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 exactement 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'évaluation.

Opérations sur les entiers

$$\begin{array}{llll}
add_I & I \times I & \longrightarrow & I \\
sub_I & I \times I & \longrightarrow & I \\
mul_I & I \times I & \longrightarrow & I \\
intdiv_I & I \times I & \longrightarrow & I \cup \{\texttt{zero_divisor}\} \\
rem_I & I \times I & \longrightarrow & I \cup \{\texttt{zero_divisor}\} \\
mod_I & I \times I & \longrightarrow & I \cup \{\texttt{zero_divisor}, \texttt{undefined}\} \\
neg_I & I \times I & \longrightarrow & I \\
abs_I & I \times I & \longrightarrow & I \\
sign_I & I \times I & \longrightarrow & I
\end{array}$$

Les opérations add_I, sub_I, mul_I et neg_I ont leur sens habituel et on a les définitions suivantes :

$$\begin{array}{llll}
intdiv_I(x,y) & = & rnd_I(x/y) & \text{si } y \neq 0 \\
 & = & \texttt{zero_divisor} & \text{si } y = 0 \\
 & & & \\
rem_I(x,y) & = & x - (rnd_I(x/y) * y) & \text{si } y \neq 0 \\
 & = & \texttt{zero_divisor} & \text{si } y = 0 \\
 & & & \\
mod_I(x,y) & = & x - (rnd_I(\lfloor x/y \rfloor) * y) & \text{si } y \neq 0 \\
 & = & \texttt{zero_divisor} & \text{si } y = 0
\end{array}$$

$$\begin{array}{lll}
sign_I(x) & = 1 & \text{si } x \geq 0 \\
 & = 0 & \text{si } x < 0
\end{array}$$

Prolog IV utilise la définition de la partie entière suivante (« round toward zero ») :

$$|rnd_I(x)| \leq |x|$$

Opérations sur les flottants

$$\begin{array}{llll}
add_F & F \times F & \longrightarrow & F \cup \{\texttt{overflow}, \texttt{underflow}\} \\
sub_F & F \times F & \longrightarrow & F \cup \{\texttt{overflow}, \texttt{underflow}\} \\
mul_F & F \times F & \longrightarrow & F \cup \{\texttt{overflow}, \texttt{underflow}\} \\
div_F & F \times F & \longrightarrow & F \cup \{\texttt{overflow}, \texttt{underflow}, \texttt{zero_divisor}\} \\
neg_F & F \times F & \longrightarrow & F \\
abs_F & F \times F & \longrightarrow & F \\
sqrt_F & F \times F & \longrightarrow & F \cup \{\texttt{undefined}\} \\
sign_F & F \times F & \longrightarrow & F \\
intpart_F & F \times F & \longrightarrow & F \\
fractpart_F & F \times F & \longrightarrow & F
\end{array}$$

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 :

$$\begin{array}{lll}
sign_F(x) & = 1 & \text{si } x \geq 0 \\
 & = 0 & \text{si } x < 0
\end{array}$$

$$intpart_F(x,y) \quad = \quad sign_F(x) * \lfloor |x| \rfloor$$

$$fractpart_F(x,y) \quad = \quad x - intpart_F(x)$$

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 = 25925925925925925923333074074074074074 1.
?- 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(111111111111111111111111).
X = 1.111111111111111e+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*