'il n'existe aucune telle clause, l'effacement de `clause(`T`,`C`)` échoue.

Erreurs :

1. Le premier argument est une variable.
2. Le premier argument ne peut pas être converti en une tête de clause.
3. Le premier argument détermine une clause appartenant à une procédure statique.
4. Le deuxième argument n'est pas une variable et ne peut pas être converti en un corps de clause.

Exemples :

```
?- consult.
Consulting ...

pattes(X, 4) :- quadrupede(X).
pattes(pieuvre, 8).
pattes(X, 2) :- bipede(X).

tetes(X, 1).

quadrupede(cheval).
quadrupede(chien).

bipede(autruche).
bipede(canard).

insecte(mouche).

end_of_file.
true.
```

```
?- clause(pattes(X, 8), Q).
Q = true,
X = pieuvre.
?- clause(pattes(_, 4), Q).
Q ~ quadrupede(tree).
?- clause(pattes(X,Y), Q).
Q = quadrupede(X),
Y = 4,
X ~ tree;
Q = true,
Y = 8,
X = pieuvre;
Q = bipede(X),
Y = 2,
X ~ tree.
?- clause(T, bipede(X)).
error: error(instantiation_error,clause/2)
```

Voir également : current_predicate/1.

close/1, close/2 — Clôture d'un flux

Types : close(@flux_ou_alias, @options_pour_close)
close(@flux_ou_alias)

Description : Sauf cas d'erreur, l'exécution de `close(`$Flux$`,`$Options$`)` ou de `close(`$Flux$`)` réussit toujours. Elle produit la fermeture du flux indiqué par $Flux$, qui doit être le descripteur ou un alias d'un flux ouvert autre qu'un flux standard. Tenter de fermer un flux standard ne provoque pas d'erreur, mais n'a aucun effet.

Si le flux n'est pas un flux standard, la valeur de son descripteur devient invalide, et tout alias du flux est effacé. Dans le cas d'un flux de sortie, le tampon qui lui est associé est vidangé ; ainsi, la fermeture garantit que toutes les écritures logiques faites sur le flux ont été effectivement accomplies.

Si le flux fermé était le flux d'entrée [resp. de sortie] courant, alors le flux standard d'entrée [resp. de sortie] devient le flux d'entrée [resp. de sortie] courant.

Lorsqu'il est présent, l'argument $Options$ représente une liste dont les éléments sont parmi les suivants :

`force(`$Bool$`)` : Si $Bool$ est `false` (c'est l'option par défaut) une erreur se produisant durant l'exécution de `close` sera signalée et empêchera la fermeture effective du flux.

Si $Bool$ est `true`, toute erreur se produisant durant l'exécution de close sera ignorée ; le flux sera fermé quoi qu'il arrive.

Erreurs :

1. $Flux$ est une variable.
2. $Options$ est une variable.
3. $Options$ représente une liste contenant une variable.
4. $Options$ n'est pas une variable et ne représente pas une liste.

5. *Flux* n'est pas une variable et ne représente ni un descripteur de flux ni un alias.
6. Un élément de la liste *Options* n'est pas une option valide.
7. *Flux* n'est pas le descripteur ou un alias d'un flux ouvert.

Exemples :

```
?- open("data.out", write, _, [ alias(sortie) ]).
true.
?- write(sortie, "Premiere ligne"), nl(sortie).
true.
?- write(sortie, "Deuxieme ligne"), nl(sortie).
true.
?- write(sortie, "Derniere ligne"), nl(sortie).
true.
?- close(sortie).
true.
?- open("data.out", write, _, [ alias(sortie) ]).
true.
?- write(sortie, "Un texte"), nl(sortie).
true.
?- close(sortie, [ force(true) ]).
true.
```

Voir également : open/3, open/4.

compound/1 — Test de terme composé

Types : compound(@terme)

Description : `compound(T)` réussit si le terme `T` représente un arbre d'étiquette initiale connue et dont le nombre de fils immédiats est non nul, échoue sinon.

Exemples :

```
?- compound(123).
false.
?- compound(X).
false.
?- compound(f(1)).
true.
?- compound([1,2,3]).
true.
?- compound(2 + 3).
true.
?- compound( - X).
X ~ tree.
?- prolog4.
true.
>> compound(2 + 3).
false.
>> compound( - X).
false.
>> iso.
true.
?-
```

copy_term/2 — Copie de terme

Types : copy_term(?terme,?terme)

Description : `copy_term`(T_1, T_2) réussit si et seulement si T_2 s'unifie avec un terme T qui est une copie à variables renommées de T_1.

Erreurs : Aucune.

Exemples :

```
?- copy_term(X,Y).
Y ~ tree,
X ~ tree.
?- copy_term(X,3).
X ~ tree.
?- copy_term(3, X).
X = 3.
?- copy_term(a+X, X+b).
X = a.
?- copy_term(a+X, X+b), copy_term(a+X, X+b).
false.
?- copy_term(demoen(X,X), demoen(Y,f(Y))).
Y = f(Y),
X ~ tree.
```

Voir également : =../2 .

current_input/1 — Flux d'entrée courant

Types : current_input(?flux)

Description : Sauf cas d'erreur, l'exécution de `current_input`(U) unifie U avec le descripteur du flux d'entrée en service.

Erreurs :

1. U n'est ni une variable, ni un descripteur de flux.

Exemples :

```
?- current_input(X).
X = 6780088.
```

Voir également : open/3, open/4, set_input/1.

current_op/3 — Gestion de la table des opérateurs

Types : current_op(?entier,?spécificateur,?atome)

Description : `current_op`(P, S, A) explore la table des opérateurs et retourne par backtracking les diverses définitions trouvées qui sont unifiables avec les arguments.

L'argument A est l'opérateur (un atome). La priorité P d'un opérateur est un nombre compris entre 1 et 1200. S est le spécificateur : c'est un atome décrivant la classe (infixé, préfixé ou postfixé) et l'associativité du ou des opérateurs qu'on déclare. Cet atome est l'un parmi `fx`, `fy`, `xfx`, `yfx`, `xfy`, `xf`

et `yf`. Ces concepts sont expliqués dans le chapitre «Syntaxe ISO» du présent manuel.

Erreurs :
1. P n'est ni une variable, ni une priorité (nombre entre 1 et 1200).
2. S n'est ni une variable, ni un spécificateur.

Exemples :

```
>> current_op(P,xfy,Z).
Z = (>>),
P = 1200;
Z = (?-),
P = 1200;
Z = (:-),
P = 1200.
>>
```

Voir également : op/3 .

current_output/1 — Flux de sortie courant

Types : current_output(?flux)

Description : Sauf cas d'erreur, l'exécution de `current_output(`U`)` unifie U avec le descripteur du flux de sortie en service.

Erreurs :
1. U n'est ni une variable, ni un descripteur de flux.

Exemples :

```
?- current_output(X).
X = 6780148.
```

Voir également : open/3, open/4, set_output/1.

current_predicate/1 — Recherche de prédicats

Types : current_predicate(?indicateur_de_prédicat)

Description : Sauf cas d'erreur, `current_predicate(`P`)` s'efface en unifiant P avec N/A, où N et A représentent respectivement le nom et l'arité d'une procédure existant dans le programme, pour laquelle l'unification est possible.

Cet effacement produit la création d'un point de choix, dont les alternatives correspondent aux autres procédures ayant la même propriété. L'ordre dans lequel les procédures sont prises en considération n'est pas spécifié.

S'il n'existe aucune telle procédure, `current_predicate(`P`)` échoue.

Erreurs :
1. L'argument n'est ni une variable, ni un terme de la forme N/A.

Exemples :

```
?- consult.
Consulting ...

pattes(X, 4) :- quadrupede(X).
pattes(pieuvre, 8).
pattes(X, 2) :- bipede(X).

tetes(X, 1).

quadrupede(cheval).
quadrupede(chien).

bipede(autruche).
bipede(canard).

insecte(mouche).

end_of_file.
true.
```

```
?- current_predicate(pattes / 2).
true.
?- current_predicate(pattes / N).
N = 2.
?- current_predicate(P).
false.
?- current_predicate(X / 2).
false.
```

Note : Contrairement à `clause`/2, `current_predicate`/1 explore la totalité du programme courant, c'est-à-dire les procédures statiques aussi bien que les procédures dynamiques.

Voir également : clause/2.

current_prolog_flag/2 — Interrogation des flags

Types : current_prolog_flag(?paramètre,?terme)

Description : `current_prolog_flag(A, V)` réussit si `A` est un paramètre implanté et si `V` est la valeur actuelle de ce paramètre. Sinon, échoue en dehors des cas d'erreur.

Erreurs :
1. `A` n'est ni une variable ni un atome, `type_error(atom,1),current_prolog_flag/2`

Exemples :

```
?- current_prolog_flag(max_arity, X).
X = 1000.
?- current_prolog_flag(X, Y).
Y = off,
X = debug;
Y = nyi,
X = unknown;
Y = atom,
X = double_quotes;
Y = off,
X = char_conversion;
Y = simple,
X = interval_mode;
Y = false,
X = bounded;
Y = toward_zero,
X = integer_rounding_function;
Y = 1000,
X = max_arity.
```

Voir également : set_prolog_flag/2.

dynamic/1 — Déclaration de règle

Types : dynamic(+indicateur_de_prédicat)

Description : P étant de la forme N/A, où N est un atome et A un entier non-négatif, `dynamic`(P) déclare le prédicat de nom N et d'arité A comme étant dynamique, c.à.d. comme pouvant être utilisé par les primitives `assert`, `retract`, etc.

Erreurs :
1. P n'est pas un indicateur de prédicat ($atome/entier$).
2. N est négatif.
3. P représente une primitive prédéfinie.

Exemples :

```
>> dynamic(^/(toto,2)).
true.
>> asserta( (toto(1,X) :- write(X), nl) ).
X ~ tree.
```

Notes : 1. Cette primitive ne fait pas partie de la liste des primitives prolog ISO.
2. La directive `:-dynamic/1` effectue la même opération.

Voir également : asserta/1, assertz/1, retract/1.

fail/0 — Echec

Types : fail

Description : `fail` échoue toujours.

Exemples :

```
?- consult.

Consulting ...

echec(R) :- call(R), !, fail.
echec(R).
end_of_file.
true.
?- ami(X).
X = pierre;
X = paul;
X = marie.
?- echec(ami(X)).
false.
?- echec(echec(ami(X))).
X ~ tree.
```

Voir également : true/0.

findall/3 — Liste de solutions

Types : findall(@terme, @terme_exécutable,?liste)

Description : Sauf cas d'erreur, `findall`$(Terme,\ But,\ Liste)$ s'efface en unifiant $Liste$ avec la liste L construite comme suit :

Pour un terme Y, notons $\rho(Y)$ un terme obtenu par recopie de Y et renommage des variables de Y, de telle manière que toutes les variables de $\rho(Y)$ soient nouvelles, i.e. n'apparaissent dans aucun autre terme présentement construit ou en cours de construction.

Soit X une variable n'apparaissant ni dans $Terme$ ni dans But. Dans ces conditions, nous pouvons définir L comme la liste $[\ \rho(X_1), \rho(X_2), ...\rho(X_n)\ ]$, où X_1, X_2, ... X_n sont les valeurs successivement prises par la variable X dans tous les solutions possibles du but : $But, X = Terme$.

Erreurs :

1. Le second argument (But) est une variable.
2. Le second argument (But) n'est pas une variable et ne représente pas un terme exécutable.
3. Le troisième argument ($Liste$) n'est pas une variable et ne représente pas une liste.

Exemples :

```
?- consult.
Consulting ...

pattes(X, 4) :- quadrupede(X).
pattes(pieuvre, 8).
pattes(X, 2) :- bipede(X).
pattes(X, 6) :- insecte(X).

tetes(X, 1).

quadrupede(cheval).
quadrupede(chien).
quadrupede(chat).

bipede(autruche).
bipede(canard).

insecte(mouche).

end_of_file.
true.
```

```
?- pattes(X, N).
N = 4,
X = cheval;
N = 4,
X = chien;
N = 4,
X = chat;
N = 8,
X = pieuvre;
N = 2,
X = autruche;
N = 2,
X = canard;
N = 6,
X = mouche.
?- findall(X, pattes(X, N), L).
L = [cheval,chien,chat,pieuvre,autruche,canard,mouche],
N ~ tree,
X ~ tree.
?- findall([X, N], pattes(X, N), L).
L = [[cheval,4],[chien,4],[chat,4],[pieuvre,8],[autruche,2],
          [canard,2],[mouche,6]],
N ~ tree,
X ~ tree.
```

Voir également : bagof/3, setof/3.

float/1 — Test de flottant

Types : float(@terme)

Description : `float(F)` réussit si le terme `F` représente un arbre réduit à une feuille dont l'étiquette est un flottant IEEE connu.

Exemples :

```
?- float(1).
false.
?- float(1.5).
true.
?- prolog4.
true.
>> float(1.5).
false.
>> X = 1.5.
X = 3/2.
>> float(sqrt(2)).
false.
>> X = sqrt(2).
X ~ cc('>1.4142135','>1.4142136').
>> iso.
true.
?-
```

flush_output/1, flush_output/2 _ Vidange du tampon d'écriture

Types : flush_output(@flux_ou_alias)
flush_output

Description : Sauf erreur, l'exécution de `flush_output(`$Flux$`)` ou `flush_output` réussit toujours. $Flux$ doit être le descripteur ou un alias d'un flux de sortie ; l'exécution de `flush_output(`$Flux$`)` produit la « vidange » du tampon associé. Cette exécution garantit donc que toutes les écritures logiques faites sur le flux indiqué ont été effectivement accomplies.

Erreurs :

1. $Flux$ est une variable.
2. $Flux$ n'est pas une variable et ne représente ni un descripteur de flux ni un alias.
3. $Flux$ n'est pas le descripteur ou un alias d'un flux ouvert.
4. $Flux$ est le descripteur ou un alias d'un flux d'entrée.

Exemples :

```
?- write("Un texte"), flush_output.
Un textetrue.
?- open("data.out", write, _, [ alias(sortie) ]).
true.
?- write(sortie, "Un texte"), flush_output(sortie).
true.
```

Voir également : open/3, open/4.

functor/3 — Extraction/création d'étiquette

Types : functor(-nonvar, +atomique, +entier)
functor(@nonvar,?atomique,?entier)

Description : `functor` ($Terme$, Nom, $Arité$) réussit si et seulement si :

- $Terme$ est un terme composé dont le nom du foncteur est Nom et son arité $Arité$, ou bien si
- $Terme$ est un terme atomique (atome ou nombre) égal à Nom et $Arité$ vaut zéro.

Erreurs :

1. $Terme$ et Nom sont des variables.
2. $Terme$ et $Arité$ sont des variables.
3. $Terme$ est une variable et Nom n'est ni une variable ni un terme atomique.
4. $Terme$ est une variable et $Arité$ n'est ni une variable ni un entier.
5. $Terme$ est une variable, Nom est une constante qui n'est pas un atome, et $Arité$ est différent de 0.
6. $Terme$ est une variable et $Arité$ est un entier plus grand que le paramètre d'implantation `max_arity`.
7. $Terme$ est une variable et $Arité$ est un entier négatif.

Exemples :

```
?- functor(toto(a,b,c), toto,3).
true.
?- functor(toto(a,b,c), X,Y).
Y = 3,
X = toto.
?- functor(X, toto, 3).
X = toto(tree,tree,tree).
?- functor(X, toto,0).
X = toto.
?- functor([_|_], '.', 2).
true.
?- functor([], [],0).
true.
?- functor(1, X,Y).
Y = 0,
X = 1.
?- functor(X, foo, N).
error: error(instantiation_error,functor/3)
```

Voir également : =../3, arg/3, copy_term/2.

get_byte/1, get_byte/2 — Lecture d'un octet

Types : get_byte(@flux_ou_alias,?octet_entré)
get_byte(?octet_entré)

Description : Sauf cas d'erreur, l'exécution de `get_byte(`B`)` [resp. `get_byte(`F`,`B`)`] produit l'obtention d'une donnée depuis le flux d'entrée standard [resp. le flux représenté par F], suivie d'une tentative d'unification de cette donnée avec B. L'exécution en question réussit si et seulement si cette unification est possible.

F doit être est le descripteur ou un alias d'un flux ouvert. Le flux concerné par un appel d'un de ces prédicats doit être d'entrée et binaire. Si le flux n'était pas dans un état « sur la fin du flux » ou « au-delà de la fin du flux », la donnée acquise est le premier octet non encore lu, vu comme un nombre entier. S'il est certain qu'une éventuelle prochaine lecture ne pourra pas obtenir un nouvel octet valide (typiquement : si on vient de lire le dernier caractère d'un fichier non interactif), le flux passe dans l'état « sur la fin du flux ».

Si le flux était dans l'état « sur la fin du flux », alors un appel de `get_byte`/1 ou `get_byte`/2 le fait passer dans l'état « au-delà de la fin du flux » et la donnée acquise est -1.

Si le flux était dans l'état « au-delà de la fin du flux », alors l'effet de l'exécution de `get_byte`/1 ou `get_byte`/2 dépend de la valeur de l'option `eof_action(`A`)`, fixée à l'ouverture du flux.

On notera que, sauf en cas d'erreur ou de fin du flux, l'appel de `get_byte`/1 ou `get_byte`/2 produit toujours l'acquisition d'un caractère et l'avancement d'un cran de la position courante sur le flux, aussi bien lorsque l'appel réussit que lorsqu'il échoue.

Erreurs :

1. F est une variable.
2. B n'est ni une variable ni un octet.
3. F n'est ni une variable, ni un descripteur de flux, ni un alias d'un flux.
4. F n'est pas associé à un flux ouvert.
5. F est associé à un flux de sortie.
6. F est associé à un flux de texte.
7. Le flux d'entrée courant est associé à un flux de texte.
8. F représente un flux dans l'état « au delà de la fin du flux » et la réaction prévue sur une tentative de lecture lorsque le flux est dans cet état est le déclenchement d'une erreur (i.e. F possède les deux propriétés `end_of_stream(past)` et `eof_action(error)`).
9. Le flux d'entrée courant est dans l'état « au delà de la fin du flux » et la réaction prévue sur une tentative de lecture lorsque le flux est dans cet état est le déclenchement d'une erreur (i.e. le flux d'entrée courant possède les deux propriétés `end_of_stream(past)` et `eof_action(error)`).

Voir également : peek_byte/1, peek_byte/2.

get_char/1, get_char/2 — Lecture d'un caractère

Types : get_char(?caractère_entré)
get_char(@flux_ou_alias,?caractère_entré)

Description : Sauf cas d'erreur, l'exécution de `get_char(`C`)` [resp. `get_char(`F`,`C`)`] produit l'obtention d'une donnée depuis le flux d'entrée standard [resp. le flux représenté par F], suivie d'une tentative d'unification de cette donnée avec C. L'exécution en question réussit si et seulement si cette unification est possible.

F doit être le descripteur ou un alias d'un flux ouvert. Le flux concerné par un appel d'un de ces prédicats doit être d'entrée et de texte. Si le flux n'était pas dans un état « sur la fin du flux » ou « au-delà de la fin du flux », la donnée acquise est le premier caractère non encore lu. S'il est certain qu'une éventuelle prochaine lecture ne pourra pas obtenir un nouveau caractère valide (typiquement : si on vient de lire le dernier caractère d'un fichier non interactif), le flux passe dans l'état « sur la fin du flux ».

Si le flux était dans l'état « sur la fin du flux », alors un appel de get_char le fait passer dans l'état « au-delà de la fin du flux » et la donnée acquise est l'atome `end_of_file`.

Si le flux était dans l'état « au-delà de la fin du flux », alors l'effet de l'exécution d'un des prédicats ci-dessus dépend de la valeur de l'option `eof_action(`A`)`, fixée à l'ouverture du flux.

On notera que, sauf cas d'erreur ou de fin du flux, l'appel de `get_char`/1 ou `get_char`/2 produit toujours l'acquisition d'un caractère et l'avancement d'un cran de la position courante sur le flux, aussi bien lorsque l'appel réussit que lorsqu'il échoue.

Erreurs :

1. F est une variable.
2. C n'est ni une variable ni un caractère.
3. F n'est ni une variable, ni un descripteur de flux, ni un alias d'un flux.
4. F n'est pas associé à un flux ouvert.
5. F est associé à un flux de sortie.
6. F est associé à un flux binaire.
7. Le flux d'entrée courant est un flux binaire.
8. F représente un flux dans l'état « au delà de la fin du flux » et la réaction prévue sur une tentative de lecture lorsque le flux est dans cet état est le déclenchement d'une erreur (i.e. F possède les deux propriétés `end_of_stream(past)` et `eof_action(error)`).
9. Le flux d'entrée courant est dans l'état « au delà de la fin du flux » et la réaction prévue sur une tentative de lecture lorsque le flux est

dans cet état est le déclenchement d'une erreur (i.e. le flux d'entrée courant possède les deux propriétés `end_of_stream(past)` et `eof_action(error)`).

10. La donnée extraite du flux n'est pas un caractère.

Voir également : get_code/1, get_code/2, peek_char/1, peek_char/2.

get_code/1, get_code/2 _____ Lecture du code d'un caractère

Types : get_code(?code_caractère_entré)
get_code(@flux_ou_alias,?code_caractère_entré)

Description : Sauf cas d'erreur, l'exécution de `get_code(`C`)` [resp. `get_code(`F`,`C`)`] produit l'obtention d'une donnée depuis le flux d'entrée standard [resp. le flux représenté par F], suivie d'une tentative d'unification de cette donnée avec C. L'exécution en question réussit si et seulement si cette unification est possible.

F doit être le descripteur ou un alias d'un flux ouvert. Le flux concerné par un appel d'un de ces prédicats doit être d'entrée et de texte. Si le flux n'était pas dans un état « sur la fin du flux » ou « au-delà de la fin du flux », la donnée acquise est le code numérique du premier caractère non encore lu. S'il est certain qu'une éventuelle prochaine lecture ne pourra pas obtenir un nouveau caractère valide (typiquement : si on vient de lire le dernier caractère d'un fichier non interactif), le flux passe dans l'état « sur la fin du flux ».

Si le flux était dans l'état « sur la fin du flux », alors un appel de get_code le fait passer dans l'état « au-delà de la fin du flux » et la donnée acquise est le nombre -1.

Si le flux était dans l'état « au-delà de la fin du flux », alors l'effet de l'exécution d'un des prédicats ci-dessus dépend de la valeur de l'option `eof_action(`A`)`, fixée à l'ouverture du flux.

On notera que, sauf cas d'erreur ou de fin du flux, l'appel de `get_code`/1 ou `get_code`/2 produit toujours l'acquisition d'un caractère et l'avancement d'un cran de la position courante sur le flux, aussi bien lorsque l'appel réussit que lorsqu'il échoue.

Erreurs :

1. F est une variable.
2. C n'est ni une variable ni un entier.
3. F n'est ni une variable, ni un descripteur de flux, ni un alias d'un flux.
4. F n'est pas associé à un flux ouvert.
5. F est associé à un flux de sortie.
6. F est associé à un flux binaire.
7. Le flux d'entrée courant est un flux binaire.
8. F représente un flux dans l'état « au delà de la fin du flux » et la réaction prévue sur une tentative de lecture lorsque le flux est dans cet état

est le déclenchement d'une erreur (i.e. F possède les deux propriétés `end_of_stream(past)` et `eof_action(error)`).

9. Le flux d'entrée courant est dans l'état « au delà de la fin du flux » et la réaction prévue sur une tentative de lecture lorsque le flux est dans cet état est le déclenchement d'une erreur (i.e. le flux d'entrée courant possède les deux propriétés `end_of_stream(past)` et `eof_action(error)`).
10. La donnée extraite du flux n'est pas un caractère.
11. C n'est ni une variable ni le code d'un caractère.

Voir également : get_char/1, get_char/2, peek_code/1, peek_code/2.

halt/1
halt/2 — Abandon de Prolog

Types : halt(+entier)
halt

Description : `halt(N)` Abandonne l'exécution de Prolog IV. L'entier `N` est transmis comme «message» au système d'exploitation sous-jacent.

Erreurs :
1. `N` est une variable
   `instantiation_error`
2. `N` n'est ni une variable ni un entier
   `type_error(integer,N)`

Exemples :
```
?- halt(0).
machine %
```

integer/1 — Test d'entier

Types : integer(@terme)

Description : `integer(E)` réussit si le terme `E` représente un arbre réduit à une feuille dont l'étiquette est un entier connu.

Exemples :

```
?- integer(0).
true.
?- integer(0.0).
false.
?- integer(X), X=5.
false.
?- X=5, integer(X).
X = 5.
?- integer(0 + 0).
false.
?- prolog4.
true.
>> integer(0 + 0).
true.
>> integer(1.5 + 1).
false.
>> integer(1.5 + 0.5).
true.
>> iso.
true.
?-
```

is/2 — Evaluation directionnelle

Types : is(?terme, @évaluable), ?terme is @évaluable

Description : `T is E` pose l'équation `T` = `v`, où `v` est le résultat de l'évaluation du terme `E`.

Erreurs :

1. `E` est une variable
   `instantiation_error`

Exemples :

```
?- X = 2+3.
X = 2+3.
?- X is 2+3.
X = 5.
?- 1 is 0.5 + 0.5.
false.
?- X is 0.5 + 0.5.
X = 1.0.
?- X is Y + 2, Y=10.
error: error(instantiation_error,(is)/2)
?- Y=10, X is Y + 2.
X = 12,
Y = 10.
```

Voir également : =:=/2, =\=/2, </2.

nl/1, nl/0 — Ecriture d'une fin de ligne

Types : nl(@flux_ou_alias)
nl

Description : L'exécution de `nl` [resp. `nl(`F`)`] envoie le caractère « nouvelle ligne » sur le flux de sortie standard [resp. le flux de sortie spécifié par F].

F doit être le descripteur ou un alias d'un flux ouvert. Le flux concerné par un appel d'un de ces prédicats doit être de sortie et de texte.

Dans tous les cas, la position courante sur le flux en question est incrémentée d'une unité.

Erreurs :

1. F est une variable.
2. F n'est ni une variable, ni un descripteur de flux, ni un alias d'un flux.
3. F n'est pas associé à un flux ouvert.
4. F est associé à un flux d'entrée.
5. F est associé à un flux binaire.
6. Le flux de sortie courant est un flux binaire.

Exemples :

```
?- write(a), write(b).
abtrue.
?- write(a), nl, write(b), nl.
a
b
true.
?- open("data.out", write, _, [ alias(sortie) ]).
true.
?- write(sortie, a), nl(sortie), write(sortie, b).
true.
?- close(sortie).
true.
?- halt.
D :  type data.out
a
b
D :
```

Voir également : put_char/1, put_char/2.

nonvar/1 — Test de non-variable

Types : nonvar(@terme)

Description : `nonvar(T)` réussit si le terme `T` n'est pas réduit à une variable. Echoue dans le cas inverse. Plus précisément, soit n le nombre de fils immédiats de l'arbre représenté par le terme `T`, `nonvar(T)` réussit si l'une au moins des deux conditions suivantes est vérifiée :

1. l'étiquette initiale de l'arbre représenté par le terme `T` est connue,

2. l'un au moins des deux systèmes $S \cup \{n = 0\}$, $S \cup \{n \neq 0\}$ est insoluble.

Exemples :

```
?- nonvar(X).
false.
?- X = 0, nonvar(X).
X = 0.
?- X = Y, nonvar(X).
false.
?- X = 1, nonvar(X).
X = 1.
?- nonvar(X), X = 1.
false.
```

Voir également : var/1.

number/1 — Test de nombre

Types : number(@terme)

Description : `number(R)` réussit si le terme `R` représente un arbre réduit à une feuille dont l'étiquette initiale est un nombre connu.

Exemples :

```
?- number(0).
true.
?- number(1.0).
true.
?- number(2 + 3).
false.
?- prolog4.
true.
>> number(2 + 3).
true.
>> number(2 / 3).
true.
>> number(sqrt(2)).
false.
>> iso.
true.
```

Voir également : float/1, integer/1, rational/1.

number_chars/2 — Eclatement d'un nombre

Types : number_chars(+nombre, ?liste_de_caractères)
number_chars(-nombre, +liste_de_caractères)

Description : `number_chars(N,L)` réussit si `L` est une liste dont les éléments sont des caractères correspondants à une séquence de caractères qui constituent une représentation correcte de `N`.

Erreurs :

1. Si `N` est une variable et `L` est une liste partielle ou une liste dont un élément au moins est une variable,
2. Si `N` n'est ni une variable ni un nombre,

3. Si N est une variable et L n'est ni une liste ni une une liste partielle,
4. Si il existe un élément E de L qui n'est pas un caractère.
5. Si L n'est pas une liste de caractères qui forment un nombre.

Exemples :

```
?- number_chars(12345, X).
X = ['1','2','3','4','5'].
?- number_chars(X, ['1', '0', '0', '0']).
X = 1000.
?- number_chars(X, ['1', '0', a, '0']).
error: error(syntax_error(p4_not_a_number_syntax),['1','0',a,'0'])
```

Voir également : number_codes/2.

number_codes/2 — Eclatement d'un nombre

Types : number_codes(+nombre,?liste_de_codes_caractères)
number_codes(-nombre, +liste_de_codes_caractères)

Description : `number_codes(A,L)` réussit si L est la liste dont les éléments sont les codes des caractères correspondants aux caractères successifs formant l'atome A.

Erreurs :

1. Si N est une variable et L est une liste partielle ou une liste dont un élément au moins est une variable,
2. Si N n'est ni une variable ni un nombre,
3. Si N est une variable et L n'est ni une liste ni une une liste partielle,
4. Si il existe un élément E de L qui n'est pas un code de caractère.

Exemples :

```
?- number_codes(12345, X).
X = [49,50,51,52,53].
?- number_codes(X, [49, 48, 48, 48]).
X = 1000.
?- number_codes(X, [49, 48, 65, 48]).
error: error(syntax_error(p4_not_a_number_syntax),[49,48,65,48])
```

Voir également : number_chars/2.

once/1 — Exécution unique

Types : once(+terme_exécutable)

Description : `once(T)` se comporte comme `call(T)`, mais n'est pas réexécutable. On effectue donc l'appel à T, en ne laissant aucun point de choix.

Erreurs :

1. T est une variable

   `error: instantiation_error`

2. T n'est ni une variable, ni un terme exécutable

   `error: undefined_call(T).`

Exemples :

```
?- ami(X).
X = pierre;
X = paul;
X = marie.
?- once(ami(X)).
X = pierre.
?- once(ami(pierre)).
true.
?- once(ami(jean)).
false.
?- once(X).
error: error(instantiation_error,call/1)
?- once(1).
error: error(type_error(callable,1),'$Control_construct')
```

Voir également : !/0, call/1, repeat/0.

op/3 — Gestion de la table des opérateurs

Types : op(@entier, @spécificateur, @atome)
op(@entier, @spécificateur, @liste_d'atomes)

Description : Sauf cas d'erreur, l'exécution de `op(`P`,` S`,` A`)` réussit. Cette primitive permet d'altérer pendant l'exécution la table des opérateurs. L'argument A peut être un atome ou une liste d'atomes. La priorité d'un opérateur est un nombre compris entre 1 et 1200. S est le spécificateur : c'est un atome décrivant la classe (infixé, préfixé ou postfixé) et l'associativité du ou des opérateurs qu'on déclare. Ce spécificateur est un atome parmi `fx`, `fy`, `xfx`, `yfx`, `xfy`, `xf` et `yf`. Ces concepts sont expliqués dans le chapitre « Syntaxe ISO » du présent manuel.

Si P est une priorité, alors le ou les opérateurs dans A sont ajoutés à la table, avec cette priorité et la spécification S.

Si P vaut zéro, le ou les opérateurs dans A, de la classe indiquée par S sont otés de la table des opérateurs.

Erreurs :

1. P est une variable.
2. S est une variable.
3. A est une variable, ou une liste dont un élément est une variable.
4. P n'est ni une variable, ni un entier.
5. S n'est ni une variable, ni un atome.
6. A n'est ni une variable, ni un atome, ni une liste.
7. Un élément de la liste A n'est ni une variable, ni un atome.
8. P n'est pas compris entre 0 et 1200.
9. S n'est pas un spécificateur valide.
10. A est la virgule `','`.
11. Un élément de la liste A est `','`.

12. S est un spécificateur tel que A possèderait un ensemble invalide de spécificateurs.

Exemples :

```
>> write(a++(b++c)).
...
error: error(syntax_error(p4_cant_parse),read_term)
>> op(30, xfy, ++).
true.
>> write(a++(b++c)), nl.
a++b++c
true.
>>
```

Note : Dans l'exemple, la première requête ne peut être lue par Prolog IV : elle est syntaxiquement incorrecte de par l'inexistence de l'opérateur ++ dans la table des opérateurs. La suite de l'exemple montre comment cette table est étendue.

Voir également : current_op/3 .

open/3, open/4 — Ouverture d'un flux d'entrée ou de sortie

Types : open(@source_ou_puits, @mode_es, -flux, @options_pour_open)
open(@source_ou_puits, @mode_es, -flux)

Description : Sauf cas d'erreur, l'exécution de `open(`$SourcePuits$`,` $Mode$`,` $Flux$`,` $Options$`)` ou `open(`$SourcePuits$`,` $Mode$`,` $Flux$`)` réussit toujours. $SourcePuits$ doit représenter une chaîne de caractères qui identifie une source ou un puit (fichier, console, communication avec un autre processus, etc.), en accord avec la manière dont le système d'exploitation sous-jacent nomme de telles entités.

L'argument $Mode$ représente un atome qui spécifie la nature des opérations qui seront effectuées sur le flux d'entrée et sortie à ouvrir. Les modes possibles sont : `read` (lecture), `write` (écriture) et `append` (allongement).

L'argument $Flux$ doit être une variable qui, au retour de l'exécution de open, aura pour valeur le descripteur du flux ouvert.

Lorsqu'il est présent, l'argument $Options$ représente une liste dont les éléments sont parmi :

`type(`T`)` : Indique si le fichier est de texte ou binaire. T doit être un des atomes `text` ou `binary`. Lorsque cette option est absente, le flux est supposé être de texte.

`reposition(`B`)` : Si la valeur de B est `true`, cette option signale que l'on souhaite avoir le droit de repositionner le flux par des appels du prédicat `set_stream_position/2`. Une erreur se produit si les propriétés du flux et du système d'exploitation sous-jacent rendent la chose impossible.

`alias(`A`)` : A doit être un atome qui n'est pas déjà un alias pour un autre flux ouvert. Cet atome sera associé au flux présentement ouvert, et constituera un moyen commode de désigner ce flux dans les opérations d'entrée/sortie.

`eof_action(`A`)` : A est un atome qui précise la suite qu'il faut donner à une tentative de lecture sur un flux qui se trouve dans la position « au-delà de la fin du flux », c'est-à-dire dont des lectures précédentes ont déjà extrait la dernière donnée valide et l'objet conventionnel qui indique la fin du fichier.

Les valeurs possibles pour A sont :

`error` : une lecture au delà de la fin du fichier doit déclencher une erreur

`eof_code` : les lectures au-delà de la fin du fichier produisent l'objet conventionnel qui indique la fin du fichier.

`reset` : le fichier est réinitialisé (il faut que cela ait un sens pour l'organe d'entrée/sortie et le système d'exploitation sous-jacent).

Erreurs :

1. $SourcePuits$ est une variable.
2. $Mode$ est une variable.
3. $Options$ est une variable.
4. $Options$ représente une liste contenant une variable.
5. $Mode$ n'est pas une variable et ne représente pas un atome.
6. $Options$ n'est pas une variable et ne représente pas une liste.
7. $Flux$ n'est pas une variable.
8. $SourcePuits$ ne représente pas un nom correct pour une source ou un puits.
9. $Mode$ représente bien un atome, mais ne correspond pas à un mode d'entrée/sortie correct.
10. Un élément de la liste $Options$ n'est pas une option valide.
11. La source ou le puits spécifié par $SourcePuits$ n'existe pas.
12. La source ou le puits spécifié par $SourcePuits$ ne peut pas être ouvert.
13. Un élément de la liste $Options$ est de la forme `alias(`A`)` et A est déjà associé par ailleurs à un flux ouvert.
14. Un élément de la liste Options est de la forme `reposition(true)` alors qu'il n'est pas possible de repositionner le flux.

Exemples :

```
?- open("data.out", write, F),
        write(F, "Ligne 1"), nl(F), close(F).
F = 6780268.
?- open("data.out", append, F),
        write(F, "Ligne 2"), nl(F), close(F).
F = 6780268.
?- open("data.out", append, F),
        write(F, "Ligne 3"), nl(F), close(F).
F = 6780268.
?- halt.
D :  type trace.log
Ligne 1
Ligne 2
Ligne 3
D :
?- open("data.out", write, _, [ alias(sortie) ]).
true.
?- write(sortie, "Premiere ligne"), nl(sortie).
true.
?- write(sortie, "Deuxieme ligne"), nl(sortie).
true.
?- write(sortie, "Derniere ligne"), nl(sortie).
true.
?- close(sortie).
true.
```

Voir également : close/1, close/2.

peek_byte/1, peek_byte/2 — Prochain octet à lire

Types : peek_byte(@flux_ou_alias,?octet_entré)
peek_byte(?octet_entré)

Description : Sauf cas d'erreur, l'exécution de `peek_byte(`C`)` [resp. `peek_byte(`F`,`C`)`] se traduit par une tentative d'unification de C avec une donnée déduite du flux d'entrée standard [resp. le flux représenté par F], sans aucune modification de l'état de ce flux.

F doit être le descripteur ou un alias d'un flux ouvert. Le flux concerné par un appel d'un de ces prédicats doit être d'entrée et binaire.

Si le flux n'était pas dans un état « sur la fin du flux » ou « au-delà de la fin du flux », la donnée en question est le premier octet non encore lu. Si le flux était dans l'état « sur la fin du flux », alors la donnée acquise est la valeur -1.

Si le flux était dans l'état « au-delà de la fin du flux », l'effet de l'exécution d'un des prédicats ci-dessus dépend de la valeur de l'option `eof_action(`A`)`, fixée à l'ouverture du flux.

Erreurs :

1. F est une variable.
2. B n'est ni une variable ni un octet.
3. F n'est ni une variable, ni un descripteur de flux, ni un alias d'un flux.
4. F n'est pas associé à un flux ouvert.

5. F est associé à un flux de sortie.
6. F est associé à un flux de texte.
7. Le flux d'entrée courant est associé à un flux de texte.
8. F représente un flux dans l'état « au delà de la fin du flux » et la réaction prévue sur une tentative de lecture lorsque le flux est dans cet état est le déclenchement d'une erreur (i.e. F possède les deux propriétés `end_of_stream(past)` et `eof_action(error)`).
9. Le flux d'entrée courant est dans l'état « au delà de la fin du flux » et la réaction prévue sur une tentative de lecture lorsque le flux est dans cet état est le déclenchement d'une erreur (i.e. le flux d'entrée courant possède les deux propriétés `end_of_stream(past)` et `eof_action(error)`).

Voir également : get_byte/1, get_byte/2.

peek_char/1, peek_char/2 — Prochain caractère à lire

Types : peek_char(?caractère_entré)
peek_char(@flux_ou_alias,?caractère_entré)

Description : Sauf cas d'erreur, l'exécution de `peek_char(`C`)` [resp. `peek_char(`F`,`C`)`] se traduit par une tentative d'unification de C avec une donnée déduite du flux d'entrée standard [resp. le flux représenté par F], sans aucune modification de l'état de ce flux.

F doit être le descripteur ou un alias d'un flux ouvert. Le flux concerné par un appel d'un de ces prédicats doit être d'entrée et de texte. Si le flux n'était pas dans un état « sur la fin du flux » ou « au-delà de la fin du flux », la donnée en question est le premier caractère non encore lu.

Si le flux était dans l'état « sur la fin du flux », alors la donnée acquise est l'atome `end_of_file`.

Si le flux était dans l'état « au-delà de la fin du flux », alors l'effet de l'exécution d'un des prédicats ci-dessus dépend de la valeur de l'option `eof_action(`A`)`, fixée à l'ouverture du flux.

Erreurs :
1. F est une variable.
2. C n'est ni une variable ni un caractère.
3. F n'est ni une variable, ni un descripteur de flux, ni un alias d'un flux.
4. F n'est pas associé à un flux ouvert.
5. F est associé à un flux de sortie.
6. F est associé à un flux binaire.
7. Le flux d'entrée courant est un flux binaire.
8. F représente un flux dans l'état « au delà de la fin du flux » et la réaction prévue sur une tentative de lecture lorsque le flux est dans cet état

est le déclenchement d'une erreur (i.e. F possède les deux propriétés `end_of_stream(past)` et `eof_action(error)`).

9. Le flux d'entrée courant est dans l'état « au delà de la fin du flux » et la réaction prévue sur une tentative de lecture lorsque le flux est dans cet état est le déclenchement d'une erreur (i.e. le flux d'entrée courant possède les deux propriétés `end_of_stream(past)` et `eof_action(error)`).
10. La première donnée du flux n'est pas un caractère.

Voir également : get_char/1, get_char/2, peek_code/1, peek_code/2.

peek_code/1, peek_code/2 ___ Code du prochain caractère à lire

Types : peek_code(?code_caractère_entré)
peek_code(@flux_ou_alias,?code_caractère_entré)

Description : Sauf cas d'erreur, l'exécution de `peek_code(`C`)` [resp. `peek_code(`F`,`C`)`] se traduit par une tentative d'unification de C avec une donnée déduite du flux d'entrée standard [resp. le flux représenté par F], sans aucune modification de l'état de ce flux.

F doit être le descripteur ou un alias d'un flux ouvert. Le flux concerné par un appel d'un de ces prédicats doit être d'entrée et de texte. Si le flux n'était pas dans un état « sur la fin du flux » ou « au-delà de la fin du flux », la donnée en question est le code du premier caractère non encore lu.

Si le flux était dans l'état « sur la fin du flux », alors la donnée acquise est la valeur -1.

Si le flux était dans l'état « au-delà de la fin du flux », alors l'effet de l'exécution d'un des prédicats ci-dessus dépend de la valeur de l'option `eof_action(`A`)`, fixée à l'ouverture du flux.

Erreurs :

1. F est une variable.
2. C n'est ni une variable ni un entier.
3. F n'est ni une variable, ni un descripteur de flux, ni un alias d'un flux.
4. F n'est pas associé à un flux ouvert.
5. F est associé à un flux de sortie.
6. F est associé à un flux binaire.
7. Le flux d'entrée courant est un flux binaire.
8. F représente un flux dans l'état « au delà de la fin du flux » et la réaction prévue sur une tentative de lecture lorsque le flux est dans cet état est le déclenchement d'une erreur (i.e. F possède les deux propriétés `end_of_stream(past)` et `eof_action(error)`).
9. Le flux d'entrée courant est dans l'état « au delà de la fin du flux » et la réaction prévue sur une tentative de lecture lorsque le flux est

dans cet état est le déclenchement d'une erreur (i.e. le flux d'entrée courant possède les deux propriétés `end_of_stream(past)` et `eof_action(error)`).

10. La première donnée du flux n'est pas un caractère.
11. C n'est ni une variable ni le code d'un caractère.

Voir également : get_code/1, get_code/2, peek_char/1, peek_char/2.

put_byte/1, put_byte/2 — Ecriture d'un octet

Types : put_byte(@flux_ou_alias, @octet)
put_byte(@octet)

Description : Sauf cas d'erreur, l'exécution de `put_byte(`B`)` [resp. `put_byte(`F`,`B`)`] envoie l'octet specifié par la valeur de B sur le flux de sortie standard [resp. le flux de sortie spécifié par F].

F doit être le descripteur ou un alias d'un flux ouvert. Le flux concerné par un appel d'un de ces prédicats doit être de sortie et binaire.

Dans tous les cas, la position courante sur le flux en question est incrémentée d'une unité.

Erreurs :

1. F est une variable.
2. B est une variable.
3. B n'est pas une variable et ne représente pas un octet.
4. F n'est ni une variable, ni un descripteur de flux, ni un alias d'un flux.
5. F n'est pas associé à un flux ouvert.
6. F est associé à un flux d'entrée.
7. F est associé à un flux de texte.
8. Le flux d'entrée courant est associé à un flux de texte.

Voir également : get_byte/1, get_byte/2.

put_char/1, put_char/2 — Ecriture d'un caractère

Types : put_char(@flux_ou_alias, @caractère)
put_char(@caractère)

Description : Sauf cas d'erreur, l'exécution de `put_char(`C`)` [resp. `put_char(`F`,`C`)`] envoie le caractère specifié par la valeur de C sur le flux de sortie standard [resp. le flux de sortie spécifié par F].

F doit être le descripteur ou un alias d'un flux ouvert. Le flux concerné par un appel d'un de ces prédicats doit être de sortie et de texte.

Dans tous les cas, la position courante sur le flux en question est incrémentée d'une unité.

Erreurs :

1. F est une variable.
2. C est une variable.
3. C n'est ni une variable, ni un caractère.
4. F n'est ni une variable, ni un descripteur de flux, ni un alias d'un flux.
5. F n'est pas associé à un flux ouvert.
6. F est associé à un flux d'entrée.
7. F est associé à un flux binaire.
8. Le flux de sortie courant est un flux binaire.

Voir également : put_code/1, put_code/2.

put_code/1, put_code/2 — Ecriture du code d'un caractère

Types : put_code(@flux_ou_alias, @code_caractère)
put_code(@caractère)

Description : Sauf cas d'erreur, l'exécution de `put_code(`C`)` [resp. `put_code(`F`,`C`)`] envoie le caractère ayant C pour code interne sur le flux de sortie standard [resp. le flux de sortie spécifié par F].

F doit être le descripteur ou un alias d'un flux ouvert. Le flux concerné par un appel d'un de ces prédicats doit être de sortie et de texte.

Dans tous les cas, la position courante sur le flux en question est incrémentée d'une unité.

Erreurs :

1. F est une variable.
2. C est une variable.
3. C n'est ni une variable ni un entier.
4. F n'est ni une variable, ni un descripteur de flux, ni un alias d'un flux.
5. F n'est pas associé à un flux ouvert.

6. F est associé à un flux d'entrée.
7. F est associé à un flux binaire.
8. Le flux de sortie courant est un flux binaire.

Voir également : put_char/1, put_char/2.

rational/1 — Test de rationnel

Types : rational(@terme)

Description : `rational(Q)` réussit si le terme `Q` représente un nombre rationnel en précision infinie (un entier ou une fraction).

Note : Ce prédicat est fourni en Prolog IV pour permettre d'identifier les constantes rationnelles, qui forment un nouveau type Prolog. Un prédicat discriminant est demandé par la norme ISO pour tout nouveau type introduit.

read_term/3, read_term/2 read/2, read/1 — Lecture d'un terme

Types : read_term(@flux_ou_alias,?terme, +options_pour_read)
read_term(?terme, +options_pour_read)
read(@flux_ou_alias,?terme)
read(?terme)

Description : Sauf cas d'erreur, l'exécution de `read_term(`$Terme$`,`$Options$`)` [resp. `read_term(`$Flux$`,`$Terme$`,`$Options$`)`] produit la construction d'un terme, formé à partir des caractères extraits du flux d'entrée standard [resp. le flux d'entrée représenté par $Flux$], suivie d'une tentative d'unification de ce terme avec l'argument $Terme$. L'exécution réussit si et seulement si cette unification est possible. Les primitives `read` appellent `read_term` avec une liste vide dans $Options$.

La stratégie adoptée pour la lecture consiste à lire autant d'unités lexicales qu'il s'en présente, jusqu'à la rencontre d'un point, et à tenter ensuite l'analyse de la suite d'unités ainsi formée selon la syntaxe d'un terme.

Si la valeur de l'indicateur `char_conversion` est `on`, chaque caractère lu, s'il n'est pas *quoté*, est converti selon une éventuelle table de conversion préalablement définie (voir la directive `char_conversion/2`).

L'argument $Options$ est une liste dont les éléments sont parmi les suivants :

`variables(`$Vars$`)` : Après l'acquisition d'un terme, $Vars$ est une liste formée des variables du terme lu, dans l'ordre de leur apparition lors de la lecture (i.e. de la gauche du terme vers sa droite). Les variables anonymes figurent dans cette liste.

`variable_names(`$Liste\_NV$`)` : Après l'acquisition d'un terme, $Liste\_NV$ doit s'unifier avec une liste dont chaque élément est de la forme $A = V$, où V est une variable nommée du terme lu et A est un atome dont les caractères sont ceux de V. Les variables anonymes ne figurent pas dans $Liste\_VN$.

`singletons(`$Liste\_VN$`)` : Après l'acquisition d'un terme, $Liste\_VN$ doit s'unifier avec une liste dont chaque élément est de la forme $A = V$, où V est une variable nommée qui apparaît une et une seule fois dans le terme lu et A est un atome dont les caractères sont ceux de V. Les variables anonymes ne figurent pas dans $Liste\_VN$.

Erreurs :

1. $Flux$ est une variable.
2. $Options$ est une variable.
3. $Options$ est une liste contenant une variable.
4. $Flux$ n'est ni une variable, ni un descripteur, ni un alias d'un flux.
5. $Options$ n'est ni une variable ni une liste.
6. Un élément de la liste $Options$ n'est ni une variable ni une option de lecture valide.
7. $Flux$ n'est pas associé à un flux ouvert.
8. $Flux$ est associé à un flux de sortie.
9. $Flux$ est associé à un flux binaire.
10. Le flux d'entrée courant est un flux binaire.
11. $Flux$ a les deux propriétés `end_of_stream(past)` et `eof_action(error)`.
12. Le flux d'entrée courant a les deux propriétés `end_of_stream(past)` et `eof_action(error)`.
13. Le terme construit dépasse les limites données par les indicateurs `max_arity`, `max_integer` ou `min_integer`.
14. Un ou plusieurs caractères ont été lus, mais ils ne peuvent pas être reconnus comme une suite d'unités lexicales correctes.
15. Compte tenu de l'ensemble d'opérateurs couramment définis, la séquence d'unités lexicales formée ne peut pas être reconnue comme un terme correct.

Exemples :

```
?- read(T).
f(X,Y,X,_,X).
A ex
T ~ f(A,tree,A,tree,A),
A ~ tree.
?- read_term(T, [variables(L)]).
f(X,Y,X,_,X).
A ex B ex C ex
L = [C,B,A],
T = f(C,B,C,A,C),
A ~ tree,
B ~ tree,
C ~ tree.
```

```
?- read_term(T, [variable_names(L)]).
f(X,Y,X,_,X).
A ex B ex
L = ['X'=B,'Y'=A],
T ~ f(B,A,B,tree,B),
A ~ tree,
B ~ tree.
```

```
?- read_term(T, [singletons(L)]).
f(X,Y,X,Z,X).
A ex B ex C ex
L = ['Y'=B,'Z'=A],
T = f(C,B,C,A,C),
A ~ tree,
B ~ tree,
C ~ tree.
?- read_term(T, [variables(LV),
      variable_names(LN), singletons(LS)]).
   f(X,Y,X,_,X).
A ex B ex C ex
LS = ['Y'=A],
LN = ['X'=B,'Y'=A],
LV = [B,A,C],
T = f(B,A,B,C,B),
A ~ tree,
B ~ tree,
C ~ tree.
```

```
?- consult.
Consulting ...

job(TE, TL, LV) :-
    open("data.tmp", write, Out),
    write(Out, TE), write(Out, .), nl(Out),
    close(Out), open("data.tmp", read, In),
    read_term(In, TL, [ variables(LV) ]).

end_of_file.
true.
```

```
?- job(123, X, Y).
Y = [],
X = 123.
?- job(f(X,Y,X,_,Z), T, V).
A ex B ex C ex D ex
V = [D,C,B,A],
T = f(D,C,D,B,A),
Z ~ tree,
Y ~ tree,
X ~ tree,
A ~ tree,
B ~ tree,
C ~ tree,
D ~ tree.
```

Voir également : op/3, current_op/3, write/1.

repeat/0 — Multiples exécutions

Types : repeat

Description : `repeat` réussit toujours et est réexécutable. Par exemple, après la requête `repeat,write('hello'),fails` la chaine 'hello' est imprimée indéfiniment sur le flux de sortie courant.

Exemples :

```
?- repeat.
true;
true;
true;
true;
etc.
?- repeat, write("Bonjour"), nl, fail.
Bonjour
Bonjour
Bonjour
Bonjour
etc.
```

Voir également : !/0, call/1, once/1.

retract/1 — Suppression de clauses

Types : retract(+clause)

Description : Cas 1. Si T s'unifie avec un terme ayant `:-/2` pour foncteur principal, alors l'exécution de `retract(`T`)` unifie T avec $Tete$:- $Corps$, où $Tete$:- $Corps$ est la première des clauses du programme pour lesquelles cette unification est possible. En même temps, cette clause est supprimée du programme.

Cette exécution produit la création d'un point de choix, dont les alternatives correspondent aux autres clauses $Tete$:- $Corps$ ayant la même propriété. En Prolog IV, ces diverses clauses sont prises en considération dans l'ordre où elles figurent dans le programme.

S'il n'existe aucune clause ayant la propriété indiquée, `retract(`T`)` échoue.

Cas 2. Si T ne s'unifie pas avec un terme ayant `:-/2` pour foncteur principal, alors $retract(T)$ unifie T avec $Fait$, où $Fait$. est le premier des faits du programme courant pour lesquels cette unification est possible. En même temps, ce fait est supprimé du programme. Les autres points de la description du cas 1 restent valables.

Dans un cas comme dans l'autre, T doit représenter un arbre suffisamment connu pour que le prédicat (nom et arité) de la clause soit connu. De plus, ce prédicat doit être dynamique.

Erreurs :

1. L'argument est une variable.
2. Le premier argument de T (cas 1) ou T tout entier (cas 2) ne peut pas être converti en une tête de clause.

3. Le prédicat de la clause déterminée par T est celui d'une procédure statique.

Notes : 1. Une fois créée, une procédure continue d'exister même lorsque toutes ses clauses sont supprimées par des exécutions de `retract`/1. En particulier, à la suite des exécutions montrées ci-dessus, l'appel

```
?- clause(pattes(X,Y), Z).
no
```

échoue, car la procédure `pattes`/2 n'a plus de clause, mais l'appel suivant réussit :

```
?- current_predicate(pattes/N).
N=2
```

2. La norme ISO du langage Prolog préconise le « point de vue logique » pour les modifications du programme courant. Cela signifie que si l'exécution d'un appel a pour effet d'ajouter ou de soustraire des clauses à la procédure correspondant à cet appel, alors les modifications ne sont effectives que lors des exécutions ultérieures de cette procédure ; les clauses ajoutées ou supprimées n'affectent pas l'exécution en cours.

3. Seules les clauses des procédures dynamiques peuvent être supprimées par des exécutions de `retract`/1.

Exemples :

```
?- consult.
Consulting ...

pattes(X, 4) :- quadrupede(X).
pattes(pieuvre, 8).
pattes(X, 2) :- bipede(X).
pattes(X, 6) :- insecte(X).

tetes(X, 1).

quadrupede(cheval).
quadrupede(chien).

bipede(autruche).
bipede(canard).

insecte(mouche).

end_of_file.
true.
```

```
?- pattes(X, N), N >= 6.
N = 8,
X = pieuvre;
N = 6,
X = mouche.
?- retract((pattes(_, _) :- insecte(_))).
true.
?- pattes(X, N), N >= 6.
N = 8,
X = pieuvre.
?- retract((pattes(_, 8) :- _ )).
true.
?- pattes(X, N), N >= 6.
false.
?- pattes(X, 4).
X = cheval;
X = chien.
?- retract(quadrupede(chien)).
true.
?- pattes(X, 4).
X = cheval.
```

Voir également : asserta/1, assertz/1, abolish/1.

setof/3 — Liste de solutions

Types : setof(@terme, +terme_exécutable,?liste)

Description : Sauf cas d'erreur, l'exécution de `setof` $(Terme, But, Liste)$ unifie $Liste$ avec la liste L construite de la manière suivante :

S étant le système de contraintes courantes, soit I une affectation des variables non existentielles de But n'apparaissant pas dans $Terme$, telle que But s'efface sous les contraintes $S \cup I$.

Dans ces conditions, X étant une variable qui n'apparaît ni dans $Terme$ ni dans But, L est la liste des valeurs successivement prises par X lorsque, avec le système de contraintes $S \cup I$, on efface de toutes les manières possibles le but : $But, X = Terme$.

La liste L est sans répétition et ordonnée selon l'ordre des termes.

Cet effacement produit la création d'un point de choix dont les alternatives correspondent aux autres choix possibles pour I (i.e. les autres affectations possibles des variables libres non existentielles de But).

Erreurs :

1. Le terme déquantifié correspondant à But est une variable.
2. Le terme déquantifié correspondant à But n'est ni une variable ni un terme exécutable.
3. $Liste$ n'est ni une variable ni une liste.

Exemples :

```
?- consult.
Consulting ...

pattes(X, 4) :- quadrupede(X).
pattes(pieuvre, 8).
pattes(X, 2) :- bipede(X).
pattes(X, 6) :- insecte(X).

tetes(X, 1).

quadrupede(cheval).
quadrupede(chien).
quadrupede(chat).
quadrupede(cheval).
quadrupede(chien).

bipede(autruche).
bipede(canard).

insecte(mouche).

end_of_file.
true.
?- pattes(X, 4).
X = cheval;
X = chien;
X = chat;
X = cheval;
X = chien.
?- bagof(X, pattes(X, 4), L).
L = [cheval,chien,chat,cheval,chien],
X ~ tree.
?- setof(X, pattes(X, 4), L).
L = [chat,cheval,chien],
X ~ tree.
```

Voir également : bagof/3, findall/3.

set_input/1 — Sélection du flux d'entrée courant

Types : set_input(?flux_ou_alias)

Description : Sauf cas d'erreur, `set_input(`U`)` réussit toujours ; U doit représenter le descripteur d'un flux ouvert en entrée, ou un alias d'un tel flux, qui devient alors le flux d'entrée courant

Erreurs :

1. U est une variable.
2. U n'est ni une variable, ni un descripteur de flux, ni un alias.
3. U n'identifie pas un flux ouvert.
4. U identifie un flux de sortie.

Exemples :

```
?- open("data.in", read, _, [alias(in)]).
true.
?- set_input(in), read(X), set_input(user_input).
X = 111.
```

Voir également : current_input/1, open/3, open/4.

set_output/1 — Sélection du flux de sortie courant

Types : set_output(?flux_ou_alias)

Description : Sauf cas d'erreur, `set_output(`U`)` réussit toujours ; U doit représenter le descripteur d'un flux ouvert en sortie, ou un alias d'un tel flux, qui devient alors le flux de sortie courant

Erreurs :

1. U est une variable.
2. U n'est ni une variable, ni un descripteur de flux, ni un alias.
3. U n'identifie pas un flux ouvert.
4. U identifie un flux d'entrée.

Exemples :

```
?- consult.
Consulting ...

debut_trace :-
        open("trace.log", write, _, [ alias(trace) ]).

tracer(T) :-
        write(T), nl,
        current_output(S), set_output(trace),
        write(T), nl,
        set_output(S).

fin_trace :- close(trace).

end_of_file.
true.
?- debut_trace.
true.
?- tracer(ligne(1)).
ligne(1)
true.
?- tracer(ligne(2)).
ligne(2)
true.
?- tracer(ligne(3)).
ligne(3)
true.
?- fin_trace.
true.
?- halt.
D :  type trace.log
ligne(1)
ligne(2)
ligne(3)
D :
```

Voir également : current_output/1, open/3, open/4.

set_prolog_flag/2 — Mise à jour des paramètres

Types : set_prolog_flag(@paramètre, @terme)

Description : `set_prolog_flag(A,T)` met à jour la valeur du flag représenté par `A` si il existe en lui affectant la valeur représentée par `T` si elle est valide.

Erreurs :

1. `A` ou `T` est une variable,
   `instantiation_error`
2. `A` n'est ni une variable ni un atome,
   `type_error(atom,A)`
3. `A` est un atome mais représente un flag invalide,
   `domain_error(prolog_flag,A)`
4. `A` est un atome représentant un flag valide, mais `T` n'est pas une valeur appropriée pour ce flag,
   `domain_error(flag_value,V)`
5. `A` est un atome représentant un flag valide, `T` est une valeur appropriée pour ce flag, mais `A` n'est pas modifiable,
   `permission-error(modify,flag,A)`

Exemples :

```
?- set_prolog_flag(double_quotes, atom).
true.
?- set_prolog_flag(max_arity, 1500).
error: error(permission_error(modify,max_arity,1500),set_prolog_flag/2)
```

Voir également : current_prolog_flag/2.

set_stream_position/2 — Repositionnement d'un flux

Types : set_stream_position(@flux_ou_alias, @position_dans_flux)

Description : Sauf cas d'erreur, l'exécution de `set_stream_position(`$Flux$`,`Pos`)` réussit toujours, avec pour effet le repositionnement du flux identifié par $Flux$, qui est le descripteur ou un alias d'un flux ouvert, à la position déterminée par Pos. En principe, la valeur de Pos provient de la propriété `position(`Pos`)`, acquise par un précédent appel de `stream_property`/2. Il n'y a aucune raison de penser que Pos est d'un type numérique.

Erreurs :

1. $Flux$ est une variable.
2. Pos est une variable.
3. $Flux$ n'est ni une variable, ni un descripteur de flux, ni un alias.
4. Pos n'est ni une variable, ni une position sur un flux.
5. $Flux$ n'est pas associé à un flux ouvert.

6. La valeur de la propriété `reposition` du flux indiqué par $Flux$ est `false`.

Exemples : Soit le fichier `data.in` contenant les lignes suivantes :

```
1001. 1002. 1003. 1004. 1005.
1006. 1007. 1008. 1009. 1010.
1011. 1012. 1013. 1014. 1015.
1016. 1017. 1018. 1019. 1020.
```

```
?- consult.
Consulting ...

job :-
    open("data.in", read, In, [ reposition(true) ]),

    read(In, X1), read(In, X2), read(In, X3),
    write([X1, X2, X3]), nl,

    stream_property(In, position(P)),

    read(In, X4), read(In, X5), read(In, X6),
    write([X4, X5, X6]), nl,

    set_stream_position(In, P),

    read(In, X7), read(In, X8), read(In, X9),
    write([X7, X8, X9]), nl.

end_of_file.
true.
?- job.
[1001,1002,1003]
[1004,1005,1006]
[1004,1005,1006]
true.
```

Voir également : stream_property/2.

stream_property/2 — Propriétés d'un flux

Types : stream_property(?flux,?propriété_de_flux)

Description : Sauf cas d'erreur, l'effacement de `stream_property(`$Flux$`,` $Propriete$`)` produit l'unification de $(Flux, Propriete)$ avec $(un\_flux, une\_propriete)$, où un_flux est le descripteur d'un flux couramment ouvert ayant $une\_propriete$ parmi ses propriétés.

Cet effacement produit la création d'un point de choix, dont les alternatives correspondent aux autres couples $(un\_flux, une\_propriete)$ pour lesquels l'unification indiquée est possible.

Les propriétés des flux sont les suivantes :

`file_name(`F`)` : Lorsqu'un flux est connecté à une source ou un puits qui est un fichier, F représente la chaîne de caractères qui identifie ce fichier pour le système d'exploitation sous-jacent.

`mode(`M`)` : M représente le mode qui a été indiqué lors de l'ouverture du flux. C'est donc l'un des atomes `read`, `write` ou `append`.

`input` : Le flux est connecté à une source, c'est-à-dire un organe d'entrée/sortie qui produit des données.

`output` : Le flux est connecté à un puits, c'est-à-dire un organe d'entrée/sortie qui consomme des données.

`alias(`A`)` : A (un atome) est un des alias du flux.

`position(`P`)` : Si le flux a la propriété reposition(true) alors P représente la position courante dans le flux. P est un terme sans variables, sur la nature duquel on ne doit pas faire d'autres hypothèses.

`end_of_stream(`E`)` : E est l'un des atomes at, past ou not, pour indiquer que le flux se trouve dans la position « sur la fin du flux » (at), sur la position « au-delà de la fin du flux » (past) ou sur une position sans rapport avec la fin du flux (not).

`eof_action(`A`)` : Si une option `eof_action` figurait dans la liste d'options donnée lors de l'appel de `open`/4 qui a ouvert le flux, alors A est l'atome, parmi `error`, `eof_code` ou `reset`, qui a été spécifié à cette occasion. Sinon, A est l'atome qui traduit l'action par défaut.

`reposition(`B`)` : B est `true` si le repositionnement est autorisé sur ce flux, `false` autrement.

`type(`T`)` : T est l'atome, parmi `text` ou `binary`, correspondant au type du flux.

Erreurs :

1. $Flux$ n'est pas une variable et ne représente pas le descripteur d'un flux.
2. $Propriete$ n'est pas une variable et ne représente pas une propriété de flux.

Exemples :

```
?- stream_property(S, alias(user_input)).
S = 6780088.
?- stream_property(S, alias(user_input)), stream_property(S, P).
P = type(text),
S = 6780088;
P = reposition(false),
S = 6780088;
P = eof_action(error),
S = 6780088;
P = end_of_stream(not),
S = 6780088;
P = alias(user_input),
S = 6780088;
P = input,
S = 6780088;
P = mode(read),
S = 6780088;
P = file_name(stdin),
S = 6780088.
```

Voir également : at_end_of_stream/1, open/4.

sub_atom/5 — Découpage d'atome

Types : sub_atom(+atome,?entier,?entier,?entier,?atome)

Description : `sub_atom(A1,I1,I2,I3,A2)` réussit si l'atome `A1` peut être découpé en trois parties `Ag, A2, Ad` de telle sorte que `I1` est le nombre de caractères de `Ag`, `I2` le nombre de caractères de `A2` et `I3` le nombre de caractères de `Ad`. Ce prédicat est non-déterministe et génère tous les découpages possibles.

Erreurs :

1. Si `A1` n'est ni une variable ni un atome,
2. Si `A2` n'est ni une variable ni un atome,
3. Si `I1` n'est ni une variable ni un entier,
4. Si `I2` n'est ni une variable ni un entier,
5. Si `I3` n'est ni une variable ni un entier,
6. Si `I1` est un entier négatif
7. Si `I2` est un entier négatif
8. Si `I3` est un entier négatif

Exemples :

```
?- sub_atom(abracadabra, 2, 4, N, S).
S = raca,
N = 5.
?- sub_atom(abracadabra, X, _, _, aca).
X = 3.
?- sub_atom(abracadabra, X, _, Z, bra).
Z = 7,
X = 1;
Z = 0,
X = 8.
?- sub_atom(abracadra, _, 5, _, S).
S = abrac;
S = braca;
S = racad;
S = acadr;
S = cadra.
?- sub_atom(X, 1, _, 1, abcd).
error: error(instantiation_error,sub_atom/5)
```

Voir également : atom_concat/3, atom_length/2.

throw/1 — Gestion d'erreur

Types : throw(+terme)

Description : `throw(T)` génère une erreur et lance une « balle » à un ancêtre de type `catch/3`. Plus précisément, l'exécution recherche dans l'arbre de recherche l'ancêtre le plus proche de type `catch(T1,T2,T3)` dont l'argument `T1` est toujours en cours d'exécution et tel que l'équation `T = T2` soit satisfaisable (tel que `T` et `T2` soient unifiables).

- Si un tel ancêtre n'existe pas une erreur est générée,

– sinon :

– tous les nœuds entre le nœud courant et le nœud ancêtre sont rendus déterministes (les points de choix sont éliminés),

– on remonte au nœud ancêtre, on ajoute l'équation `T` = `T2` au système de contraintes courant (on unifie `T` et `T2`) et on exécute l'appel à `T3`.

Erreurs :

1. `T` est une variable,
   `instantiation_error`
2. Aucune contrainte de type `T` = `T2`, où `T2` est le second argument d'un appel actif de la forme `catch(T1,T2,T3)` n'est satisfaisable (`T` et `T2` ne sont pas «unifiables»).
   `type_error(callable)`

Exemples :

```
?- catch(throw(mess(1)), mess(X), (write("erreur "), write(X), nl)).
erreur 1
X = 1.
?- consult.
Consulting ...

job(R) :- catch( call(R), local(N),
   ( write("erreur locale interceptee "), write(N), nl, false )).

end_of_file.
true.
?- job((write(debut), nl, throw(local(1)), write(fin))).
debut
erreur locale interceptee 1
false.
?- job((write(debut), nl, throw(non_local(1)), write(fin))).
debut
error: non_local(1)
```

Voir également : catch/3.

true/0 — Vrai

Types : true

Description : `true` réussit toujours.

Exemples :

```
?- true.
true.
```

Voir également : fail/0.

unify_with_occurs_check/2 — Egalité avec test d'occurence

Types : unify_with_occurs_check(?terme,?terme)

Description : `unify_with_occurs_check(T1,T2)` se comporte de la même façon que `=/2`, mais échoue si un arbre infini est construit pendant l'unification.

Exemples :

```
?- X = [1,2,3,X].
X = [1,2,3,X].
?- unify_with_occurs_check(X, f(X)).
false.
?- unify_with_occurs_check(X, [X]).
false.
?- unify_with_occurs_check(X,[1,2,3,X]).
X = [1,2,3,X].
?- X=[1,2,3,X], unify_with_occurs_check(X,Y).
X = [1,2,3,X],
Y = [1,2,3,Y].
```

Voir également : =/2, eq/2, ==/2, \=/2, dif/2.

var/1 — Test de variable

Types : var(?terme)

Description : réussit si le terme T est réduit à une variable. Echoue dans le cas inverse. Plus précisément, `var((t)` réussit si l'étiquette du nœud initial du terme T est inconnue.

Exemples :

```
?- var(X).
X ~ tree.
?- var(_).
true.
?- X=0, var(X).
false.
?- var(X), X=0.
X = 0.
?- cc(X,1,2), var(X).
X ~ cc(1,2).
```

Voir également : nonvar/1.

write_term/3, write_term/2
write/1, write/2
writeq/1, writeq/2
write_canonical/1, write_canonical/2 _ Ecriture d'un terme

Types : write_term(@flux_ou_alias,?terme, +options_pour_write)
write_term(?terme, +options_pour_write)
write(?terme)
write(@flux_ou_alias,?terme)
writeq(?terme)
writeq(@flux_ou_alias,?terme)
write_canonical(?terme)
write_canonical(@flux_ou_alias,?terme)

Description : Sauf cas d'erreur, l'exécution de `write_term(`$Flux$`,`$Terme$`,`$Options$`)` affiche le terme $Terme$ dans le flux de sortie, en tenant compte des options données dans la liste $Options$, puis réussit.

Quand le flux n'est pas précisé en argument, la sortie est le flux courant.

L'argument $Options$ est une liste dont les éléments sont parmi les suivants :

`ignore_ops(`*bool*`)` : *bool* étant `true` ou `false`. Si la valeur de l'option est `true`, tout terme composé est affiché en utilisant exclusivement la notation fonctionnelle. Ni la notation avec opérateurs ni la notation des listes ne sont employées.

`numbervars(`*bool*`)` : *bool* étant `true` ou `false`. Si la valeur de l'option est `true`, tout (sous-)terme de la forme `'$VAR'(`N`)`, N étant un entier, est affiché comme un nom de variable de la forme *lettre majuscule* suivie d'un entier. La lettre est la $(i+1)$ième lettre de l'alphabet, et l'entier est j tels que
`i = N mod 26` et `j = N / 26`. L'entier j est omis s'il vaut zéro.
Par exemple :
`'$VAR'(0)` s'affiche `A`
`'$VAR'(1)` s'affiche `B`
`...`
`'$VAR'(25)` s'affiche `Z`
`'$VAR'(26)` s'affiche `A1`
`'$VAR'(27)` s'affiche `B1`
`...`

`quoted(`*bool*`)` : *bool* étant `true` ou `false`. Quand la valeur de l'option est `true`, tout atome ou foncteur qui doit être affiché est quoté s'il lui est indispensable de l'être pour être relu par les primitives `read`.

Quand des options contradictoires figurent dans la liste, c'est celle la plus à droite qui prévaut.

Les primitives de la famille `write` s'écrivent à l'aide de `write_term` et divers jeux d'options. Voici les équivalences :

write(T) :- write_term(T, [numbervars(true)]).

```
write(S,T) :- write_term(S,T, [numbervars(true)]).
writeq(T) :- write_term(T, [quoted(true),numbervars(true)]).
writeq(S,T) :- write_term(S,T, [quoted(true),numbervars(true)]).
write_canonical(T) :- write_term(T, [quoted(true),ignore_ops(true)]).
write_canonical(S,T) :- write_term(S,T, [quoted(true),ignore_ops(true)]).
write_term(T,L) :- current_output(S), write_term(S,T, L).
```

Voici une description plus fonctionnelle des variantes de `write_term` :

write écrit sans introduire d'apostrophes.

writeq écrit un terme qui pourra être relu par `read`.

write_canonical écrit un terme de façon très basique, sans employer les notations de liste ni les opérateurs.

Note : Le foncteur associé à la paire-pointée (les listes en sont) est le point `'.'` .

Erreurs :

1. $Flux$ est une variable.
2. $Options$ est une variable.
3. $Options$ est une liste dont un élément est une variable.
4. $Options$ n'est ni une variable ni une liste.
5. $Flux$ n'est ni une variable, ni un descripteur, ni un alias d'un flux.
6. Un élément de la liste $Options$ n'est ni une variable ni une option d'écriture valide.
7. $Flux$ n'est pas associé à un flux ouvert.
8. $Flux$ est associé à un flux d'entrée.
9. $Flux$ est associé à un flux binaire.
10. Le flux d'entrée courant est un flux binaire.

Exemples : Les exemples suivants réussissent et affichent les caractères dans la sortie courante :

Requête	Sortie
write([1,2,3])	`[1,2,3]`
write_canonical([1,2,3])	`.(1,.(2,.(3,[])))`
write_term('1<2')	`1<2`
writeq('1<2')	`'1<2'`
writeq('$VAR'(0))	`A`
write_term('$VAR'(1), [numbervars(false)])	`$VAR(1)`
write_term('$VAR'(51), [numbervars(true)])	`Z1`

Foncteurs évaluables	Description	Opération
`'+'/2`	addition	add_I, add_F, add_{FI}, add_{IF}
`'-'/2`	soustraction	sub_I, sub_F, sub_{FI}, sub_{IF}
`'*'/2`	multiplication	mul_I,mul_F, mul_{FI}, mul_{IF}
`'//'/2`	divison entière	$intdiv_I$, div_I, div_F, div_{FI}, div_{IF}
`rem/2`	reste de la division entière	rem_I
`mod/2`	modulo	mod_I
`'-'/1`	moins unaire	neg_I, neg_F
`abs/1`	valeur abolue	abs_I, abs_F
`sqrt/1`	racine carrée	$sqrt_I$, $sqrt_F$
`sign/1`	signe	$sign_I$, $sign_F$
`float/1`	conversion vers un `float`	$float_{I\to F}$
`float_fractional_part/1`	partie décimale d'un `float`	$fractpart_F$
`float_integer_part/1`	partie entière d'un `float`	$intpart_F$
`floor/1`	plus grand des entiers inférieurs	$floor_{F\to I}$
`truncate/1`	partie entière	$truncate_{F\to I}$
`round/1`	entier le plus proche	$round_{F\to I}$
`ceiling/1`	plus petit des entiers supérieurs	$ceiling_{F\to I}$
`'**'/2`	puissance	$exponent_{II}$, $exponent_{IF}$, $exponent_{FI}$, $exponent_{FF}$
`log/1`	logarithme (en base e)	log_I, log_F
`exp/1`	exponentielle	exp_I, exp_F
`sin/1`	sinus	cos_I, cos_F
`cos/1`	cosinus	sin_I, sin_F
`atan/1`	arc tangente	$atan_I$, $atan_F$

TAB. 5.1 – *Les foncteurs arithmétiques*

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. (1.71e98) 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*$(t_1 \ , \ \ldots \ , \ t_n)$ 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 $[t_1 \ , \ \ldots \ , \ t_n]$ est une EBF appelée liste.
- Si t_1,..., t_n, t_{n+1} avec ($n \geq 1$) sont des EBF, alors $[t_1 \ , \ \ldots \ , \ t_n \ | \ t_{n+1}]$ 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.

7

La syntaxe complète de Prolog ISO

CETTE SECTION décrit de manière précise la syntaxe des termes, et donc des textes [1] et des données Prolog.

Les termes sont les structures de données manipulées durant l'exécution par les applications Prolog. La sous-section 2 définit comment les termes forment les textes et les données Prolog. La sous-section 3 définit comment les unités lexicales doivent être combinées pour former les termes. La sous-section 4 définit comment les suites de caractères constituent les unités lexicales.

7.1 Notations

La syntaxe de Prolog est décrite à l'aide d'une grammaire BNF obéissant aux conventions suivantes.

Les méta-opérateurs '=', ' | ' et ' , ' sont associatifs à droite. Le méta-opérateur ' , ' est plus «fort» que ' | ', qui est lui-même plus fort que '='. Ainsi, une règle écrite comme ceci :

$$\alpha \ = \ \beta \ , \ \gamma \ | \ \delta$$

se lit comme ceci :

$$(\ \alpha \ = \ (\ (\ \beta \ , \ \gamma \ ) \ | \ \delta \ ) \ )$$

Les symboles de la grammaire sont définis de la manière suivante :

- = Métasymbole fondamental de réécriture. Il sépare le membre gauche du membre droit de chaque règle.
- ; Un point-virgule termine chaque règle de grammaire.
- , La virgule est le méta-opérateur de concaténation : α , β représente la concaténation de chaînes respectivement définies par les suites de symboles α et β.

1. La notion de programme n'est pas la même en prolog et dans la plupart des autres langages. A la place du mot programme nous emploierons donc l'expression «texte prologIV».

`|` La barre verticale est le méta-opérateur d'alternative : α | β représente une alternative dont les termes sont deux chaînes respectivement définies par les suites de symboles α et β.

`(...)` Les parenthèses servent à regrouper les symboles, afin de donner à une règle une structure différente de celle que détermineraient, sans parenthèses, les méta-opérateurs qui y figurent.

`[...]` Les éléments encadrés par des crochets sont optionnels.

`{...}` Les éléments encadrés par des accolades peuvent apparaître un nombre quelconque de fois.

`"..."` Une suite de caractères encadrée par des guillemets constitue un symbole terminal de la grammaire.

`'...'` Une suite de caractères encadrée par des apostrophes constitue également un symbole terminal de la grammaire.

`(*...*)` Un texte compris entre '`(*`' et '`*)`' est tenu pour un commentaire et ne fait pas partie de la grammaire.

caractères

Une suite de caractères qui n'est pas encadrée par des guillemets ou des apostrophes et ne contient aucun des méta-symboles précédents constitue un symbole non terminal.

Sous chaque règle de grammaire peuvent apparaître diverses autres lignes qui spécifient des attributs de la règle. Chacune de ces lignes est introduite par une étiquette caractéristique :

Abstr. Cette ligne montre les termes abstraits associés aux symboles de la grammaire.

Prior. Cette ligne exprime la priorité des termes.

A chaque terme et chaque opérateur est associé une priorité, qui est un entier r vérifiant $0 \leq r \leq 1201$.

Un terme atomique et un terme composé en notation fonctionnelle ont une priorité égale à zéro.

Un terme composé exprimé en notation opérationnelle (i.e. son foncteur principal apparaît comme un opérateur) a une priorité supérieure ou égale à celle de son foncteur principal.

Cond. Cette ligne exprime des conditions qui doivent être satisfaites pour que la règle de grammaire s'applique.

Spécif. Cette ligne donne les spécificateurs des opérateurs (voyez la table 7.1) intervenant dans la règle.

7.2 Textes et données Prolog

Un texte Prolog est une suite de termes lisibles[2] qui expriment des directives ou des clauses.

2. *Terme lisible* est une tentative de traduction de *read term*

7.2.1 Texte Prolog

	`texte prolog =`	`directive,`	`texte prolog;`
Abstr.	$d \cdot t$	d	t
	`texte prolog =`	`clause,`	`texte prolog;`
Abstr.	$d \cdot t$	d	t
	`texte prolog = ;`		
Abstr.	*nil*		

Directives

	`directive =`	`terme directive , fin;`
Abstr.	d	d
	`terme directive =`	`terme;`
Abstr.	$:-(d)$	$:-(d)$
Prior.		1201

Une directive figurant dans un texte Prolog spécifie :

- des propriétés des procédures définies dans le texte Prolog,
- le format et la syntaxe abstraite de termes lisibles du texte Prolog,
- un but qui devra être exécuté lorsque le texte Prolog aura été préparé pour l'exécution,
- un autre texte Prolog devant être préparé pour l'exécution.

Clauses

	`clause =`	`terme clause , fin;`
Abstr.	c	c
	`terme clause =`	`terme;`
Abstr.	c	c
Prior.		1201
Cond.	Le foncteur principal de c n'est pas `':-'/1`.	

Une clause figurant dans un texte Prolog spécifie une clause d'une procédure définie par l'utilisateur qui doit être ajoutée à la base de données.

Les caractères de la clause supportent les mêmes contraintes que ceux d'un terme lisible dans une exécution réussie du prédicat prédéfini `read/1`.

La clause supporte les mêmes contraintes que $terme$ dans une exécution réussie du prédicat prédéfini `assertz`$(terme)$, sauf une : il n'y aura pas d'erreur si la clause correspond à une procédure statique[3].

Le P/N[4] de la clause ne doit pas être celui d'un prédicat prédéfini ni d'une

3. Alors que c'est une erreur que d'utiliser `assertz/1` ou `asserta/1` pour tenter d'ajouter des clauses à des procédures atatiques.

4. Par «le P/N» nous désignons le couple constitué du prédicat P et de l'arité N du terme composé qui est le membre gauche de la clause.

structure de contrôle.

Toutes les clauses d'une procédure définie par l'utilisateur doivent être des termes lisibles consécutifs d'un même texte Prolog, sauf si la directive `discontiguous` a été utilisée pour indiquer le contraire.

Toutes les clauses d'une procédure définie par l'utilisateur doivent être des termes lisibles se trouvant dans un même texte Prolog, sauf si la directive `multifile` a été utilisée pour indiquer le contraire.

Si aucune clause n'a été spécifiée pour une procédure mentionnée dans une directive `dynamic/1`, `multifile/1` ou `discontiguous/1`, alors la procédure existe mais elle n'a pas de clause.

7.2.2 Données Prolog

Un terme lisible Prolog peut être lu comme une donnée par un appel du prédicat prédéfini `read_term/3`.

	`terme lisible =`	`terme , fin ;`
Abstr.	a	a
Prior.		1201

7.3 Termes

Un terme Prolog est soit un terme atomique, soit une variable, soit un terme composé.

7.3.1 Termes atomiques

Nombres

	`terme =`	`entier ;`
Abstr.	n	n
Prior.	0	

	`terme =`	`nombre flottant ;`
Abstr.	r	r
Prior.	0	

Nombres négatifs

	`terme =`	`nom , entier ;`
Abstr.	n	$-$ n
Prior.	0	

	`terme =`	`nom , nombre flottant ;`
Abstr.	r	$-$ r
Prior.	0	

Atomes

	`terme =`	`atome ;`
Abstr.	a	a
Prior.	0	

Cond. a n'est pas un opérateur.

```
        terme =  atome ;
Abstr.  a        a
Prior.  0
```

Cond. a est un opérateur.

Un atome qui est un opérateur ne peut pas être l'opérande immédiat d'un autre opérateur (cela découle de la priorité maximale donnée à un tel terme).

```
        atome =  nom ;
Abstr.  a        a

        atome =  crochet gauche ,   crochet droit ;
Abstr.  []

        atome =  accolade gauche ,  accolade droit ;
Abstr.  {}
```

Un atome est un nom, ou [] (la liste vide) ou {} (la paire d'accolades vide).

7.3.2 Variables

```
        terme =  variable ;
Abstr.  v        v
Prior.  0
```

7.3.3 Termes composés – notation fonctionnelle

Un terme composé peut toujours être exprimé en notation fonctionnelle. Si le foncteur principal est un opérateur, le terme peut être exprimé aussi en notation opérationnelle. Si le foncteur principal est `'.'/2`, le terme peut aussi être exprimé en notation de liste, et parfois aussi comme une liste à guillemets. Si le foncteur principal est `{}/1`, le terme peut aussi être noté avec une paire d'accolades.

```
    terme =      atome ,   par gauche contr , liste args , par droite ;
Abstr. f(l)      f                           l
Prior. 0

    liste args = argument ;
Abstr. a         a

    liste args = argument , virgule ,        liste args ;
Abstr. a, l      a                           l
```

Arguments

Un argument peut apparaître comme argument d'un terme composé ou comme élément d'une liste. Ce peut être un atome qui est un opérateur ou un terme avec une priorité inférieure à 999. Lorsqu'un argument est un terme quelconque, sa priorité est inférieure à celle de la virgule, si bien qu'il ne peut pas y avoir de conflit entre les occurrences de la virgule en tant qu'opérateur infixe et les occurrences de la virgule en tant que séparateur d'arguments.

```
        argument =  atome ;
Abstr.  a           a
```

Cond. a est un operateur autre qu'une virgule.

TAB. 7.1 – *Spécificateurs des opérateurs*

Spécificateur	Classe	Associativité
`fx`	préfixe	non associatif
`fy`	préfixe	associatif à droite
`xfx`	infixe	non associatif
`xfy`	infixe	associatif à droite
`yfx`	infixe	associatif à gauche
`xf`	postfixe	non associatif
`yf`	postfixe	associatif à gauche

	`argument =`	terme ;
Abstr.	*a*	*a*
Prior.		999

7.3.4 Termes composés – notation opérationnelle

Les termes composés dont le symbole du foncteur est un opérateur défini dans la table 7.4 peuvent être écrits dans une notation opérationnelle.

Un opérateur est défini par son nom, son spécificateur et sa priorité.

La priorité d'un opérateur est un nombre entier r vérifiant $1 \leq r \leq 1200$.

Le spécificateur d'un opérateur est une chaîne de caractères mnémonique qui définit la classe (préfixe, infixe ou postfixe) et l'associativité (associativité à gauche, associativité à droite ou sans associativité) de l'opérateur. La table 7.1 récapitule les différents spécificateurs.

Un opérande avec une priorité égale ou inférieure à celle d'un opérateur associatif à droite, écrit à droite de cet opérateur, n'a pas besoin d'être écrit entre parenthèses. Un opérande avec une priorité inférieure à celle d'un opérateur associatif à gauche, écrit à gauche de cet opérateur, n'a pas besoin d'être écrit entre parenthèses. Un opérande avec une priorité égale à celle d'un opérateur associatif à gauche, écrit à gauche de cet opérateur, doit être écrit entre parenthèses uniquement si le foncteur principal de l'opérande est un opérateur associatif à droite. Un opérande avec la même priorité qu'un opérateur non associatif, écrit à droite ou à gauche de cet opérateur, doit être écrit entre parenthèses. Dans les tables 7.2 et 7.3 on suppose que chaque atome `fx`, `fy`, `xfx`, `xfy`, `yfx`, `xf` et `yf` est un opérateur ayant le spécificateur correspondant. La table 7.2 montre quelques termes invalides et la manière dont ils doivent être parenthésés pour devenir valides.

La table 7.3 montre des termes (valides) non parenthésés en correspondance avec des termes parenthésés équivalents. On suppose ici que les opérateurs `xfy` et `yfx` (resp. `fy` et `yf`) ont même priorité.

Opérande

Un opérande est un terme.

Ci-après, le non-terminal `gterme` représente le sous-ensemble de l'ensemble

TAB. 7.2 – *Termes invalides et termes valides correspondants*

Terme invalide	terme valide
`fx fx 1`	`fx (fx 1)`
`1 xf xf`	`(1 xf) xf`
`1 xfx 2 xfx 3`	`(1 xfx 2) xfx 3`
`1 xfx 2 xfx 3`	`1 xfx (2 xfx 3)`

TAB. 7.3 – *Termes équivalents*

Termes non parenthésés	Termes équivalents parenthésés
`fy fy 1`	`fy (fy 1)`
`1 xfy 2 xfy 3`	`1 xfy (2 xfy 3)`
`1 xfy 2 yfx 3`	`1 xfy (2 yfx 3)`
`fy 2 yf`	`fy (2 yf)`
`1 yf yf`	`(1 yf) yf`
`1 yfx 2 yfx 3`	`(1 yfx 2) yfx 3`

des termes formé des termes qui peuvent être l'opérande gauche d'un opérateur associatif à gauche ayant une priorité donnée.

	`terme =`	`gterme;`
Abstr.	a	a
Prior.	n	n

	`gterme =`	`terme;`
Abstr.	a	a
Prior.	n	$n-1$

A tout endroit où un terme de priorité n est permis peut figurer un terme de priorité inférieure à n.

	`terme = par gauche , terme , par droite;`
Abstr.	a a
Prior.	0 1201

Les parenthèses servent à contourner la priorité des opérateurs.

Opérateurs comme fonctions

	`gterme =`	`terme ,`	`oper ,`	`terme ;`
Abstr.	$f(a,b)$	a	f	b
Prior.	n	$n-1$	n	$n-1$
Spécif.			`xfx`	

	`gterme =`	`gterme ,`	`oper ,`	`terme ;`
Abstr.	$f(a,b)$	a	f	b
Prior.	n	n	n	$n-1$
Spécif.			`yfx`	

	`terme =`	`terme ,`	`oper ,`	`terme ;`
Abstr.	$f(a,b)$	a	f	b
Prior.	n	$n-1$	n	n
Spécif.			`xfy`	

	`gterme =`	`gterme ,`	`oper ,`
Abstr.	$f(a)$	a	f
Prior.	n	n	n
Spécif.			`yf`

	`gterme =`	`terme ,`	`oper ,`
Abstr.	$f(a)$	a	f
Prior.	n	$n-1$	n
Spécif.			`xf`

	`terme =`	`oper ,`	`terme ,`
Abstr.	$f(a)$	f	a
Prior.	n	n	n
Spécif.		`fy`	

	`gterme =`	`oper ,`	`terme ,`
Abstr.	$f(a)$	f	a
Prior.	n	n	$n-1$
Spécif.		`fx`	

Opérateurs

Un opérateur est un nom ou une virgule.

	`oper =`	`nom ;`
Abstr.	a	a
Prior.	n	n
Spécif.	s	s

Cond. a est un opérateur (cf. table 7.4)

	`oper =`	`virgule ;`
Abstr.	,	
Prior.	1000	
Spécif.	`xfy`	

Il ne peut pas exister deux opérateurs avec la même classe (préfixe, infixe ou postfixe) et le même nom.

Il ne peut pas y avoir un opérateur infixe et un opérateur postfixe avec le même nom.

TAB. 7.4 – *La table des opérateurs*

Priorité	Spécificateur	Opérateur(s)
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	:

La table des opérateurs

La table des opérateurs détermine quels atomes doivent être considérés comme des opérateurs lorsque

- une suite d'unités lexicales est analysée comme un terme lisible par le prédicat `read_term/3`, ou bien
- un texte Prolog est préparé pour l'exécution, ou bien
- l'expression écrite d'un terme Prolog est produite par un des prédicats `write_term/...`, `write/...`, etc. La table 7.4 montre les opérateurs prédéfinis, c'est-à-dire ceux qui constituent l'état initial de la table des opérateurs.

Cette table peut être modifiée à l'exécution, grâce au prédicat prédéfini `op/3`. Elle contient les opérateurs associés aux :

1. Prédicats définis comme des opérateurs :
   - unification de termes `'='/2`, `'\='/2`
   - comparaison de termes `'=='/2`, `'\='/2`, `'@<'/2`, `'@=<'/2`, `'@>'/2`, `'@>='/2`,
   - décomposition de termes (univ) `'=..'/2`,
   - évaluation arithmétique `is/2`,
   - comparaison arithmétique `'=:='/2`, `'=\='/2`, `'<'/2`, `'=<'/2`, `'>'/2`, `'>='/2`,
   - négation par échec `'\+'/1`
2. Structures de contrôle définies comme des opérateurs :
   - conjonction `','/2`,
   - disjonction `';'/2`,

- si-alors `'->'/2`

3. Fonctions évaluables définies comme des opérateurs :
   - les opérateurs arithmétiques binaires `'+'/2`, `'-'/2`, `'*'/2`, `'//'/2`, `'/'/2`, `rem/2`, `mod/2`, `'**'/2`,
   - les opérateurs arithmétiques unaires`'-'/1`,
   - les fonctions logiques `'>>'/2`, `'<<'/2`, `'/\'/2`, `'\/'/2`, `'\'/1`

7.3.5 Termes composés - notation de liste

Une liste est soit la liste vide, soit un terme composé dont le foncteur principal est `'.'/2`.

	terme =	crochet gauche ,	éléments ,	crochet droit ;
Abstr.	l		l	
Prior.	0			
	éléments =	argument,	virgule,	éléments ;
Abstr.	$.(h, l)$	h		l
	éléments =	argument,	sépar tête queue,	éléments ;
Abstr.	$.(h, t)$	h		t
	éléments =	argument ;		
Abstr.	$.(t, [])$	t		

Exemples

Voici des termes exprimés en notation de liste et en notation fonctionnelle

```
[a] == .(a, []).

[a, b] == .(a, .(b, [])).

[a | b] == .(a, b).
```

7.3.6 Termes composés - notation avec une paire d'accolades

	terme =	accolade gauche ,	terme ,	accolade droite ;
Abstr.	(l)		l	
Prior.	0	1201		

Exemples

Voici des termes exprimés en notation avec une paire d'accolades et en notation fonctionnelle

```
{a} == '{}'(a).
{a, b} == '{}'(','(a, b))
```

7.3.7 Termes composés - notation en liste à guillemets

Une liste à guillemets est soit un atome, soit une liste non vide.

Si l'indicateur `double_quotes` a la valeur `chars`, une unité lexicale de type liste à guillemets ldq contenant L caractères est une liste l de L éléments dont le nième est le caractère représenté par le nième caractère de ldq.

Si l'indicateur `double_quotes` a la valeur `codes`, une unité lexicale de type liste à guillemets ldq contenant L caractères est une liste l de L éléments dont le nième est le nombre entier associé au nième caractère de ldq.

Si l'indicateur `double_quotes` a la valeur `atoms`, une unité lexicale de type liste à guillemets ldq contenant L caractères est un atome l formé par la concaténation d'une suite de L caractères dont le nième est le caractère représenté par le nième caractère de ldq.

```
        terme =  liste à guillemets ;
 Abstr. l        ccl
 Prior. 0
```

Exemples

```
( current_prolog_flag(double_quotes, chars),
  atom_chars('bonjour', "bonjour") ;
  current_prolog_flag(double_quotes, codes),
  atom_codes('bonjour', "bonjour") ;
  current_prolog_flag(double_quotes, atom),
  'bonjour' == "bonjour" ).
Succès.
( current_prolog_flag(double_quotes, chars),
  [] == "" ;
  current_prolog_flag(double_quotes, codes),
  [] == "" ;
  current_prolog_flag(double_quotes, atom),
   » == "" ).
Succès.
```

7.4 Unités lexicales

```
unité lexicale =
     nom |
     variable |
     entier |
     nombre flottant |
     liste à guillemets |
     par gauche |
     par gauche contr |
     par droite |
     crochet gauche |
     crochet droit |
     accolade gauche |
     accolade droite |
     sépar tête queue |
     virgule ;

nom =
     [ suite de textes décoratifs ] ,
     unité nom ;

variable =
     [ suite de textes décoratifs ] ,
     unité variable ;

entier =
```

```
     [ suite de textes décoratifs ] ,
     unité entier ;

nombre flottant =
     [ suite de textes décoratifs ] ,
     unité nombre flottant ;

liste à guillemets =
     [ suite de textes décoratifs ] ,
     unité liste à guillemets ;

par gauche =
     [ suite de textes décoratifs ] ,
     unité par gauche ;

par gauche contr =
     unité par gauche ;

par droite =
     [ suite de textes décoratifs ] ,
     unité par droite ;

crochet gauche =
     [ suite de textes décoratifs ] ,
     unité crochet gauche ;

crochet droit =
     [ suite de textes décoratifs ] ,
     unité crochet droit ;

accolade gauche =
     [ suite de textes décoratifs ] ,
     unité accolade gauche ;

accolade droite =
     [ suite de textes décoratifs ] ,
     unité accolade droite ;

sépar tête queue =
     [ suite de textes décoratifs ] ,
     unité sépar tête queue ;

virgule =
     [ suite de textes décoratifs ] ,
     unité virgule ;

fin =
     [ suite de textes décoratifs ] ,
     unité fin ;
```

7.4.1 Textes décoratifs

Les textes décoratifs séparent les unités lexicales et résolvent deux ambiguïtés :

- un point . est-il un caractère graphique ou un terminateur ?
- lorsqu'il est suivi d'une parenthèse ouverte, un atome est-il le foncteur d'un terme composé ou bien un opérateur préfixe ?

```
suite de textes décoratifs =
     texte décoratif ,
      texte décoratif ;

texte décoratif =
     caractére décoratif
     | commentaire;
```

Un commentaire de fin de ligne ne doit pas contenir le caractère « nouvelle ligne». Un commentaire avec délimiteurs ne doit pas contenir le délimiteur de fin de commentaire.

```
commentaire =
     commentaire de fin de ligne
     | commentaire avec délimiteurs;

commentaire de fin de ligne =
     début de commentaire de fin de ligne ,
     texte du commentaire ,
     caractère de fin de ligne;

commentaire avec délimiteurs =
     début de commentaire ,
     texte du commentaire ,
     fin de commentaire;

texte du commentaire = { caractère };

début de commentaire =
     limite commentaire 1 , limite commentaire 2;

fin de commentaire =
     limite commentaire 2 , limite commentaire 1;

limite commentaire 1 = "/";

limite commentaire 2 = "*";
```

7.4.2 Noms

```
unité nom =
     unité à lettres et chiffres |
     unité graphique |
     unité à apostrophes |
     unité point-virgule |
     unité coupure;

unité à lettres et chiffres =
     caractère lettre minuscule ,
     { caractère alphanumérique };
```

Une unité graphique ne doit pas commencer par la suite de caractères `début de commentaire`.

Une unité graphique ne doit pas être le caractère . (point) suivi d'un caractère décoratif.

```
unité graphique =
     caractère d'unité graphique ,
     { caractère d'unité graphique } ;

caractère d'unité graphique =
     caractère graphique |
     caractère backslash ;
```

Une unité à apostrophes est faite d'une suite de caractères encadrée par des apostrophes. Si la suite, sans les apostrophes, constitue un atome valide, alors l'unité désigne ce même atome.

Une unité à apostrophes qui ne contient aucun caractère (c'est-à-dire qui est réduite aux apostrophes) est l'atome nul.

Une unité à apostrophes peut être découpée sur plusieurs lignes à l'aide d'une séquence de continuation. Une unité à apostrophes contenant des séquences de continuation est équivalente à l'unité obtenue en supprimant les séquences de continuation.

```
unité à apostrophes =
     caractère apostrophe ,
     { élément d'unité à apostrophes }
     caractère apostostrophes |

séquence de continuation =
     caractère d'unité à apostrophes |
     séquence de continuation ;

séquence de continuation =
     caractère backslash ,
     caractère de fin de ligne ;

unité point-virgule = caractère point-virgule ;

unité coupure = caractère coupure ;
```

Ainsi, `'abc'` et `abc` désignent la même unité lexicale, mais `'\\/'` et `\\/` non, car dans une unité à apostrophes, `\` indique le début d'une séquence d'échappement.

Caractères d'une unité à apostrophes

```
caractère d'unité à apostrophes =
     caractère hors quote |
     caractère apostrophe , caractère apostrophe |
     caractère guillemet |
     caractère back quote ;

caractère d'unité à guillemets =
     caractère hors quote |
     caractère apostrophe |
     caractère guillemet , caractère guillemet |
```

```
     caractère back quote;

caractère d'unité à back quote =
     caractère hors quote |
     caractère apostrophe |
     caractère guillemet |
     caractère back quote , caractère back quote;

caractère hors quote =
     caractère graphique |
     caractère alphanumérique |
     caractère solitaire |
     caractère espace |
     méta-séquence d'échappement |
     séquence d'échappement de contrôle |
     séquence d'échappement octale |
     séquence d'échappement hexadécimale;
```

Il y a trois sortes d'unités lexicales « quotées » : les unités encadrées par des apostrophes, celles encadrées par des guillemets et celles encadrées par des « back quotes ». Dans les trois cas, lorsque le caractère d'encadrement doit figurer dans l'unité, il doit être doublé.

Dans une unité encadrée, un caractère graphique, alphanumérique, solitaire ou un espace, représente ce caractère lui-même.

Une méta-séquence d'échappement représente le caractère introduit par la méta-séquence.

```
méta-séquence d'échappement =
     caractère backslash ,
     méta-caractère;
```

Une séquence d'échappement de contrôle représente le caractère de contrôle indiqué par le nom du caractère de contrôle symbolique si et seulement si ce caractère appartient à l'ensemble de caractères du processeur.

```
séquence d'échappement de contrôle =
     caractère backslash ,
     caractère de contrôle symbolique;

caractère de contrôle symbolique =
     caractère symbolique alert |
     caractère symbolique backspace |
     caractère symbolique carriage return |
     caractère symbolique form feed |
     caractère symbolique horizontal tab |
     caractère symbolique new line |
     caractère symbolique vertical tab;

caractère symbolique alert = "a";

caractère symbolique backspace = "b";

caractère symbolique carriage return = "r";
```

```
caractère symbolique form feed = "f";

caractère symbolique horizontal tab = "t";

caractère symbolique new line = "n";

caractère symbolique vertical tab = "v";
```

Une séquence d'échappement octale ou hexadécimale représente le caractère, appartenant à l'ensemble de caractères du processeur, dont le code interne correspond au nombre donné par la séquence octale ou hexadécimale.

```
séquence d'échappement octale =
     caractère backslash ,
     chiffre octal ,
     { chiffre octal } ,
     caractère backslash;

séquence d'échappement hexadécimale =
     caractère backslash ,
     caractère symbolique hexadécimal ,
     chiffre octal ,
     { chiffre octal } ,
     caractère backslash;

caractère symbolique hexadécimal = "x";
```

7.4.3 Variables

```
unité variable =
     variable anonyme |
     variable nommée;

variable anonyme =
     caractère indicateur de variable;

variable nommée =
     caractère indicateur de variable ,
     caractère alphanumérique ,
     { caractère alphanumérique } |
     caractère lettre majuscule ,
     { caractère alphanumérique };

caractère indicateur de variable =
     caractère blanc souligné;
```

7.4.4 Nombres entiers

```
entier =
     constante entière |
     constante code caractère |
     constante binaire |
     constante octale |
     constante hexadécimale;

constante entière =
```

```
     chiffre décimal ,
     { chiffre décimal };

constante code caractère = "0" ,
     caractère apostrophe ,
     caractère d'unité à apostrophes;
```

Une constante code caractère représente la valeur du code interne du caractère.

```
constante binaire =
     indicateur de constante binaire ,
     chiffre binaire ,
     { chiffre binaire };

indicateur de constante binaire = "0b";

constante octale =
     indicateur de constante octale ,
     chiffre octal ,
     { chiffre octal };

indicateur de constante octale = "0o";

constante hexadécimale =
     indicateur de constante hexadécimale ,
     chiffre hexadécimal ,
      chiffre hexadécimal ;

indicateur de constante hexadécimale = "0x";
```

Les constantes entières sont sans signe. Les entiers négatifs sont définis par la syntaxe des termes.

7.4.5 Nombres à virgule flottante

```
nombre flottant =
     constante entière ,
     fraction ,
     [ exposant ];

fraction =
     caractère point décimal ,
     chiffre décimal ,
      chiffre décimal ;

exposant =
     caractère exposant ,
     signe ,
     constante entière;

signe =
     caractère signe moins |
     [ caractère signe plus ];

caractère signe plus = "+";

caractère signe moins = "-";
```

```
caractère point décimal = "." ;

caractère exposant = "e" | "E" ;
```

Une constante flottante est sans signe. Les flottants négatifs sont définis par la syntaxe des termes.

7.4.6 Listes à guillemets

Une unité lexicale de type liste encadrée par des guillemets représente un terme qui dépend de la valeur de l'indicateur `double_quotes` au moment où le terme ou le texte Prolog contenant l'unité en question est lu.

Une liste à guillemets peut être découpée sur plusieurs lignes à l'aide d'une séquence de continuation. Une liste à guillemets contenant des séquences de continuation est équivalente à l'unité obtenue en supprimant les séquences de continuation.

```
liste à guillemets =
     caractère guillemet ,
     élément d'unité à guillemets  ,
     caractère guillemet ;

élément d'unité à guillemets =
     caractère d'unité à guillemets |
     séquence de continuation ;
```

7.4.7 Chaînes à back quotes

Une chaîne à back quotes est une suite de caractères encadrés par deux caractères ` (« back quote »). Une chaîne à back quotes peut être découpée sur plusieurs lignes à l'aide d'une séquence de continuation. Une chaîne à back quotes contenant des séquences de continuation est équivalente à la chaîne obtenue en supprimant les séquences de continuation.

```
chaîne à back quotes =
     caractère back quote ,
      élément de chaîne à back quote  ,
     caractère back quote ;

élément de chaîne à back quote =
     caractère de chaîne à back quote |
     séquence de continuation ;
```

La norme ISO définit la syntaxe des chaînes à back quote mais pour le moment elle n'introduit pas d'unité lexicale ni de terme qui obéirait effectivement à cette syntaxe.

7.4.8 Autres unités lexicales

```
unité par gauche = caractère par gauche ;

unité par droite = caractère par droite ;
```

```
unité crochet gauche = caractère crochet gauche;

unité crochet droit = caractère crochet droit;

unité accolade gauche = caractère accolade gauche;

unité accolade droite = caractère accolade droite;

unité sépar tête queue = caractère sépar tête queue;

unité virgule = caractère virgule;

unité fin = caractère fin;

caractère fin = ".";
```

Un caractère de fin ne peut pas être suivi par un caractère autre qu'un caractère décoratif ou un `%`.

7.5 Caractères du processeur

L'ensemble des caractères du processeur dépend de chaque implémentation. Il doit contenir tous les caractères définis par la règle caractère, mais peut inclure des éléments additionnels, appelés caractères étendus. Selon l'implémentation, ces caractères sont graphiques, alphanumériques, solitaires, etc.

```
caractère =
     caractère graphique |
     caractère alphanumérique |
     caractère solitaire |
     caractère décoratif |
     méta-caractère;
```

7.5.1 Caractères graphiques

```
caractère graphique =
     "#" | "$" | "&" | "*" | "+" | "-" | "." |
     "/" | ":" | "<" | "=" | ">" | "?" | "@" |
     "^" | "~";
```

7.5.2 Caractères alphanumériques

```
caractère alphanumérique =
     caractère alphabétique |
     caractère chiffre décimal;

caractère alphabétique =
     caractère blanc souligné |
     caractère lettre;

caractère lettre =
     caractère lettre majuscule |
     caractère lettre minuscule;

caractère lettre minuscule =
```

```
     "a" | "b" | "c" | "d" | "e" | "f" | "g" | "h" |
     "i" | "j" | "k" | "l" | "m" | "n" | "o" | "p" |
     "q" | "r" | "s" | "t" | "u" | "v" | "w" | "x" |
     "y" | "z";

caractère lettre majuscule =
     "A" | "B" | "C" | "D" | "E" | "F" | "G" | "H" |
     "I" | "J" | "K" | "L" | "M" | "N" | "O" | "P" |
     "Q" | "R" | "S" | "T" | "U" | "V" | "W" | "X" |
     "Y" | "Z";

caractère chiffre décimal =
     "0" | "1" | "2" | "3" | "4" |
     "5" | "6" | "7" | "8" | "9" |

caractère chiffre binaire = "0" | "1";

caractère chiffre octal =
     "0" | "1" | "2" | "3" | "4" | "5" | "6" | "7";

caractère chiffre hexadécimal =
     "0" | "1" | "2" | "3" | "4" |
     "5" | "6" | "7" | "8" | "9" |
     ("a" | "A") | ("b" | "B") | ("c" | "C") |
     ("d" | "D") | ("e" | "E") | ("f" | "f");

caractère blanc souligné = "_";
```

7.5.3 Caractères solitaires

```
caractère solitaire =
     caractère de coupure |
     caractère par gauche |
     caractère par droite |
     caractère virgule |
     caractère point-virgule |
     caractère crochet gauche |
     caractère crochet droite |
     caractère accolade gauche |
     caractère accolade droite |
     caractère sépar tête queue |
     caractère commentaire de fin de ligne;
```

Dans une unité quotée (encadrée par des apostrophes, des guillemets, etc.) un caractère solitaire se représente lui même.

Un caractère solitaire qui n'est pas à l'intérieur d'une unité lexicale quotée constitue une unité à lui tout seul (excepté `%` et les caractères restants sur la ligne, qui n'ont pas de signification dans un texte Prolog ou un terme lisible).

Un caractère solitaire n'a pas besoin d'être séparé de ses unités lexicales voisines par des caractères décoratifs.

7.5.4 Caractères décoratifs

```
caractère décoratif =
```

```
        caractère espace |
        caractère tabulation horizontale |
        caractère fin de ligne;

caractère espace = " ";

caractère tabulation horizontale =
        dépend de l'implantation;

caractère fin de ligne =
        dépend de l'implantation;
```

Les caractères décoratifs sont parfois requis pour séparer deux unités lexicales. Un texte décoratif ne constitue jamais une unité lexicale.

7.5.5 Méta-caractères

```
méta-caractère =
        caractère backslash |
        caractère apostrophe |
        caractère guillemet |
        caractère back quote;

caractère backslash = "\";

caractère apostrophe = "'";

caractère guillemet = """;

caractère back quote = "`";
```

7.6 Table des codes des caractères

La table des codes des caractères est définie implicitement par l'association d'un entier à chaque caractère. Cette table dépend de l'implantation, avec les restrictions suivantes :

- les codes des lettres majuscules `A`, `B`, ... `Z` doivent être croissants,
- les codes des lettres minuscules `a`, `b`, ... `z` doivent être croissants,
- les codes des chiffres décimaux `0`, `1`, ... `9` doivent être croissants et contigus.

Les tables ASCII et EBCDIC satisfont ces contraintes.

7.7 Lexique

Voici les traductions utilisées de quelques termes et expressions utilisés par les rédacteurs de la norme ISO.

back quote	back quote
backslash	backslash
built-in predicate	prédicat prédéfini
clause (in Prolog text)	clause (dans un texte Prolog)
close curly	accolade droite
close list	crochet droit
close	par droite
collating sequence	suite des codes internes des caractères
control construct	structure de contrôle
curly brackets	accolades
data (Prolog data)	donnée (donnée Prolog)
database (the Prolog database)	base de données (la base de données Prolog)
directive (in Prolog text)	directive (dans un texte Prolog)
double quoted char	caractère d'unité à guillemets
double quoted list / token	liste / unité à guillemets
flag	indicateur
ht sep	separ tête queue
layout (layout text)	décor (texte décoratif)
new line char	caractère de fin de ligne
open ct	par gauche contr
open curly	accolade gauche
open list	crochet gauche
open	par gauche
operator notation	notation opérationnelle
P/N (clause P/N)	P/N (P/N d'une clause)
predicate indicator	indicateur de prédicat
quote, quoted	quote, quoté
quoted char	caractère d'unité à apostrophes
quoted list / token	liste / unité à apostrophes
read term	terme lisible
single quoted char	caractère d'unité à apostrophes
single quoted list / token	liste / unité à apostrophes
solo char	caractère solitaire
specifier (of an operator)	spécificateur (d'un opérateur)
static procedure	procédure statique
text (Prolog text)	texte (texte Prolog)
underscore	blanc souligné
user-defined procedure	procédure définie par l'utilisateur

8

Environnement

DANS CE CHAPITRE on trouvera la description des options de la ligne de commande, de la compilation, de la mise au point des programmes, et d'autres fonctionnalités qui facilitent la vie du programmeur.

8.1 Les options de lancement

Le lancement depuis un shell de l'exécutable Prolog IV sans son environnement graphique s'effectue en lançant une commande de la forme suivante :

```
% prolog4 [ option ... option] [-- optionsUtilisateur...]
```

dans laquelle chacune des options est décrite plus loin, et où les options-utilisateurs sont des chaînes quelconques. Ce qui est entre crochets (qui ne doivent pas être tapés) est optionnel. C'est le double-tiret qui sépare les options destinées à Prolog IV de celles destinées au programme écrit en Prolog IV.

La totalité des options peut être récupérée par la primitive `argv/1` et la partie «utilisateur» peut être récupérée au moyen de la primitive `user_argv/1`.

8.1.1 Options

La plupart des options de la ligne de commande concernent le paramétrage des piles et espaces de travail utilisés par Prolog IV. D'autres options positionnent des modes de fonctionnement de Prolog IV.

Ces options peuvent tout aussi bien être fournies à Prolog IV lorsqu'il démarre d'un terminal que lancé sous son interface graphique (où ces options peuvent être données au travers d'un dialogue de préférences).

Dans ce qui suit, on appelle cellule un double mot-machine [1].

Voici quelques-unes de ces options, avec entre crochets la valeur par défaut donnée par PrologIA, lorsque cette information est appropriée. Nb est un entier positif à fournir.

1. Une cellule occupe donc huit octets sur une machine 32 bits.

`-help`	Affiche la liste des options disponibles.
`-heap` Nb	Attribue à la pile $heap$ la place pour Nb cellules [700000].
`-trail` Nb	Nb double-cellules pour la pile $trail$ [400000].
`-env` Nb	Nb cellules pour la pile env [50000].
`-choice` Nb	Nb cell. pour la pile $choice$ [50000].
`-global` Nb	Nb cell. pour la zone statique (consult,record,...) [500000].
`-local` Nb	Nb cell. pour la pile locale (unification, solveur de contraintes) [5000].
`-work` Nb	Nb cell. pour la zone temporaire (calculs rationnels, Gauss) [50000].
`-nbpcindex` Nb	Nb maximum de littéraux pour les programmes compilés (Pcode) [10000].
`-union`	Démarre Prolog IV en mode «unions d'intervalle» [intervalles «simples»].
`-iso`	Démarre Prolog IV en mode «iso étendu» [mode «prolog4»].
`-g`	Démarre Prolog IV en mode «débogueur».
`-Qcalculator=off`	Démarre Prolog IV sans utiliser la calculatrice rationnelle [on].

8.1.2 Primitives

argv(L), user_argv(L) récupèrent la liste de paramètres de la ligne de commande. Cette liste se compose d'atomes. `user_argv/1` ne récupère que les arguments situés après le séparateur `--` ; il rend `[]` s'il n'y en a aucun. Par exemple :

```
machine% prolog4 -heap 30000 -- carres 9
...
» argv(L).
L = ['prolog4','-heap','30000',--,carres,'9'].
» user_argv(L).
L = [carres,'9'].
```

8.2 Interruption utilisateur

A tout moment, un programme Prolog IV peut être interrompu au moyen d'une touche ou d'une action déterminée, dépendante du système d'exploitation utilisé (par exemple les touches `CTRL-C` sous Unix, ou le bouton «Stop» dans la fenêtre «Console» de l'environnement graphique de Prolog IV).

L'interruption de programme sert deux desseins :

- Arrêter un programme qui tourne (qu'il boucle ou qu'il dure trop longtemps).
- Passer dans un autre mode d'exécution (en mode débogueur).

Sous Unix, la même touche sert à arrêter un programme et à passer en mode débogueur. Ce qui discrimine est la façon d'utiliser cette touche :

- Si cette touche est pressée *deux fois* en moins d'une seconde, on passe en mode débogueur qui donne la main à l'utilisateur (en affichant le prompt du débogueur).
- Sinon, il s'agit d'une demande d'arrêt de programme : une erreur «`user_interrupt`» est générée par Prolog IV. Si cette erreur n'est pas récupérée par le programme, celui-ci termine et le prompt invitant à taper une nouvelle requête est affiché.

8.3 Compilation des règles

Après un bref rappel sur les primitives d'entrée de règles, il sera donnée une description des directives comprises par celles-ci.

8.3.1 Primitives de compilation

Il existe deux familles de primitives d'entrée de règles :

- Celles qui codent des paquets de règles dynamiques (famille `consult`).
- Celles qui codent des paquets de règles statiques (famille `compile`).

La plupart du temps, on préfèrera la compilation à la consultation : l'exécution des règles compilées est plus rapide, et le débogueur ne peut fonctionner qu'avec les règles compilées.

compile(F**), compile(**F**,** L**)** compilent le fichier de nom F (un atome). Les règles compilées sont statiques. L est une liste d'options portant sur le mode syntaxique et le mode débogueur de cette compilation.

recompile(F**), recompile(**F**,** L**)** fonctionnent comme les primitives `compile/`n, mais sans erreur de redéfinition.

consult(F**), consult** lisent et codent les règles du fichier de nom F (un atome). Les règles consultées sont dynamiques. Sans argument, les règles sont lues dans l'entrée courante.

reconsult(F**), reconsult** fonctionnent comme les primitives `consult/`n, mais sans erreur de redéfinition.

8.3.2 Directives de compilation

Les directives sont des annotations permettant de passer certaines informations au compilateur, d'altérer son comportement, de spécifier les propriétés des paquets de règles présents dans le fichier, d'inclure un fichier etc. Les directives ne sont comprises que du compilateur (primitives `compile`) et non pas des primitives `consult`.

Voici la liste de ces directives :

dynamic/1 Se comporte comme la primitive `dynamic/1`. Le compilateur ne pouvant que compiler, et non pas consulter, la présence dans le fichier d'une règle déclarée dynamique provoquera une erreur pendant la compilation.

op/3 Se comporte comme la primitive `op/3`. La portée des opérateurs déclarée est générale : ils ne sont pas éliminés en fin de compilation.

syntax/1 Informe le compilateur que les paquets de règles doivent être compilés en mode :

- `iso`, pour que les extensions syntaxiques majeures sont déconnectées,
- `prolog4` pour que la syntaxe soit augmentée afin d'alléger les notations décrivant les contraintes.

Par défaut, le mode courant (indiqué par le prompt de la console) sert de mode de compilation du fichier.

include/1 La directive `:-include(`$fichier$`)` indique au compilateur de lire le fichier $fichier$ (un atome) avant de poursuivre la compilation du fichier courant. Tout se passe comme si la directive était remplacée par le contenu du fichier indiqué en argument.

debug/0 Les paquets de règles qui suivent jusqu'à la fin du fichier ou jusqu'à la prochaine directive `:-no_debug` sont compilés d'une manière qui permet au débogueur de suivre l'exécution de ces paquets.

no_debug/0 Désactive la compilation en mode `debug` des paquets de règles qui suivent.

set_prolog_flag/2 Appelle simplement en ce point la primitive `set_prolog_flag/2`.

8.4 Le débogueur Prolog IV

8.4.1 Introduction

Tout programme (même écrit en Prolog IV), peut être faux !
En effet, il n'existe pas de langage de programmation permettant de rentrer en machine «le problème que l'on a en tête» sans laisser la porte ouverte aux erreurs. Bien sûr, le premier précepte à suivre est de relire son code quand il ne fonctionne pas comme on pense qu'il le «devrait». Il faut aussi corriger les éventuels avertissements lancés pendant la compilation ou la vérification syntaxique. Dans bien des circonstances, l'examen détaillé du déroulement d'un programme et le suivi interactif de son exécution peuvent apporter de précieux indices au programmeur laissé perplexe face à son «bug»[2]. Le mode de fonctionnement `debug` de Prolog IV répond à ce besoin de dépistage.

8.4.2 Le modèle des boîtes

Le débogueur de Prolog IV est fondé sur le «modèle des boîtes» que l'on retrouve dans des prologs plus classiques. Le concept important de ce modèle est que *chaque appel peut être considéré comme une boîte*, opaque ou transparente selon le désir de l'utilisateur d'avoir une trace d'exécution plus ou moins détaillée. Les boîtes de ce modèle comprennent quatre ouvertures, appelées ports. Deux autres ports sont disponibles en Prolog IV, et permettent de voir à l'intérieur des boîtes.

2. Bogue.

8.4.3 Les ports

Ces ports correspondent à des endroits précis de l'exécution d'un but. Ils schématisent :

- l'activation d'un paquet de règles (`CALL`),
- la terminaison correcte d'une des règles du paquet (`EXIT`),
- le retour au sein de ce paquet par backtracking (`REDO`),
- la désactivation de ce paquet par épuisement de ses choix (`FAIL`),
- le choix de la n^{ieme} règle d'un paquet (`RULE`).
- l'unification avec succès de la tête de règle avec le but courant (`OK`).

Les deux derniers sont des ports secondaires, internes à la boîte.

Le comportement d'une boîte étant naturellement non-déterministe, une même boîte peut être sollicitée à plusieurs reprises pour fournir différentes réponses. De façon séquentielle, on entre pour la première fois dans une boîte par `CALL`. On en sort avec succès par `EXIT` qui indique qu'une solution a été trouvée. Lorsque des choix ont été laissés en attente, on revient dans la boîte pour demander une solution alternative par `REDO`. L'absence d'une (autre) solution est indiquée par `FAIL`, qui provoque la sortie définitive de la boîte.

L'affichage des ports respecte un même schéma. A partir de l'exemple,

```
4 [2] EXIT (r1) : dessert(glace)
```

voici la description des différents champs de ce schéma :

`4` : Profondeur d'un appel : C'est le numéro d'ordre de la boîte dans laquelle on se trouve avant d'avoir effectué l'action.

`[2]` : Niveau de l'appel : il est défini par le niveau d'imbrication de la boîte correspondant à cet appel. Ce numéro est tel que tous les appels situés dans une même queue de règle (ou dans une requête) ont un même niveau.

`EXIT` : C'est le nom du port, l'une des chaînes `CALL`, `EXIT`, `REDO`, `FAIL`, `RULE` ou `OK`.

`(r1)` : Le numéro de la règle appliquée. Lorsqu'il est suivi d'une étoile, il indique qu'il s'agit de la dernière règle du paquet.

`dessert(glace)` : C'est le littéral qu'on traite. Il est affiché dans son état actuel d'instanciation.

Les ports sont affichés avant d'être franchis.

8.4.4 Sessions de déboguage

On verra dans ce qui suit diverses sessions montrant quelques possibilités du débogueur Prolog IV, en présentant au passage les principales commandes. Dans tout ce qui suit, ce qui doit être tapé par le programmeur est souligné.

Soit le petit programme suivant, qu'on place dans le fichier `dessert.p4` :

```
dessert(fruit, 2) .
dessert(ice_cream,6) .
```

Compilons-le en mode `debug` :

```
>> compile('dessert.p4', [debug(on)]).
true.
```

Maintenant, suivons le déroulement de l'appel au paquet `dessert` ; à chaque fois que le prompt du débogueur s'affiche, nous répondrons par la commande `s` (raccourci de la commande `step` qui permet la progression de port en port).

Pour simplifier la présentation, certains retours chariots ont été omis, ainsi que divers séparateurs de solutions.

```
>> debug.
true.

>> dessert(X,Y).

   4[4]CALL    : dessert(_267, _268) (DBG IV) s
    5[5]RULE(r1): dessert/2(_267, _268) (DBG IV) s
    5[5]OK(r1): dessert/2(fruit, 2) (DBG IV) s
    5[5]EXIT(r1): dessert/2(fruit, 2) (DBG IV) s
Y = 2,
X = fruit
  4[3]REDO(r1): dessert/2(_267, _268) (DBG IV) s
    5[5]RULE(r2*): dessert/2(_267, _268) (DBG IV) s
    5[5]OK(r2*): dessert/2(ice_cream, 6) (DBG IV) s
    5[5]EXIT(r2*): dessert/2(ice_cream, 6) (DBG IV) s
Y = 6,
X = ice_cream
    5[5]FAIL(r2*): dessert/2(_267, _268) (DBG IV) s

>>
```

Voici une explication de la session précédente, ligne par ligne :

- (`CALL`) On appelle `dessert`.
- (`RULE`) La première règle (`r1`) de `dessert` est essayée.
- (`OK`) La tête de cette règle est unifiée avec l'appel.
- (`EXIT`) Tous les littéraux de la queue de règle ont été exécutés (il n'y en avait aucun).
- La solution est affichée, car il n'y a plus de littéraux en attente d'exécution.
- (`REDO`) Il existe un point de choix pour l'appel à `dessert`.
- (`RULE`) On essaie la seconde règle (`r2`) de `dessert`.
- (`OK`) La tête de cette règle est unifiée avec l'appel.
- (`EXIT`) Tous les littéraux de la queue de règle ont été exécutés (il n'y en avait aucun).
- La solution est affichée, car il n'y a plus de littéraux en attente d'exécution.

– (FAIL) Il n'y a plus de choix à essayer pour l'appel à `dessert`.

– Retour au prompt Prolog IV. Tout l'arbre de recherche (un arbre avec des nœuds *et* et des nœuds *ou*) a été exploré.

Pour aller plus loin, utilisons maintenant le fichier `menu.p4` décrit ci-après.

```
repasleger(H,M,D) :-
    horsdoeuvre(H,I),
    plat(M,J),
    dessert(D,K).

horsdoeuvre(radis,1) .
horsdoeuvre(melon,6) .

plat(M,I) :- viande(M,I) .
plat(M,I) :- poisson(M,I) .

dessert(fruit, 2) .
dessert(glace,6) .

viande(poulet,5)  .
viande(porc,7)  .

poisson(sole,2) .
poisson(thon,4) .
```

Montrons avec cette session comment se déplacer de boîte en boîte sans voir le détail des appels.

```
>> compile('menu.p4', [debug(on)]).
true.

>> debug.
true.

>> repasleger(X,Y,Z).

    2[2]CALL    : repasleger(_267, _268, _269) (DBG IV) s
     3[3]RULE(r1*): repasleger/3(_267, _268, _269) (DBG IV) s
     3[3]OK(r1*): repasleger/3(_267, _268, _269) (DBG IV) s
     3[3]CALL    : horsdoeuvre(_267, _1213) (DBG IV) n
     3[3]CALL    : plat(_268, _1214) (DBG IV) s
     3[3]CALL    : dessert(_269, _1215) (DBG IV) s
      7[4]RULE(r1): dessert/2(_269, _1215) (DBG IV)
     ...
```

– Pour chacun des trois premiers ports, on fait du pas à pas (`step`).

– On veut ensuite sauter le détail de l'exécution des appels à `horsdoeuvre`, et à `plat`. On effectue pour ce faire la commande `next` (abrégée par `n`).

– Arrivé sur l'appel à `dessert`, on effectue du pas à pas pour voir le détail.

Montrons avec cette session comment progresser plus rapidement encore, par

le biais de point d'arrêts, toujours sans voir le détail des appels.

```
>> debug.
true.

>> repasleger(X,Y,Z).
    2[2]CALL    : repasleger(_267, _268, _269) (DBG IV) spy dessert

   dbg: spying: (dessert/2).
   (DBG IV) c
   **   3[3]CALL    : dessert(_269, _1215) (DBG IV) s
   **    7[4]RULE(r1): dessert/2(_269, _1215) (DBG IV) s
   **    7[4]OK(r1): dessert/2(fruit, 2) (DBG IV) c
   **    7[4]EXIT(r1): dessert/2(fruit, 2) (DBG IV) c
   Z = fruit,
   Y = poulet,
   X = radis
   **  6[2]REDO(r1): dessert/2(_269, _1215) (DBG IV) s
   **    7[4]RULE(r2*): dessert/2(_269, _1215) s
   **    7[4]OK(r2*): dessert/2(glace, 6)
   ...
```

– On utilise la commande `spy` pour installer un point d'arrêt sur les paquets de règles `dessert`.

– On effectue la commande `cont` (abrégée par `c`) pour continuer l'exécution du programme jusqu'à ce qu'un port faisant intervenir le littéral espionné (`dessert` dans notre cas) soit détecté. La machine s'y arrête alors.

– On peut ensuite au moyen de `step` détailler notre appel.

Les marques `**` qui figurent en marge indiquent que le port en question est un point d'arrêt.

Montrons quelques commandes affichant des informations diverses sur la démonstration et sur les variables :

```
(DBG IV) where
---\/---  NEW  ---\/---
   7[3](r2*) dessert/2(glace, 6)
   3[2](r1*) repasleger/3(radis, poulet, glace)
   2[1](r1*) sysh__query/1(['X'=radis,'Y'=poulet,'Z'=glace])
---/\---  OLD  ---/\---
(DBG IV) chpt
---\/---  NEW  ---\/---
   6[4](r1) viande/2(poulet, 5)
   5[3](r1) plat/2(poulet, 5)
   4[3](r1) horsdoeuvre/2(radis, 1)
---/\---  OLD  ---/\---
(DBG IV) up
(DBG IV) printvar
H = radis
M = poulet
D = glace
I = 1
J = 5
K = 6
(DBG IV)
```

- La commande `where` est utilisée pour montrer où l'on se trouve dans la démonstration courante (l'empilement des appels).
- La commande `chpt` sert à visualiser les points de choix qui restent à traiter.
- La commande `up` nous permet de retourner dans la procédure appelante. Ce changement de contexte nous permet d'interroger le débogueur sur les valeurs des variables de ce contexte-là.
- La commande `printvar`, sans argument, nous montre les valeurs de toutes les variables créées lors de la création de ce contexte.

8.4.5 Options du débogueur

Les options[3] sont des variables internes au débogueur. Elles servent essentiellement de paramètres implicites à certaines commandes.

Dans plusieurs commandes et options, les types de port sont repérés par une lettre. On a *in extenso* c pour CALL, e pour EXIT, r pour REDO, f pour FAIL, R pour RULE et Y pour OK. Une combinaison de ports est un suite de ces lettres, comme par exemple «`ceR`». La combinaison d'aucun port est donné par « { } ».

Les options du débogueur sont `stepports`, `nextports`, `contports`, `spyports`. Voici leur description :

stepports : est utilisée par la commande `step` pour la progression de port en port. Par défaut, cette option contient la chaîne «cerfRY » (tous les ports).

3. Le mot «option» est préféré au mot «variable» pour éviter toute confusion avec les variables prolog.

nextports : est utilisée par la commande `next` pour limiter l'affichage pendant la progression. Par défaut, cette option contient la chaîne vide (aucun port).

contports : est utilisée par la commande `cont` pour limiter l'affichage pendant la progression. Par défaut, cette option contient la chaîne vide (aucun port).

spyports : est utilisée par la commande `spy` pour installer un point d'arrêt sur un ou plusieurs paquet de règles, pour les ports décrits dans cette option. Par défaut, cette option contient la chaîne «cerfRY» (tous les ports).

8.4.6 Commandes

Les commandes de progression permettent d'avancer par pas variables dans un programme. Il faut noter qu'en prolog, suivant le sens du contrôle, le port suivant peut être à une profondeur plus grande ou plus faible que le port courant. Voici la liste de ces commandes : `step`, `next`, `cont`.

La gestion des points d'arrêt : `spy`, `unspy`.

L'affichage d'informations : `printvar`, `subdomain`, `chpt`, `where`, `up`, `down`.

Commandes diverses : `prolog`, `kill`, `help`.

On appelle *contexte* ou encore *contexte d'appel* un point de la démonstration courante. On peut voir un contexte comme une règle appliquée, ses variables étant dans un certain état d'instanciation, selon les appels qui ont déjà été effectués dans la queue de règle. Les contextes sont naturellement chaînés par la structure des appels imbriqués (un seul appel est actif à la fois.) On peut se déplacer parmi les contextes au moyen des commandes `up` et `down`.

step

C'est la commande de base pour suivre la progression pas à pas de la machine prolog. Avance pas à pas de port en port. Seuls les ports dont le type est indiqué par l'option `stepports` sont pris en compte. On peut donc ignorer certains types de ports et avancer plus vite. Son raccourci est `s`.

step *ports*

Positionne l'option `stepports` avec la valeur *ports*, puis effectue la commande `step`. Raccourci : `s`.

next

Avance d'appel en appel en affichant au passage les ports dont le type est indiqué par l'option `nextports`. Seuls les appels de niveaux inférieurs ou égaux au niveau courant sont montrés. Autrement dit, on ne détaille pas les appels, que l'on peut considérer comme autant de boîtes noires. Si on ne peut aller à l'appel suivant (pour cause d'échec prolog par exemple), cette commande se comporte comme `step`. Raccourci : `n`.

next *ports*

Positionne l'option `nextports` avec la valeur *ports*, puis effectue la commande `next`. Raccourci : `n`.

cont

Continue l'exécution en affichant au passage les ports dont le type est indiqué par l'option `contports`. Si une point d'arrêt est détectée, la main est rendue au programmeur. Raccourci : `c`.

cont *ports*

Positionne l'option `contports` avec la valeur *ports*, puis effectue la commande `cont`. Raccourci : `c`.

where

Affiche l'empilement des appels, du plus récent au plus ancien. Raccourci : `w`.

source 1 *ou* 0

Active/désactive le mode «*source*» du débogueur. Ce mode n'a d'intérêt qu'en présence de l'environnement graphique de Prolog IV. Quand celui-ci est présent, le mode `source` est automatiquement positionné (à 1).

pbox 1 *ou* 0

Active/désactive l'affichage des lignes de statut du débogueur, c.a.d. ne montre plus les affichages textuels des ports rencontrés. Désactiver cet affichage (en mettant 0) n'a d'intérêt qu'en présence de l'environnement graphique, quand on ne veut plus se servir que du mode `source`. Par défaut, cet affichage est présent.

kill

Abandonne l'exécution de la requête courante. On revient au prompt Prolog IV. Le mode `debug` reste toutefois actif. Raccourci : `k`.

no_debug

Abandonne le mode `debug`. La prochaine commande tapée permet d'abandonner l'exécution du programme en cours (avec `kill`), ou de la poursuivre (avec `cont` par exemple).

halt

Termine l'exécution de Prolog IV. On revient au système d'exploitation.

up

Déplace le pointeur de contextes dans le contexte supérieur (celui de la règle appelante). Si on est dans le contexte le plus haut, un avertissement est affiché et rien n'est fait. Raccourci : `u`.

down

Déplace le pointeur de contextes dans le contexte inférieur (celui de la règle appelée) Si on est dans le contexte le plus bas, un avertissement est affiché et rien n'est fait. Raccourci : `d`.

printvar

Affiche toutes les variables du contexte courant. En conjonction avec des commandes `up` et `down`, permet d'examiner les variables qui participent à la démonstration courante. Il n'y a pas d'affichage de sous-domaine (voir la commande `subdomain` pour cela). Cette commande est basée sur la primitive Prolog `write`. Raccourci : `p`.

printvar *var* ...

Même chose, mais en se restreignant a certaines variables du contexte courant. Raccourci : `p`.

subdomain

Affiche les sous-domaines de toutes les variables appartenant au contexte courant (ou au contexte positionné par `up` ou `down`). La commande `subdomain` donne des informations sur les variables qui entrent en jeu d'une façon plus précise et plus élégante qu'avec `printvar`. Cette commande est basée sur le système d'affichage de la solution d'une requête. Raccourci : `sd`.

subdomain *var* ...

Même chose, mais en se restreignant a certaines variables du contexte courant. Raccourci : `sd`.

help

Affiche la liste des commandes disponibles, avec une description succincte. Raccourci : `h`.

help *cmd* ...

Affiche une description succincte de la commande *cmd*. Raccourci : `h`.

chpt

Affiche la pile des points de choix, du plus récent au plus ancien.

spy

Met un point d'arrêt sur le paquet dont le nom est celui de l'appel courant, pour les seuls ports contenus dans l'option `spyports`.

spy *nom* ...

Met un point d'arrêt sur le paquet de nom *nom*, pour les seuls ports contenus dans l'option `spyports`. Les noms peuvent avoir la forme *identificateur/arité* ou *identificateur*. Dans ce dernier cas, tous les paquets de noms *identificateur*, quelle que soit leur arité, sont espionnés.

unspy

Enlève un (éventuel) point d'arrêt sur le paquet dont le nom est celui de l'appel courant.

unspy ***nom ...***

Enlève un (éventuel) point d'arrêt sur le paquet de nom *nom*.

spyports ***ports***

Positionne l'option `spyports` (utilisé par la commande `spy`) à la valeur *ports*. Sans argument, affiche la valeur courante de l'option `spyports`.

prolog

Lance une nouvelle session Prolog IV imbriquée : le prompt de Prolog IV est affiché et on peut taper des requêtes (on sort temporairement du mode débogueur). Cette nouvelle session doit être terminée par le but « `stop.` ». On revient alors à la session de déboguage courante (sous le prompt du débogueur). Cette commande permet d'effectuer des vérifications, en lançant des buts, sans abandonner la session de déboguage courante, ni avoir à lancer une seconde exécution de Prolog IV en parallèle.

stepports ***ports***

Positionne l'option `stepports` (utilisé par la commande `step`) à la valeur *ports*. Sans argument, affiche la valeur courante de l'option `stepports`.

nextports ***ports***

Positionne l'option `nextports` (utilisé par la commande `next`) à la valeur *ports*. Sans argument, affiche la valeur courante de l'option `nextports`.

contports ***ports***

Positionne l'option `contports` (utilisé par la commande `cont`) à la valeur *ports*. Sans argument, affiche la valeur courante de l'option `contports`.

8.4.7 Primitives et compilation

Voici comment entrer ou sortir du mode `debug` d'une part, et comment compiler un programme pour que celui-ci soit déboguable pendant l'exécution.

Changement de mode

debug, no_debug La commande prolog `debug` fait passer la machine prolog dans le mode `debug` quel que soit le mode dans lequel on était précédemment. On peut parenthéser des sessions de surveillance au sein de son programme au moyen des primitives `debug` et `no_debug`, mais il faut savoir que l'on ne disposera pas de toutes les informations sur la démonstration en cours si la requête n'a pas été lancée sous le mode `debug`. De plus, les informations sur les niveaux sont relatives au point de l'exécution où `debug` a été lancé, et non plus par rapport à la requête. Il n'y a aucun intérêt à passer en mode `debug` quand le programme qui tourne n'a pas été compilé pour ce mode.

Compilation

Pour compiler un programme avec des informations utilisables par le débogueur, il faut utiliser :

- la directive `:-debug.` dans le fichier (avant le premier paquet de règles),
- l'option de compilation `debug(on)`

Il est possible de compiler des portions de fichier (certains paquets de règles) en mode `debug` à l'aide des directives `:-debug` et `:-no_debug` que l'on place entre certains paquets de règles.

8.4.8 Problèmes et limitations

- Seuls peuvent être pris en compte par le débogueur les programmes donnés à Prolog IV au moyen des commandes de la famille `(re)compile`, quand ils sont explicitement compilés en mode `debug` (par quelque moyen que ce soit).
- Il s'ensuit que les paquets de règles qui ont été entrés au moyen de la famille des primitives `(re)consult` et `assert` sont ignorés par le débogueur.
- Les méta-appels, comme ceux lancés par `call`, «`;`» ou `freeze`, sont ignorés par le débogueur.

9

L'environnement graphique de Prolog IV

CE CHAPITRE décrit brièvement l'environnement graphique de Prolog IV. Outre la description des différents menus, fenêtres et dialogues, on y trouvera la liste des primitives graphiques accessibles depuis Prolog IV.

9.1 Lancement de Prolog IV

L'environnement graphique de Prolog IV est construit autour de l'outil TCL/TK, appartenant au domaine public. TCL est un langage de programmation et TK sa librairie graphique, originellement conçu pour X11[1]. Il faut donc que X11, Motif ou quelque serveur compatible X soit en fonctionnement sur votre terminal avant de pouvoir lancer l'environnement graphique. Si c'est le cas, on peut exécuter la commande `prolog4e` dans une fenêtre « shell », comme `xterm`. L'environnement lancé ouvre alors un panneau avec boutons, et une fenêtre console.

Du texte doit alors s'afficher en haut de cette fenêtre, ressemblant à ceci, aux dates et numéro de version près :

```
[Connection OK]
Prolog IV v1.0 (222), Juillet 1996 (C)PrologIA 1995,1996
[ Fri Jun 14 10:45:43 MET DST 1996 ]

>>
```

Si aucun texte ne s'affiche dans cette fenêtre, c'est que :

- la fin de l'installation (configuration) n'a pas été effectuée complètement ;
- il n'y a pas assez de mémoire pour faire tourner Prolog IV.
  - Peut-être plusieurs Prolog IV tournent-ils sur cette machine ?
  - Peut-être les tailles de piles sont-elles trop fortes pour la configuration mémoire de votre machine ? Il faut alors les réduire, dans le dialogue de préférences, sous le thème « Prolog ».

[1] Il est maintenant disponible sur les différents Windows *xx* de la société Microsoft, ainsi que sur Macintosh, d'Apple Corp.

9.2 Configuration initiale

Ceci est à effectuer une fois au moins, généralement lors de l'installation.

Il faut configurer les paramètres en rapport avec Prolog IV, et ce dans le dialogue des préférences, sous le thème « Prolog ». Le minimum requis est de mettre dans le champ « Exec Name: » le chemin de l'exécutable Prolog IV, et de cocher les boutons « Uses Tcl » et « Auto Launch ». Les substitutions ~ du shell Unix sont acceptées au début du chemin.

Pour que cette configuration soit conservée pour d'autres sessions, il faut l'enregistrer en activant l'item « Save Preferences » du menu « Info » du panneau principal.

L'exécutable Prolog IV peut alors être lancé en activant l'item « Run » du menu « Process » de la fenêtre intitulée « Console Prolog IV ».

On peut aussi quitter l'environnement graphique de Prolog IV et le relancer.

9.3 Le panneau principal

L'environnement graphique se présente sous la forme d'un panneau principal, de dialogues et de fenêtres d'édition appelés « éditeurs ».

Le panneau principal regroupe les diverses parties de l'environnement graphique, soit l'accès aux éditeurs, à diverses informations et à Prolog IV, fonctionnant dans une fenêtre appelée « Console Prolog IV ».

D'un certain point de vue, cet environnement n'est pas lié à Prolog IV, et il est théoriquement possible de lancer tout autre programme que Prolog IV utilisant les entrées/sorties standard d'Unix. Cependant, un bon nombre de fonctions sont largement orientées vers une utilisation Prolog.

Voici la description des différents éléments du panneau principal :

9.3.1 Le menu « Info »

Le menu info regroupe divers types d'information comme l'accès à un « Lisez-Moi », aux préférences, à un dialogue pour la rédaction de rapports de bugs, etc.

A propos de Prolog IV ... : affiche la version courante de Prolog IV.

Preferences ... : accès au dialogue de Préférences.

Save Preferences : sauve les préférences dans le répertoire HOME de l'utilisateur actuel.

Credits ... : quelques remerciements concernant certaines parties du logiciel.

Bug Report ... : crée un dialogue « normalisé » pour envoyer des descriptifs de problèmes rencontrés ou des suggestions concernant le logiciel.

Read Me ... : affiche une fenêtre contenant des informations sur l'environnement graphique (dont certaines de dernière minute). C'est un « Lisez-Moi ».

9.3.2 Le menu « File »

Le menu fichier donne accès à quelques fonctionnalités générales concernant les éditeurs de texte.

New : crée une nouvelle fenêtre d'édition, vide.

Open ... : propose un dialogue d'ouverture de fichier, et ouvre celui choisi dans une nouvelle fenêtre.

Save All : sauve tous les éditeurs modifiés. N'interroge le programmeur que lorsqu'un éditeur n'avait jamais été précédemment sauvé.

Close Unedited : ferme les éditeurs non-modifiés (les détruit).

9.3.3 Le menu « Windows »

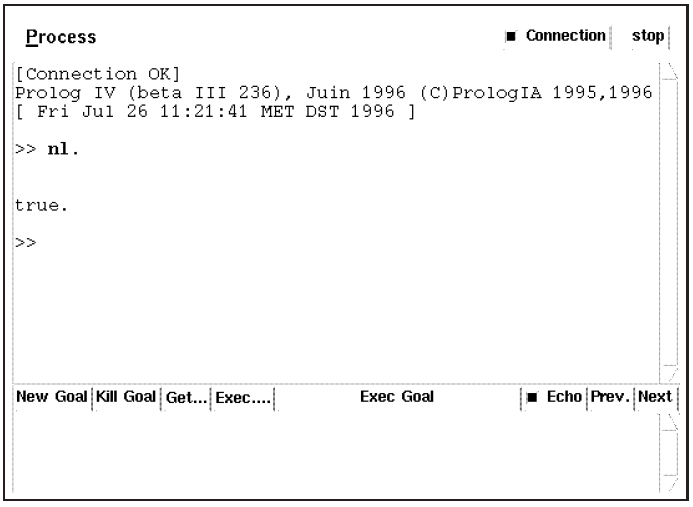
Le menu fenêtres ne montre à ce jour que les éditeurs ouverts. La console n'y figure donc pas. Cliquer dans un des items du menu fait venir la fenêtre de ce nom au premier plan (éventuellement en la faisant passer de l'état d'icône à l'état de fenêtre ouverte).

9.3.4 Le bouton « Quit »

Ce bouton permet de quitter l'environnement de développement Prolog IV. Le cas échéant, un dialogue demande de sauver les modifications effectuées dans les éditeurs.

9.4 La console Prolog IV

La console est la fenêtre qui permet au programmeur d'interagir avec Prolog IV. En plus de la barre de menu, elle contient un panneau de texte éditable qui affiche les entrées et les sorties de Prolog IV, ainsi qu'un panneau de texte qu'on appelera « Buts » permettant de conserver des buts qu'on utilise fréquemment.



La barre de menu se compose du menu et des boutons suivants :

9.4.1 Le menu « Process »

Run : lance l'exécutable désigné dans le champ « Exec. Name: » situé dans les préférences « Prolog », en lui concaténant les options du champ « Options: », puis la chaîne composée de deux tirets « -- », puis les options du champ « User-Options ». « Run » termine l'exécution d'un éventuel processus lancé préalablement dans la console, sans demander confirmation (en utilisant « Kill »).

Kill : termine brutalement l'exécution du processus lancé dans la console. Pour les curieux, c'est un signal SIGTERM qui est généré sous UNIX.

Preferences ... : donne l'accès directement aux préférences en rapport avec Prolog IV (ou plus généralement l'exécutable à lancer).

9.4.2 Le bouton « Stop »

Ce bouton envoie une interruption utilisateur au processus tournant dans la console. Dans le cas de Prolog IV, un but en cours d'exécution sera interrompu, avec un retour au prompt, à moins que le programme Prolog IV n'intercepte cette interruption. Pour les curieux, c'est un signal SIGINT qui est généré sous UNIX.

9.4.3 Le bouton « Connection »

Il est utile de savoir que s'il est coché, c'est que Prolog IV tourne effectivement dans la console, et que ses entrées/sorties standard y seront saisies ou affichées.

Un autre bouton « Connection » est présent dans la fenêtre du débogueur.

Note : la sortie « stderr » d'un processus ne peut être montrée.

9.4.4 Le panneau des buts

Le panneau « Buts » comporte une partie texte (avec ascenseur) et quelques boutons et menus. Ce panneau permet d'éditer des buts, de les lancer et de se déplacer parmi eux. L'ensemble des buts de ce panneau est organisé sous la forme d'une file. On peut insérer et supprimer des buts.

New Goal : crée un nouveau but. Il s'insère après celui présenté.

Kill Goal : détruit le but présenté dans la zone texte. Le but suivant (s'il existe) est alors visualisé à sa place (sinon c'est le précédent).

Get ... : visualise dans (pour une éventuelle édition) la zone texte le but choisi dans ce menu.

Exec ... : lance (sans le mettre dans la zone texte) le but choisi dans ce menu.

Exec Goal : lance le(s) buts contenu(s) dans la zone texte. En fait, ceci envoie tout texte contenu dans cette zone à Prolog, sans la moindre vérification. On peut dont y mettre aussi bien des requêtes que des entrées.

Echo : quand ce bouton est coché, les buts lancés sont affiches dans la console.

Prev. : visualise le but suivant dans la liste des buts. Emet un bip s'il n'existe pas.

Next : visualise le but précédent dans la liste des buts. Emet un bip s'il n'existe pas.

Des raccourcis-clavier existent pour certains boutons (attention aux majuscules et minuscules) :

- `ENTER` (du pavé numérique) pour le bouton « Exec Goal » qui envoie donc le but courant dans la console,
- `CTRL-p` pour le bouton « Prev. »,
- `CTRL-n` pour le bouton « Next »,
- `CTRL-N` pour le bouton « New Goal »,
- `CTRL-D` pour l'action « Dupliquer le but actuellement présenté » (effectue donc un « New Goal »),
- `CTRL-K` pour le bouton « Kill Goal ».

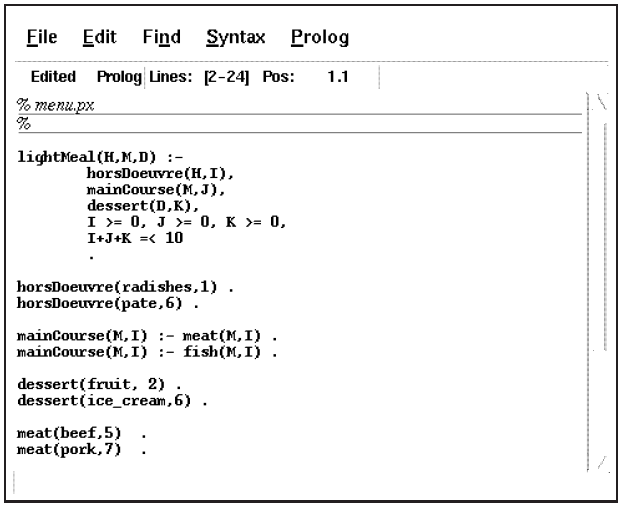
9.5 La Console Tcl/Tk

Les utilisateurs ayant une certaine connaissance du langage TCL peuvent l'utiliser.

La console Tcl/Tk permet d'exécuter des commandes TCL. Il suffit, après avoir tapé la commande, de la sélectionner et de presser le bouton « Eval Selection ». On peut aussi, si la commande ne tient que sur une seule ligne (qui est éventuellement lovée sur plusieurs lignes physiques), faire « `Enter` » (du pavé numérique), le point d'insertion étant dans cette ligne. A l'heure actuelle, les versions utilisées de cet outil sont les suivantes : Tcl 7.3 et Tk 3.6, ainsi que TclX 7.3 .

9.6 Les Editeurs

Chaque fenêtre dispose de sa propre barre de menu, ainsi qu'une ligne comportant quelques informations, placée sous la barre. Le titre de la fenêtre porte le nom du fichier (Untitled-n s'il n'a pas encore été sauvé) et son chemin.



La ligne de statut contient les éléments suivants, de la gauche vers la droite :

- Un statut du fichier édité (rien ou « Edited » si le fichier n'a pas été sauvé depuis la dernière modification).
- Un menu indiquant le langage de programmation utilisé dans d'éditeur (à positionner soi-même). Ceci n'a d'importance que pour la coloration syntaxique dynamique.
- L'intervalle de lignes visibles dans la fenêtre.
- La position actuelle du curseur d'insertion, sous la forme *ligne.colonne* (les lignes commencent à 1, les colonnes à 0).
- Une éventuelle action en cours d'exécution.

La barre de menu se compose des menus « File », « Edit », « Find », « Syntax », « Prolog » décrits ci-après.

9.6.1 Le menu « File »

New Window... : crée un nouvel éditeur.

Clear All : efface le contenu de l'éditeur. Demande confirmation s'il y a lieu.

Load File ... : présente un dialogue d'ouverture de fichier à charger dans la fenêtre courante en remplacement du fichier déjà présent. Demande confirmation s'il y a lieu.

Save : sauve le contenu de la fenêtre dans un fichier. La première fois, un nom sera demandé.

Save As ... : sauve le contenu de la fenêtre dans un fichier dont le nom sera demandé. Renomme la fenêtre le cas échéant.

Save To ... : sauve le contenu de la fenêtre dans un autre fichier, dont le nom sera demandé.

Close : ferme la fenêtre. Demande confirmation s'il y a lieu.

9.6.2 Le menu « Edit »

Le menu « Edit » ressemble beaucoup à ce qui se fait sur Macintosh ou NeXT.

? Undo : n'est pas encore implanté.

Cut : copie la sélection de la fenêtre (si elle existe) dans un « presse-papier » puis détruit cette sélection.

Copy : copie la sélection de la fenêtre (si elle existe) dans un « presse-papier »

Paste : colle le contenu du « presse-papier » au point d'insertion. Ceci ne remplace pas la sélection.

Paste Selection : colle le texte d'une sélection (de quelque fenêtre X11 qu'elle vienne) au point d'insertion. Ceci ne remplace pas la sélection.

Clear : détruit cette sélection.

Shift Left : décalage à gauche (d'un tab) des lignes de la sélection.

Shift Right : décalage à droite (d'un tab) des lignes de la sélection.

Misc. : accès à un sous-menu permettant d'établir ou non l'auto-indentation des lignes lors de la frappe d'un retour-chariot ou d'établir la coloration syntaxique dynamique.

Select All : sélectionne tout le contenu de la fenêtre.

Font list : accès à un sous-menu permettant de choisir la police de caractère à utiliser dans la fenêtre.

9.6.3 Le menu « Find »

Le menu « Find » permet d'effectuer des recherches et des remplacements, en tenant compte ou en ignorant la casse[2], en recherchant des expressions régulières. Le dialogue « Find/Replace » peut être activé par l'item « Find ... » du menu ; il est décrit plus loin.

Find ... : appelle le dialogue de Recherche/Remplacement.

Find Next : recherche et sélectionne la prochaine occurrence.

Find <Selection> : recherche et sélectionne la prochaine occurrence de la chaîne sélectionnée.

Replace : remplace la sélection par la chaîne de substitution.

Replace & Find : remplace la sélection par la chaîne de substitution puis cherche la prochaine occurrence.

Replace All : effectue le remplacement dans tout le fichier.

Find All : colorie toutes les occurrences de la chaîne à chercher (ce n'est pas une sélection, juste une coloration sans incidence sur l'édition !).

Next Found : fait défiler le texte pour montrer la prochaine occurrence colorée dans la fenêtre.

Unhighlight All Found : enlève la coloration des textes trouvés par des recherches antérieures (voir « Find all »).

Selection -> Find-field : place le texte de la sélection courante dans le champ « Find » du dialogue de Recherche/Remplacement.

Selection -> Replace-field : place le texte de la sélection courante dans le champ « Replace » du dialogue de Recherche/Remplacement.

9.6.4 Le menu « Syntax »

Check Syntax : vérifie la syntaxe du contenu de la fenêtre. Présente les lignes erronées ou comportant un avertissement dans un style (couleur) spécial (ce n'est pas une sélection, juste une coloration). On peut se servir ensuite de « Goto Next Error » pour les visualiser. Passer la souris sur des caractères de ce style fait apparaître les messages d'erreur ou d'avertissements en rapport avec cet endroit.

[2] l'attribut « majuscule » ou « minuscule » d'une lettre.

Check Syntax (Selection) : même chose que précédemment, mais ne marche que pour le contenu de la sélection.

Goto Next Error : rend visible à l'écran une éventuelle prochaine ligne colorée par « Check Syntax ». Arrivé sur la dernière erreur, fait un bip et repasse à la première erreur du fichier.

Clear All Error Marks : ôte de tout le fichier les styles spéciaux « erreur » et « avertissement » des portions de texte soupconnées comme étant erronées ou ayant donné lieu à des avertissements.

Clear Warning Marks : ôte de tout le fichier le style spécial « avertissement » des portions de texte ayant donné lieu à des avertissements.

Syntax Coloring : colorie tout le fichier (entités syntaxiques et règles). *C'est pour l'instant encore expérimental.* On ne peut pas l'enlever simplement après coup.

9.6.5 Le menu « Prolog »

ce menu permet l'accès aux commandes principales d'ajout de règles à l'exécutable Prolog IV, si celui-ci est en fonctionnement. Le compilateur Prolog IV est utilisé. Le mode (re)consult est disponible, au travers d'un sous-menu. Par défaut, toutes les opérations qui suivent font d'abord appel à la vérification syntaxique.

Check Syntax before : si cet item est coché, l'activation des items décrits ci-après entrainera préalablement la vérification syntaxique de tout le fichier (ou toute la sélection). S'il y a des erreurs, un mini-rapport est présenté dans un dialogue et l'action en cours (comme compile) est automatiquement annulée. Il n'est pas tenu compte de la présence d'avertissements (warning).

Debug : si cet item est coché, les compilations ultérieures de texte Prolog dans tous les éditeurs se feront en mode « debug ».

Compile Window : compile le contenu de la fenêtre (pas du fichier).

Compile Selection : compile le contenu de la sélection de la fenêtre.

Recompile Window : recompile le contenu de la fenêtre (pas du fichier).

Recompile Selection : recompile le contenu de la sélection de la fenêtre.

Consulting : permet d'accéder aux commandes de consultation de fenêtre.

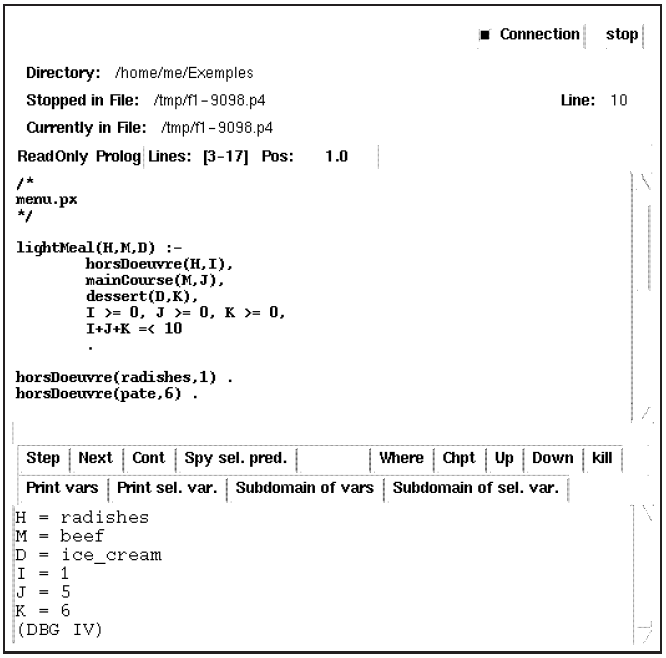
9.7 Le Débogueur

Ce dialogue est en fait une interface avec le débogueur Prolog IV décrit dans le chapitre « Environnement ». Elle offre en plus la visualisation du source de ce qui a été compilé (en mode `debug`) pendant l'exécution.

9.7.1 Constitution

Des boutons du panneau central facilitent l'usage des commandes du débogueur, en évitant à l'utilisateur de s'en souvenir et d'avoir à les taper. Les commandes les plus usuelles ont été associées à des boutons.

Cliquer sur un de ces boutons équivaut en général à taper le texte correspondant à la commande dans l'entrée courante. Il faut donc que le débogueur Prolog IV soit en attente de lecture d'une commande.



Cette fenêtre comporte trois parties principales, qui sont :

- une zone de texte dans laquelle un source prolog peut être affiché, par le débogueur.
- un panneau de boutons, effectuant des commandes du débogueur.
- un zone de texte pouvant contenir les entrées/sorties du débogueur ; son invite s'y affiche.

Un bouton à cocher, nommé Connection indique si les sorties (que ce soient celles du débogueur ou les réponses aux requêtes) se font dans cette fenêtre plutôt que dans la console Prolog IV.

9.7.2 Usage

Pour que le débogueur source soit utilisable, il faut avoir compilé le contenu des fenêtres en mode `debug`, en positionnant l'item « Debug » du menu « Prolog » de l'un des éditeurs. On actionne ensuite l'un des items de compilation (pas de consultation !) de ce même menu (« compile xxx » ou « recompile xxx »).

Dans la console Prolog IV, il faut avoir exécuté la primitive `debug.` pour se mettre dans le mode du même nom, avant de taper une requête. Pendant l'exécution de la requête, la fenêtre du débogueur vient se placer au premier plan dès que des portions compilées en mode `debug` viennent à être exécutées. Il faut parfois effectuer plusieurs fois l'action `step` (ou le bouton de même nom) avant que le source ne soit affiché, certains buts (comme les primitives prédéfinies) n'étant pas montrables au niveau source.

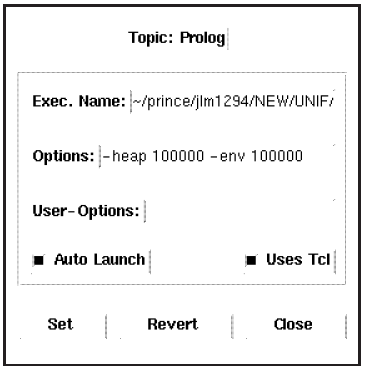
9.8 Dialogues

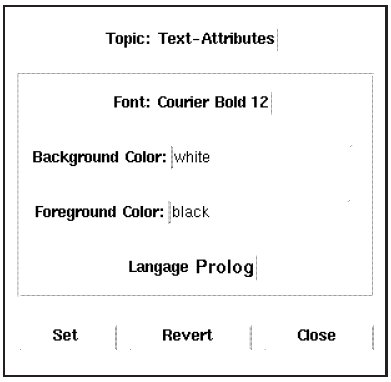
Sans aller jusqu'à décrire tous les dialogues présents dans l'environnement graphique de Prolog IV, présentons les principaux dans les sections suivantes.

9.8.1 Préférences

Ce dialogue permet, par le biais de deux sous-panneaux qui repésentent chacun un thème, de :

- positionner le paramétrage de l'exécutable Prolog IV lancé dans la fenêtre Console ;
- choisir quelques attributs généraux des éditeurs de textes.





Dans tous les cas, les boutons Set, Revert et Close sont présents. Voici leurs significations :

Set : valide les valeurs des options positionnées.

Revert : remet les options à leurs dernières valeurs enregistrées par Set.

Close : ferme la fenêtre ; les modifications non validées par Set sont perdues.

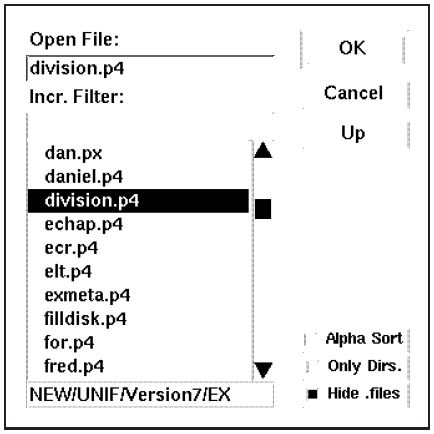
Changer de thème perd également les modifications non validées par Set.

Pour enregistrer ces divers paramètres d'une session à l'autre, il faut actionner l'item « Save Preferences » du menu « Info » du panneau principal.

9.8.2 Boîte d'ouverture/enregistrement de fichier

Ce dialogue permet de choisir un fichier à charger dans un éditeur. Ce même dialogue permet de donner un nom à un fichier que l'on veut enregistrer sur disque.

Voici les différentes parties de ce dialogue.



Les champs éditables :

Open File: : ou

Load File: : ou

Save File: : selon la nature de l'opération, contient le nom de fichier à charger ou à enregistrer.

Incr. Filter: : permet de composer incrémentalement un nom de fichier. Au fur et à mesure de la frappe, les noms de fichiers conformes au filtre courant sont affichés dans la liste des fichiers.

Une liste déroulable de fichiers du répertoire courant (et conformes au filtrage courant) est affichée. On peut bien entendu double-cliquer sur un nom de fichier pour confirmer son choix. Dans le cas d'un double-clic sur le nom d'un répertoire (une marque / est placée devant un nom de répertoire), on est placé dans celui-ci et la liste est mise à jour en fonction de son contenu. Un double-clic équivaut à faire une sélection et cliquer le bouton OK.

Sous cette liste se trouve dans une étiquette déroulable le chemin complet du répertoire qu'on visite. Cette étiquette n'est pas éditable. On peut le faire défiler avec la souris.

Les boutons :

OK : valide le choix effectué. Dans le cas d'un répertoire, s'y positionne.

Cancel : annule l'opération en cours et fait disparaître le dialogue.

Up : passe au répertoire ancêtre.

Les boutons à cocher, une fois qu'ils le sont, modifient le comportement de la boîte de dialogue de la façon suivante :

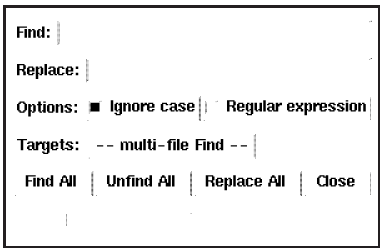
Alpha Sort. : trie les noms alphabétiquement, en ne tenant pas compte des majuscules/minuscules.

Only Dirs. : ne montre que les répertoires.

Hide .files : ne montre pas les fichiers dont le nom commence par le caractère « . » (point) .

9.8.3 Recherche/Remplacement

Ce dialogue permet d'effectuer des recherches et des remplacements de texte dans un ou plusieurs éditeurs de l'environnement.



Les champs éditables :

Find: : contient le texte à rechercher.

Replace: : contient le texte de remplacement.

Des boutons à cocher indiquent le type de recherche :

Ignore Case: : ne tient pas compte des majuscules/minuscules pendant la recherche.

Regular Expression: : une expression régulière sera interprétée dans le champ Find:.

Un menu Targets: permet de choisir les cibles parmi les éditeurs actifs pour les recherches et remplacements. Leurs noms sont dans ce menu. Un ou plusieurs peuvent être sélectionnés.

Les boutons d'actions :

Find All : recherche la chaîne du champ Find: dans toutes les cibles, en colorant les occurrences trouvées.

Unfind All : recherche la chaîne du champ Find: dans toutes les cibles, en décolorant les occurrences trouvées le cas échéant.

Replace All : remplace les occurrences de la chaîne du champ Find: par la chaîne du champ Replace: dans toutes les cibles.

Close : ferme la fenêtre.

Une jauge ainsi que des compteurs d'occurrences (un pour la cible en cours, l'autre pour le cumul en cours) sont présents en bas du dialogue.

9.9 Clavier et souris

9.9.1 La souris

Les deux boutons gauche et droit (gauche et centre si la souris a trois boutons) de la souris sont utilisés sous TCL.

- Le gauche sert toujours aux actions principales.
- Le droit dépend du contexte.

Par exemple, cliquer le bouton droit sur un nom de menu permet de détacher celui-ci.

Dans un texte, maintenir enfoncé le bouton droit en déplaçant la souris permet de se déplacer rapidement. Ceci marche aussi dans un champ éditable.

Sélectionner du texte se fait à l'aide du bouton gauche, comme sur le macintosh (ou le NeXT), avec Shift-clic pour étendre une sélection jusqu'au point cliqué.

Le bouton droit d'une souris à trois boutons n'est a priori pas utilisé.

9.9.2 Touches

Voici quelques informations sur l'utilisation du clavier dans certaines conditions.

Raccourcis-menu

Les raccourcis-menu permettent de naviguer dans les menus en ne se servant que du clavier. Ils se repèrent immédiatement par le soulignement d'un caractère du menu ou de l'un de ses items.

Pour accéder et faire apparaître un menu :

- Taper `MOD-`*lettre* où *lettre* est soulignée dans la barre de menu active.
- Ensuite quand le menu est déroulé :
  - utiliser les flèches pour se déplacer parmi les items, puis taper la touche `Return`.
  - ou bien une lettre pour activer l'item contenant la lettre soulignée.

La touche indiquée par `MOD` est un modifieur qui dépend de votre plate-forme ; c'est la plupart du temps la touche `Alt`. Sur NeXT, c'est la touche « `Command` ». Sur SUN c'est la touche « `Alt Graph` ».

Raccourcis-clavier

Les raccourcis-clavier permettent d'effectuer une action, sans passer par un menu pour l'activer. Ils sont souvent indiqués à droite de l'item de menu ayant la fonction équivalente.

Effacer la sélection

`Ctrl-d` efface la sélection, de l'objet éditeur ou d'un champ éditable.

Ouverture/Fermeture de menu

La touche `F10` ouvre le premier menu de la fenêtre.

La touche `ESC` ferme un menu déroulé.

9.10 Informations diverses

- Le sommaire du « Read-Me » est un hyper-texte.

- L'analyseur syntaxique effectuant l'opération « Check Syntax » tient compte des directives `op/3` qui figurent dans le fichier. Toutefois, son état interne, en ce qui concerne les opérateurs, reste sans rapport avec l'état interne de Prolog IV si ce dernier a été lancé.
- Les avertissements sont montrés en même temps que les erreurs dans le mini-rapport de « Check Syntax ».
- Passer sur une ligne coloriée comme « erronée » (dans un éditeur) affiche les erreurs ou avertissements en rapport avec cette ligne.
- Il y a une procédure automatique de sauvegarde des éditeurs au contenu non sauvé. Pour chaque éditeur, un fichier « backup » est créé dans `/tmp` et son nom est de la forme `b.ed`N`.frEdit-`PID avec N un entier et PID le numéro de processus de l'environnement graphique auquel appartient l'éditeur. Il n'y a backup que des éditeurs ayant le statut « modifié et non sauvé » (Edited), et ce toutes les cinq minutes. Ce fichier backup est détruit à la fermeture de la fenêtre, et en fin de session.
- Coller quelque chose quand le curseur est dans la sélection courante revient à coller après cette sélection. L'exception est quand le curseur se trouve juste au début de la sélection, auquel cas on colle à cet endroit. Tout ceci dans le but de permettre des séquences de « coller ».
- Les fichiers temporaires (créés dans /tmp) sont effacés en fin de session.

9.11 Primitives graphiques

9.11.1 Préalables

Les primitives graphiques fournies permettent de créer des dessins, sous la forme de lignes, rectangles et textes. Ce sont des objets vectoriels, dans le sens où ils ont une certaine indépendance et peuvent être détruits sélectivement. Un programmeur connaissant TCL peut enrichir de façon importante le (petit) jeu de primitives fournies, en modifiant les coordonnées et attributs de ces objets.

Système de coordonnées

On notera les coordonnées par des nombres Prolog IV, qu'ils soient entiers ou flottants. Des rationnels peuvent être également utilisés, quand les valeurs du numérateur et du dénominateur sont « petites »[3].

Le système de coordonnées est défini comme suit, dans chaque panneau graphique :

- L'axe des x est horizontal et croît de gauche à droite.
- L'axe des y est vertical et croît de haut en bas.
- L'origine des axes est dans le coin en haut à gauche du panneau graphique.

[3] c.à.d. qu'elles sont manipulables en C.

9.11.2 Primitives

gr_init doit être lancée une fois pour initialiser le graphisme et donc pour permettre l'usage des autres primitives graphiques. Cette primitive crée également la fenêtre graphique, constituée d'un panneau graphique et d'ascenseurs.

gr_moveto(X,Y) déplace le crayon au point de coordonnées (X,Y), qui devient le point courant.

gr_lineto(X,Y) tire une ligne du point courant au point (X,Y).

gr_line(DX,DY) tire une ligne du point courant (X_0,Y_0) au point X_0+DX, Y_0+DY. On transmet donc à cette primitive des variations de coordonnées.

gr_rect(V, [X,Y,X2,Y2]), gr_rect(V, X,Y,X2,Y2) dessinent un rectangle :

- V est l'action (`frame` ou `paint`).
- X,Y sont les coordonnées du coin en haut à gauche.
- $X2$,$Y2$ les coordonnées du coin en bas à droite.

gr_drawstring_at(S, X,Y) pose un objet chaîne contenant S au point (X,Y).

gr_drawstring_at(S, X,Y, X2,Y2) Pose un objet chaîne contenant S dans le rectangle de coins (X,Y) et ($X2$,$Y2$).

gr_arrow(X,Y, X2,Y2) dessine une flèche du point (X,Y) au point destination ($X2$,$Y2$).

gr_setxfont(FX11) positionne la fonte $FX11$ (en employant la dénomination des fontes X11) pour les textes qui seront créés ultérieurement.

gr_setfont(N), gr_setfont(N, T), gr_setfont(N, T, G) positionnent la fonte des textes qui seront créés ultérieurement :

- N est le nom (helvetica, times, courier, etc)
- T est la taille (12, 14, 18, 24, etc)
- G est la graisse (bold, medium, etc)

D'autres fontes peuvent être disponibles sur une plate-forme donnée.

gr_fgcolor3(R, G, B) positionne la couleur des objets ultérieurement créés à la valeur #RGB, une chaîne représentant une couleur et formée des composantes rouge, verte et bleue données comme arguments. C'est une convention X11 ; par exemple, #ff0000 désigne le rouge, #ffffff est le blanc, et #000000 est le noir.

gr_fgcolor(Coul) positionne la couleur des objets ultérieurement créés à $Coul$, qui doit être un nom de couleur X11 (comme aquamarine, slategray, etc).

gr_settag(Tag) positionne l'étiquette Tag (un atome), pour les prochains objets à créer. La valeur spéciale { } l'enlève.

gr_erase détruit tous les objets de la fenêtre graphique courante.

gr_erase(Tag) détruit tous les objets étiquetés par Tag de la fenêtre graphique courante.

gr_drawtree(T, R) dessine l'arbre T dans le rectangle (partiellement inconnu) R, qui est une liste de quatres coordonnées ($[X1, Y1, X2, Y2]$).

9.12 Quelques Problèmes et limitations

La version de TCL/TK utilisée pour faire fonctionner l'environnement graphique de Prolog IV n'étant pas la dernière, d'anciens petits problèmes liés à cet outil sont toujours présents.

La prochaine version de Prolog IV utilisera une version récente de TCL/TK et remédiera à la plupart d'entre eux.

- Dans certaines conditions, remettre une fenêtre au premier plan peut prendre quelques secondes (c'est corrigé dans la dernière version de TK, non encore intégrée).
- Pour accéder aux accélérateurs de menu (les lettres soulignées, pas les raccourcis!) il faut que le focus soit dans la partie texte, autrement dit y avoir cliqué.
- La plupart des problèmes de focus sont réglés dans la dernière version de TK.
- La coexistence des deux modes de sélection Macintosh et X11 rend les choses un peu confuses. (la dernière version de TK rend les choses plus claires).
- La coloration syntaxique (dynamique ou pas) peut prendre un certain temps.
- Toute l'interface utilisateur est en anglais.
- Il n'est pas possible de faire des Recherche/Remplacement en arrière.
- Il n'est pas possible de faire des Recherche/Remplacement de texte contenant un retour-chariot. On est donc limité à une seule ligne de texte à rechercher.
- Les colorations syntaxiques des modes C/C++ et TCL ne sont pas implantées.
- Une lettre accentuée est considérée comme un mot isolé par l'éditeur.
- Il faudrait fournir plusieurs jeux de styles pour la coloration syntaxique en fonction du nombre de couleurs de l'écran (pour l'instant, on est supposé avoir un écran couleur, ou noir et blanc)
- Il faudrait également permettre lédition de ces styles (un programmeur connaissant TCL/TK peut le faire).

- Pas de « Undo » dans l'éditeur.
- Alors que Prolog IV est conçu pour travailler avec le jeu de caractère de la machine, le cas du NeXT présente une particularité intéressante :
  - sous shell, Prolog IV doit utiliser le jeu NeXTStep.
  - sous l'environnement graphique, Prolog IV doit utiliser le jeu ISO (X11).

  Bref, Prolog IV ne fonctionne parfaitement avec les accents que dans une fenêtre Terminal. (Il faudrait une option de la ligne de commande pour choisir dynamiquement le jeu de caractère, ou bien encore les classes de caractères pour les entités lexicales)
- Des items de menus ne sont pas encore implantés (ils sont généralement grisés et commencent par un « ? »).
- Dans certains cas, le source montré dans la fenêtre du débogueur n'est pas le bon (alors que ce qui est montré dans le modèle des boîtes l'est toujours à certains numéros près.

Cette version de l'environnement graphique de Prolog IV est un premier jet et s'avère déjà à l'usage d'utilisation très agréable. La correction de la plupart des problèmes, ainsi que de nombreuses possibilités seront disponibles dès la prochaine version.

Index

Symboles

A

B

C

D

E

F

G

H

I

L

M

N

O

P

Q

R

S

T

U

V

W

X

Ref. MPH-IV/Fr/MMX

www.prolog-heritage.org