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 incluses 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 = 1/713623846352979940529142984724747568191373312.
>> 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 profondeur de n, c'est-à-dire quand n choix ont été effectués.

- c = `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 = *my_stop* : Utilisation du prédicat *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.