PROLOG II+

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

Contact : association@prolog-heritage.org

PROLOG II+

MANUEL DE REFERENCE

Table des matières

Table des matières ..... **iii**

Introduction ..... **ix**
Standardisation ix
Performances ix
Modularité x
Ouverture vers l'extérieur x
Environnement de programmation x
Interface graphique xi

Sur ce Manuel ..... **xiii**

Garantie et responsabilités ..... **xv**
Droits d'auteur xv

0. Débuter avec Prolog ..... **R 0 - 1**
0.1. Leçon 1 : Lancement de Prolog ..... R 0 - 1
0.2. Leçon 2 : Utilisation d'un programme d'exemple ..... R 0 - 2
0.3. Leçon 3 : L'Editeur ..... R 0 - 4
0.4. Les erreurs les plus courantes ..... R 0 - 5
0.4.1. Débordement de l'espace de code. ..... R 0 - 5
0.4.2. Interruption. ..... R 0 - 5
0.4.3. Erreurs détectées en cours d'insertion ..... R 0 - 6
0.4.4. Erreurs concernant les fichiers. ..... R 0 - 6
0.4.5. Erreurs de syntaxe ..... R 0 - 6
0.4.6. Erreurs d'exécution ..... R 0 - 8

1. Eléments de base ..... **R 1 - 1**
1.1. Notations utilisées ..... R 1 - 1
1.2. Jeu de caractères ..... R 1 - 2
1.3. Les variables ..... R 1 - 3
1.4. Constantes ..... R 1 - 5
1.5. Termes et arbres ..... R 1 - 9
1.6. Les opérateurs ..... R 1 - 15
1.7. Les règles et les assertions ..... R 1 - 16
1.8. Les mécanismes de base de Prolog ..... R 1 - 18
1.9. La syntaxe complète de Prolog II+ ..... R 1 - 20

1.9.1. Le niveau syntaxique ..... R 1 - 21
1.9.2. Les opérateurs ..... R 1 - 22
1.9.3. Le niveau lexical ..... R 1 - 24
1.9.4. Les caractères ..... R 1 - 26
1.9.5. Les caractères accentués ..... R 1 - 27
1.10. Le macroprocesseur ..... R 1 - 29

2. Le contrôle ..... R 2 - 1
2.1. Le contrôle ..... R 2 - 1
2.2. Geler ..... R 2 - 9
2.3. A propos des arbres infinis ..... R 2 - 11
2.4. Quelques conseils pour la programmation récursive ..... R 2 - 12
2.5. Les meta-structures ..... R 2 - 14
2.5.1. Création ..... R 2 - 14
2.5.2. Récupération ..... R 2 - 15
2.5.3. Unification forcée ..... R 2 - 15
2.5.4. Sorties ..... R 2 - 16
2.5.5. Exemple ..... R 2 - 16

3. Structuration des règles ..... R 3 - 1
3.1. Introduction ..... R 3 - 1
3.1.1. Qualification des prédicats ..... R 3 - 1
3.1.2. Qualification des foncteurs ..... R 3 - 2
3.1.3. Simplification des notations ..... R 3 - 3
3.1.4. Quelques règles simples d'utilisation ..... R 3 - 3
3.1.5. Modules et préfixes standard ..... R 3 - 4
3.2. Terminologie ..... R 3 - 5
3.3. Syntaxe des identificateurs ..... R 3 - 6
3.3.1. Paramétrage de l'écriture d'un identificateur complet ..... R 3 - 7
3.4. Contexte de Lecture et Ecriture ..... R 3 - 8
3.4.1. Lecture ..... R 3 - 8
3.4.2. Ecriture ..... R 3 - 8
3.4.3. Notation simplifiée de la suite d'identificateurs d'un contexte ..... R 3 - 9
3.4.4. Notation en Prolog ..... R 3 - 11
3.4.5. Primitives pour les contextes et les familles d'identificateurs ..... R 3 - 11
3.5. Modules ..... R 3 - 13
3.5.1. Définition d'un module ..... R 3 - 14
3.5.2. Module source ..... R 3 - 14
3.5.3. Module Objet ..... R 3 - 17

3.6. Résumé ou Approche simplifiée des identificateurs, contextes et modules ..... R 3 - 17
3.6.1. Identificateurs ..... R 3 - 17
3.6.2. Notation abrégée et contextes ..... R 3 - 18
3.6.3. Modules ..... R 3 - 19
3.7. Ajout, suppression et recherche de règles ..... R 3 - 20
3.8. Manipulation des modules compilés ..... R 3 - 32

4. Opérations prédéfinies ..... R 4 - 1
4.1. Les tests de type ..... R 4 - 1
4.2. Les opérations arithmétiques ..... R 4 - 2
4.3. Affectation ..... R 4 - 7
4.4. Opérations sur les chaînes ..... R 4 - 9
4.5. Composition et décomposition d'objets ..... R 4 - 10
4.6. Comparaison de termes quelconques ..... R 4 - 14

5. Les entrées / sorties ..... R 5 - 1
5.0. Généralités ..... R 5 - 1
5.1. Entrées ..... R 5 - 2
5.1.1. Règles pour l'entrée. ..... R 5 - 3
5.1.2. Modification de l'unité d'entrée ..... R 5 - 8
5.2. Sorties ..... R 5 - 9
5.2.1. Règles pour la sortie ..... R 5 - 9
5.2.2. Modification de l'unité de sortie ..... R 5 - 12
5.3. Chargement et adaptation du module de dessin ..... R 5 - 14
5.4. Déclaration d'opérateurs ..... R 5 - 15

6. L'environnement ..... R 6 - 1
6.1. Comment sortir de Prolog ..... R 6 - 1
6.2. Démarrage automatique d'un programme Prolog ..... R 6 - 1
6.3. Edition de programmes ..... R 6 - 2
6.3.1. Modifier des paquets de règles ..... R 6 - 2
6.3.2. Editer et recompiler un module source ..... R 6 - 3
6.3.3. Editer un fichier de texte quelconque ..... R 6 - 4
6.4. Date, temps et mesures ..... R 6 - 4
6.5. Lien avec le système ..... R 6 - 5
6.6. Outil de mise au point de programmes ..... R 6 - 5
6.6.1. Mode trace ..... R 6 - 6
6.6.2. Mode interactif ..... R 6 - 7
6.6.2.1. Points d'arrêt ..... R 6 - 7
6.6.2.2. Progression dans le code ..... R 6 - 8

6.6.2.3. Terminer l'exécution ..... R 6 - 9
6.6.2.4. Affichage des informations..... R 6 - 9
6.6.2.5. Informations annexes sur l'exécution..... R 6 - 14
6.6.2.6. Configuration ..... R 6 - 16
6.6.3. Activation d'un mode de mise au point..... R 6 - 17
6.6.4. Gestion des points d'arrêt ..... R 6 - 18
6.6.5. Commandes du mode interactif ..... R 6 - 18
6.6.6. Exemple..... R 6 - 20
6.7. Modification et visualisation de l'état courant .....R 6 - 23
6.8. Gestion automatique des espaces et des piles.....R 6 - 25

7. Extensions de Prolog avec des langages externes. ..... R 7 - 1
7.1. Principes des fonctions de communication de données.....R 7 - 2
7.2. Fonctions de communication de données simples .....R 7 - 3
7.2.1. Test du type d'un argument..... R 7 - 3
7.2.2. Transfert de données simples de Prolog vers un ..... autre langage..... R 7 - 4
7.2.3. Transfert de données simples d'un langage externe ..... vers Prolog ..... R 7 - 6
7.3. Fonctions de communication de termes quelconques.....R 7 - 8
7.3.1. Au moyen de chaînes de caractères..... R 7 - 8
7.3.2. Au moyen de structures de tableaux ..... R 7 - 10
7.3.2.1. Description du codage d'un terme..... R 7 - 11
7.3.2.2. Identificateurs..... R 7 - 14
7.3.2.3. Description des fonctions de communication R 7 - 16
7.4. Principe de la méthode des descripteurs.....R 7 - 17
7.4.1. Eléments descriptifs..... R 7 - 17
7.4.2. Déclaration statique ..... R 7 - 18
7.4.3. Déclaration dynamique..... R 7 - 19
7.5. Données partagées .....R 7 - 20
7.5.1. Exemple de zone commune de données..... R 7 - 22
7.6. Ajout de fonctions externes.....R 7 - 22
7.6.1. Exemple de déclaration de règle prédéfinie..... R 7 - 24
7.7. Ajout de fonctions externes à appel direct .....R 7 - 26
7.7.1. Primitive CallC ..... R 7 - 26
7.7.2. Conventions Prolog pour la communication ..... des données..... R 7 - 27
7.7.2.1. Convention pour des paramètres sans ..... valeur de retour ..... R 7 - 28
7.7.2.2. Convention pour le retour de la fonction..... R 7 - 29

7.7.2.3. Convention pour des paramètres avec valeur de retour ..... R 7 - 29
7.7.3. Exemple : méthode simple pour appeler une fonction C ..... R 7 - 32

8. Lancement d'un but Prolog par un programme C ..... R 8 - 1
8.1. Principes de base ..... R 8 - 1
8.2. Initialisation et terminaison de Prolog ..... R 8 - 4
8.3. Empilement d'un but Prolog ..... R 8 - 6
8.4. Programmation ..... R 8 - 7
8.5. Méthode simple d'appel d'un but Prolog ..... R 8 - 8
8.5.1. Description ..... R 8 - 8
8.5.2. Exemple ..... R 8 - 9
8.6 Autres fonctions ..... R 8 - 10

9. Interruptions ..... R 9 - 1
9.1. Concepts ..... R 9 - 1
9.2. Description des interfaces ..... R 9 - 3
9.3. Exemple complet ..... R 9 - 3

10. Extensions Edinburgh ..... R 10 - 1
10.1. Syntaxe ..... R 10 - 1
10.1.1. Généralités ..... R 10 - 1
10.1.2. Les opérateurs ..... R 10 - 2
10.2. Le contrôle ..... R 10 - 3
10.3. Manipulation des règles ..... R 10 - 5
10.4. Opérations prédéfinies sur les données ..... R 10 - 6
10.4.1. Les tests de type ..... R 10 - 6
10.4.2. Les opérations arithmétiques ..... R 10 - 7
10.4.3. Composition et décomposition d'objets ..... R 10 - 8
10.4.3. Comparaison de termes quelconques ..... R 10 - 11
10.5. Les entrées / sorties ..... R 10 - 12
10.5.1. Généralités ..... R 10 - 12
10.5.2. Entrées ..... R 10 - 14
10.5.2.1. Interprétation des chaînes de caractères ..... R 10 - 14
10.5.2.2. Prédicats ..... R 10 - 15
10.5.3. Sorties ..... R 10 - 17
10.6. L'environnement ..... R 10 - 19
10.7. Traduction de DCG ..... R 10 - 20

Introduction

Les besoins des utilisateurs de l'Intelligence Artificielle évoluent. En devenant industriels, les programmes sont de plus en plus gros, manipulent d'importantes quantités de données, communiquent avec d'autres logiciels.

Pour répondre à cette demande, PrologIA propose un nouveau système PrologII+. Tout en conservant les caractéristiques qui ont fait le succès de Prolog II, les *plus* offerts par ce produit sont:

- performances
- modularité
- standardisation
- ouverture vers l'extérieur
- environnement de programmation
- interface graphique

Standardisation

• **Choix** d'un **Mode standard** s'inspirant fortement des dernières décisions des groupes de normalisation, pour la norme ISO-Prolog.

Performances

• **Compilation incrémentale** parfaitement transparente, permettant de bénéficier de la vitesse d'un langage compilé sans avoir à renoncer à la souplesse d'un langage interpreté.

• **Optimisation de la récursivité terminale**, permettant de programmer récursivement, c'est à dire de la manière la plus naturelle en Prolog, les processus itératifs, et cela sans débordement de la mémoire.

• **Récupération dynamique de mémoire** (*garbage collector*) dans les espaces de travail de Prolog (piles, code, dictionnaire). Une technique nouvelle nous permet une récupération plus performante.

• **Réallocation dynamique des espaces** de travail de Prolog.

• **Compilation très optimisée** de certaines règles prédéfinies et notamment des règles arithmétiques.

• **Entiers en précision infinie**.

Modularité

- **Modularité**: nouvelle notion, proposée pour le standard.

- **L'écriture de gros programmes** est rendue possible grâce aux modules, compilés séparément, chargeables et déchargeables à volonté.

- **Run-time** permettant de diffuser des applications autonomes.

Ouverture vers l'extérieur

- **Liens bidirectionnels avec les autres langages** : possibilité d'appeler en Prolog des sous-programmes écrits dans d'autres langages. Possibilité d'appeler, depuis un programme quelconque, un programme écrit en Prolog (y compris avec l'obtention successive de différentes solutions). Tous les cas de figure d'appels croisés sont désormais pris en charge.

- **Communication** avec d'autres applications.

- **Structures de données entièrement ouvertes**, avec l'interface requis pour la communication inter-langage de tous les types de données, sans restriction sur la nature des termes (qui, par exemple, peuvent comporter y compris des variables et des identificateurs). Possibilité de partager des zones communes (tableaux) avec d'autres programmes.

- **Interface SQL** entre Prolog et les SGBD.

- **Manipulation de bases de faits conséquentes**.

- **Données numériques** (entiers et réels) homogènes avec les autres langages supportés par le système hôte, y compris pour leurs représentations étendues.

Environnement de programmation

- **Manipulation de règles** (*assert, clause/rule, list* etc…) intégrée au compilateur et fonctionnant sur les règles compilées.

- **Récupération d'interruptions** asynchrones en Prolog II+ autorisant, par exemple, la gestion d'environnements fortement interactifs (souris, fenêtres, etc…).

- **Debugger** de haut niveau permettant une mise au point rapide des gros programmes.

- **Editeur intégré** couplé avec la compilation incrémentale.

Interface graphique

- **Bibliothèque portable** entre divers environnements.

- **Définition d'objets**: fenêtres, menus, boutons, ….
- **Composition de dessins**.
- **Gestion de dialogues**.

Sur ce Manuel

La documentation concernant PROLOG II+ a été regroupée en deux manuels. Le premier, le *Manuel de référence*, décrit de manière précise le langage et son utilisation. Ce manuel est valable (en principe) pour toutes les implantations. Le second, le *Manuel d'utilisation*, décrit tout ce qui est dépendant d'un ordinateur et d'un système d'exploitation spécifique. Il y aura donc un manuel d'utilisation par matériel.

Pour installer le logiciel Prolog II+ et l'utiliser, voir les chapitres 1 et 2 du manuel utilisateur.

Le manuel de Référence commence par une introduction élémentaire à Prolog et son utilisation pratique. Dans le second chapitre, on présente de manière un peu plus théorique les bases de Prolog. Les chapitres suivants examinent les différentes règles prédéfinies concernant : le contrôle, la manipulation des règles, les opérations sur les données, les entrées-sorties, l'environnement, le debugger et l'éditeur. Les chapitres suivants concernent l'ouverture vers les autres langages : on décrit comment ajouter des règles prédéfinies écrites par l'utilisateur, faire des appels croisés ou déclarer des zones de données communes entre Prolog et d'autres langages, et intercepter des interruptions et déclencher un traitement Prolog. Enfin le dernier chapitre décrit la bibliothèque de primitives spécifiques à la syntaxe Edinburgh.

En plus de ce manuel, il y a une partie Annexe qui est valable pour toutes les implantations également. Elle détaille quelques points secondaires mentionnés dans le manuel de Référence. En particulier vous y trouverez les différences entre l'interpréteur Prolog II et le compilateur Prolog II+, ou bien quelques exemples de programmes pour illustrer l'utilisation de Prolog.

Le manuel d'Utilisation précise comment se réalisent de manière pratique sur votre ordinateur, certaines fonctions décrites dans le manuel de Référence. Cela concerne la structure des noms de fichiers, la gestion des états sauvés, l'utilisation de la mémoire et comment lancer Prolog. La seconde partie de ce manuel décrit ce qui est spécifique à cette version : ce qu'il y a en plus et en moins par rapport à la version de base, ainsi que les valeurs extrêmes des constantes.

Nous vous conseillons également de consulter les documents suivants :

Prolog

F. Giannnesini, H. Kanoui, R. Pasero, M. Van Caneghem,
InterEditions, Mars 1985.

Prolog, bases théoriques et développements actuels,
A. Colmerauer, H. Kanoui, M. Van Caneghem,
TSI vol 2, numéro 4, 1983.

Pour la programmation en syntaxe Edinburgh:

Prolog Programming for Artificial Intelligence.
Ivan Bratko, Addison-Wesley, International Computer Science Series, 1986.

Garantie et responsabilités

PrologIA n'offre aucune garantie, expresse ou tacite, concernant ce manuel ou le logiciel qui y est décrit, ses qualités, ses performances ou sa capacité à satisfaire à quelque application que ce soit.

PrologIA ne pourra être tenue responsable des préjudices directs ou indirects, de quelque nature que ce soit, résultant d'une imperfection dans le programme ou le manuel, même si elle a été avisée de la possibilité que de tels préjudices se produisent. En particulier, elle ne pourra encourir aucune responsabilité du fait des données mémorisées ou exploitées, y compris pour les coûts de récupération ou de reproduction de ces données.

L'acheteur a toutefois droit à la garantie légale dans les cas et dans la mesure seulement où la garantie légale est applicable nonobstant toute exclusion ou limitation.

Droits d'auteur

0. Débuter avec Prolog

0.1. Leçon 1 : Lancement de Prolog
0.2. Leçon 2 : Utilisation d'un programme d'exemple
0.3. Leçon 3 : L'Editeur
0.4. Les erreurs les plus courantes

Dans les exemples de ce manuel, ce qui est écrit par l'utilisateur apparait en *caractère penché*, et tout ce qui est écrit par l'ordinateur apparait en caractère Télétype droit.

0.1. Leçon 1 : Lancement de Prolog

Lancez Prolog, ceci le fait démarrer à partir d'un état sauvé initial nommé *initial.po*. Le caractère ">" représente le prompt de Prolog. Les différentes possibilités de démarrage sont décrites dans le manuel d'Utilisation. Prolog affiche alors sur l'écran :

```
Prolog II+ Version...
Code : ... PrologIA
>
```

Le caractère ">" indique à l'utilisateur qu'il est sous Prolog et que le système est en attente de commande. Pour commencer, essayons une commande simple, imprimons le message "Hello World":

```
> out("Hello World") line;
"Hello World"
{}
>
```

Entrons maintenant un premier petit programme; pour cela on se met en mode insertion de règles (commande *insert*). Le système est alors prêt à lire des règles (ou des commentaires) et les enregistre en mémoire au fur et à mesure. Par exemple, on entre trois règles qui expriment que Pierre, Jean et Marie habitent respectivement à Paris, Marseille et Paris :

```
> insert;
habite_a(Pierre, Paris)->;
habite_a(Jean, Marseille)->;
habite_a(Marie, Paris)->;;
```

```
{}
>
```

Remarquez que chaque règle est terminée par le caractère ";" et que l'insertion se termine par un ";" supplémentaire. Les règles sont insérées dans l'ordre où on les a tapées. On s'en assure en tapant la commande *list* (qui affiche les règles du module courant).

```
>list;
habite_a(Pierre, Paris)  -> ;
habite_a(Jean, Marseille) -> ;
habite_a(Marie, Paris) -> ;

{}
>
```

Faisons quelques essais avec ce programme : Où habite Pierre ?

```
>habite_a(Pierre, x);
{x=Paris}
>
```

Qui habite à Paris ?

```
>habite_a(x, Paris);
{x=Pierre}
{x=Marie}
>
```

Qui habite où ?

```
>habite_a(x, y);
{x=Pierre, y=Paris}
{x=Jean, y=Marseille}
{x=Marie, y=Paris}
>
```

A chaque fois, la réponse du système est l'ensemble des valeurs à donner aux variables figurant dans la question pour que la relation correspondante soit satisfaite. Pour terminer la session, on tape la commande :

```
>quit;
Bye......
```

et on se retrouve sous l'interpréteur de commandes du système d'exploitation.

0.2. Leçon 2 : Utilisation d'un programme d'exemple

Dans ce qui suit, on utilisera le programme *menu.p2* qui se trouve décrit dans de nombreuses publications. Ce programme se trouve dans le répertoire d'exemples du Kit Prolog II+. Pour cette leçon, il vous faut au préalable recopier le fichier *menu.p2* dans votre répertoire courant. Il suffit alors de lancer Prolog, puis insérer le fichier *menu.p2* en tapant :

```
PROLOG II+ ...
> echo insert("menu.p2");
```

insert lit des règles sur le fichier spécifié et les insère dans le module courant. Lorsque le fichier d'entrée est épuisé, l'entrée courante bascule sur le clavier qui est l'unité d'entrée par défaut. La primitive *echo* réalise l'affichage à la console au fur et à mesure de la lecture du fichier:

```
" la carte "

hors_d_oeuvre(Artichauts_Mélanie) -> ;
hors_d_oeuvre(Truffes_sous_le_sel) -> ;
hors_d_oeuvre(Cresson_oeuf_poche) -> ;

viande(Grillade_de_boeuf) -> ;
viande(Poulet_au_tilleul) -> ;

poisson(Bar_aux_algues) -> ;
poisson(Chapon_farci) -> ;

dessert(Sorbet_aux_poires) -> ;
dessert(Fraises_chantilly) -> ;
dessert(Melon_en_surprise) -> ;

" plat de résistance"

plat(p) -> viande(p) ;
plat(p) -> poisson(p) ;

" composition d'un repas "

repas(e,p,d) -> hors_d_oeuvre(e) plat(p) dessert(d) ;

" valeur calorique pour une portion "

calories(Artichauts_Mélanie,150) -> ;
calories(Cresson_oeuf_poche,202) -> ;
calories(Truffes_sous_le_sel,212) -> ;
calories(Grillade_de_boeuf,532) -> ;
calories(Poulet_au_tilleul,400) -> ;
calories(Bar_aux_algues,292) -> ;
calories(Chapon_farci,254) -> ;
calories(Sorbet_aux_poires,223) -> ;
calories(Fraises_chantilly,289) -> ;
calories(Melon_en_surprise,122) -> ;

" valeur calorique d'un repas "

valeur(e,p,d,v)  ->
   calories(e,x)
   calories(p,y)
   calories(d,z)
   ajouter(x,y,z,v) ;

" repas équilibré "

repas_equilibre(e,p,d) ->
   repas(e,p,d)
   valeur(e,p,d,v)
   inferieur(v,800) ;

" divers"

ajouter(a,b,c,d) ->
   val(add(a,add(b,c)),d) ;

inferieur(x,y) -> val(inf(x,y),1) ;
{}
>
```

Faisons quelques essais avec ce programme. La question : «Quels sont les plats ?» se traduit par :

```
>plat(p);
{p=Grillade_de_boeuf}
{p=Poulet_au_tilleul}
{p=Bar_aux_algues}
{p=Chapon_farci}
```

Quels sont les hors-d'oeuvres ?

```
>hors_d_oeuvre(x);
{x=Artichauts_Mélanie}
{x=Truffes_sous_le_sel}
{x=Cresson_oeuf_poche}
```

Cette leçon s'arrête là et on sauve le module courant "" (module avec le préfixe ""):

```
> save([""],"menu.mo") quit;
Bye........
```

Note: On peut utiliser également la commande *exit*: en sortant Prolog sauve l'état courant dans un fichier nommé par défaut *prolog.po*.

Le module sauvé nous servira dans la leçon suivante.

0.3. Leçon 3 : L'Editeur

Le but de cette leçon est de montrer une façon très simple de travailler avec l'éditeur intégré à Prolog, au niveau d'un paquet de règles. Sur les machines possédant un environnement fenêtrage et souris, l'utilisation de copier/coller peut être plus commode.

On commence par lancer Prolog et charger le module sauvé contenant le programme *menu.p2* si l'on a sauvé avec *save*:

```
...
> load("menu.mo");
{}
>
```

Si l'on est sorti avec *exit*, on commence par lancer Prolog avec l'état sauvé *prolog.po* contenant le programme *menu.p2* tel que nous l'avons laissé à la fin de la leçon précédente:

```
....
>
```

Supposons que l'on veuille changer notre menu, et remplacer les règles :

```
viande(Grillade_de_boeuf) -> ;
viande(Poulet_au_tilleul) -> ;
```

par :

```
viande(Veau_marengo) -> ;
```

```
viande(Poulet_roti) -> ;
```

Appeller l'éditeur par la commande :

```
> edit(viande/1);
```

où la notation *viande/1* désigne le paquet de règles concernant la relation *viande* avec *1* argument.

L'éditeur est activé et montre les règles définissant la relation *viande*. Réaliser les modifications voulues, et sortir de l'éditeur normalement. Vérifiez que Prolog a bien relu les nouvelles définitions:

```
> list(viande/1);
viande(Veau_marengo) -> ;
viande(Poulet_roti) -> ;
{}
```

On exécute le programme modifié et on finit par *quit* puisqu'on ne désire pas conserver l'état courant.

```
> viande(p) ;
{p=Veau_Marengo}
{p=Poulet_roti}
> plat(p) ;
{p=Veau_Marengo}
{p=Poulet_roti}
{p=Bar_aux_algues}
{p=Chapon_farci}
> quit ;
Bye......
```

0.4. Les erreurs les plus courantes

Voici quelques explications sur les erreurs les plus courantes qui peuvent se produire lors de l'utilisation de Prolog II+.

0.4.1. Débordement de l'espace de code.

•PAS ASSEZ DE PLACE POUR LE CODE
Il faut redémarrer Prolog avec un espace plus grand pour le code (option -c de la ligne de commande). Si le système de réallocation automatique n'est pas désactivé au démarrage de Prolog, un certain nombre de réallocations se produiront avant le débordement. Le code occupera alors toute la place disponible de la machine. Dans ce cas il faut, si c'est possible, augmenter la mémoire totale de Prolog ou bien vérifier le fonctionnement du programme, sans doute anormal.

0.4.2. Interruption.

Un programme en cours d'exécution peut être interrompu à tout instant par un caractère d'interruption (<Control-C> <return> souvent).

```
> insert;
```

```
rr -> rr;;
{}
> rr;
<Ctrl-C>
INTERRUPTION UTILISATEUR
{}
>
```

0.4.3. Erreurs détectées en cours d'insertion.

•REGLE DEJA DEFINIE
Cette erreur provient du fait que l'utilisateur n'a pas respecté le principe suivant : Toutes les règles de même nom et de même arité doivent être consécutives (deux règles de même nom/arité ne peuvent être séparées ni par un commentaire, ni par une règle de nom/arité différent). Les configurations suivantes sont donc illégales ;

```
qq(x) -> ;
rr(y) -> ;
qq(z) -> ;
```

car la règle *rr* est mal placée.
ATTENTION: cette erreur ne se produira pas si vous utilisez les primitives *reinsert ou insertz*. En effet *reinsert* écrase le paquet de règles existant par sa nouvelle définition et *insertz* complète le paquet de règles. En particulier, en utilisant *reinsert* dans le cas décrit ci-dessus, il n'y aura aucune erreur et le paquet de règles d'accés *qq* contiendra une seule règle, la dernière.

0.4.4. Erreurs concernant les fichiers.

•FIN DE FICHIER
Ce message signale que la lecture du fichier d'entrée est terminée. C'est plutôt un avertissement qu'un message d'erreur. L'entrée bascule alors automatiquement sur le clavier.

•ERREUR D'OUVERTURE DU FICHIER D'ENTREE
Ce message signale que le fichier n'existe pas, ou qu'il est inaccessible. Il suffit de refaire la commande après avoir apporté les corrections nécessaires.

0.4.5. Erreurs de syntaxe.

La plupart des erreurs de syntaxe sont suffisamment explicites pour qu'on voit tout de suite de quoi il s'agit. Il suffit de se reporter à la syntaxe (Chapitre 1) pour avoir des explications complémentaires. Cependant, certaines d'entre elles méritent un commentaire :

•UNITE MAL PLACEE

```
> insert;
plat(x) -> viande(, boeuf) ;;

-> "," : UNITE MAL PLACEE
```

Cette erreur montre que la syntaxe n'a pas été respectée. Dans cet exemple, il manque un argument.

•IL MANQUE " A LA FIN DE LA CHAINE

```
> insert;
viande("boeuf)

-> MANQUE " A LA FIN DE LA CHAINE.
```

Une chaîne doit tenir sur une ligne, si la fin de ligne n'est pas masquée. On rappelle qu'une chaîne commence et finit sur le caractère("). Si le caractère fait partie de la chaîne, il doit être doublé.

•CE TERME NE PEUT PAS ETRE UNE TETE DE REGLE

```
> insert;
p(x) -> ;;

-> v41(v60) : CE TERME NE PEUT PAS ETRE UNE TETE DE REGLE
```

Le seul accès légal à une règle est un identificateur (il commençe par *deux* lettres en syntaxe Prolog II). En particulier, l'accès ne peut être une variable (dont le nom commence par *une seule* lettre en syntaxe Prolog II), ni un nombre, ni une chaîne.

On rappelle, pour la syntaxe Prolog II, qu'un nom de variable, de relation, ou de constante est une suite de mots séparés par un "_" et éventuellement suivie d'apostrophes. Un mot est constitué d'une ou plusieurs lettres éventuellement suivies de chiffres. Si le premier mot du nom commence par au moins deux lettres, on a affaire à un nom de relation ou de constante (ce qu'on appelle un *identificateur*), ce qui constitue un accès légal à une règle. Dans le cas contraire on a affaire à une *variable*. Par exemple :

x, *x12*, *x_toto_a25*, *a'*, *a12''*, *b_titi''*

sont des noms de variables.

bonjour, *xx''*, *qq_toto_25*, *comment_allez_vous, 'avec des blancs'*

sont des noms de relations ou de constantes.

•LITTERAL INCORRECT

```
> insert;
humain(x) ->
  homme(x)
humain(x) -> femme(x);

-> LITTERAL INCORRECT
skipping:  femme(x);
```

Lorsqu'une telle erreur se produit au niveau de la flèche Prolog, il s'agit très souvent de l'oubli du ";" à la fin de la règle précédente.

0.4.6. Erreurs d'exécution.

•CE TERME NE PEUT ETRE UN BUT
Voici un exemple qui utilise la règle : *exec(p) -> p* ;

```
> exec(outm("bonjour"));
bonjour{}
>
```

La variable *p* de la règle exec est remplacée par *outm("bonjour")* qui est effacé sans problème. En revanche :

```
> exec(x);
-> CE TERME NE PEUT ETRE UN BUT

>
```

provoque une erreur car la variable *p* reste libre au moment où on tente de l'effacer.

•ARGUMENT DE MAUVAIS TYPE
Cette erreur survient lorsque l'on essaie d'effacer une règle prédéfinie avec un des arguments qui n'est pas du type attendu. Généralement l'argument qui est en cause est affiché avec le message.

•APPEL A UNE REGLE NON DEFINIE
On essaye d'effacer une règle qui n'est pas définie alors qu'on est en mode erreur (flag uE). Notez que pour faire échouer volontairement l'exécution, il faut appeler la règle prédéfinie *fail.*

Il existe plusieurs comportement possibles, pilotés par un flag dans la commande de lancement de Prolog, pour l'effacement d'un prédicat non défini:

uW : affiche un warning et continue l'exécution du programme,
uF : échoue,
uE : génère l'erreur.

Les autres erreurs d'exécution concernent principalement l'utilisation des primitives *block*, *block_exit* et de *val*. En général les diagnostics sont suffisamment explicites.
Il est conseillé de lancer Prolog avec l'option *-f uE* qui permet d'avoir un diagnostic précis lors de l'appel erroné d'une règle n'existant pas (à cause d'une faute de frappe par exemple). Lorsque l'on veut appeler une règle et avoir un échec lorsque celle-ci n'existe pas, tester d'abord sa présence avec *current_predicate*.

1. Eléments de base

1.1. Notations utilisées
1.2. Jeu de caractères
1.3. Les variables
1.4. Constantes
1.5. Termes et arbres
1.6. Les opérateurs
1.7. Les règles et les assertions
1.8. Les mécanismes de base de Prolog
1.9. La syntaxe complète de Prolog II+
1.10. Le macroprocesseur

1.1. Notations utilisées

Dans ce manuel nous utiliserons des règles hors contexte pour décrire la syntaxe de toutes les formules intervenant dans Prolog. La notation utilisée ici est la même que celle utilisée par la commission de normalisation de Prolog :

- Les symboles non terminaux sont représentés par des identificateurs et les terminaux sont entre doubles quotes "…".
- Le symbole "=" est choisi comme symbole de réécriture et une règle se termine par un ";". Les éléments d'un membre droit de règle sont séparés par des ",".
- Les accolades {…} représentent un nombre quelconque d'apparitions, éventuellement aucune, des éléments qu'elles encadrent.
- Les crochets […] signalent le caractère optionnel des éléments qu'ils encadrent.
- La barre verticale ...|... exprime l'alternative entre les éléments qu'elle sépare. Des parenthèses (...|...) sont utilisées lorsque le résultat de cette alternative apparaît dans une liste de symboles.
- Le signe "-" est utilisé pour représenter des exceptions.
- Certaines règles de réécriture dépendent des options de syntaxe sélectionnées : les règles dont le membre gauche est annoté par P ne sont actives que lorsque la syntaxe Prolog II est activée (par exemple `separator`^{P}). Les règles dont le membre gauche est annoté par E ne sont actives que lorsque la syntaxe Edinburgh est activée (par exemple `graphic_char`^{E}).

- D'autres règles, annotées par une lettre, dépendent des options choisies au lancement de Prolog et ne sont valides que dans certains cas.

La totalité de la syntaxe de Prolog est donnée à la fin de ce chapitre. Des extraits plus ou moins simplifiés de cette syntaxe sont commentés dans les paragraphes suivants.

1.2. Jeu de caractères

L'utilisateur peut choisir parmi deux jeux de caractères : celui défini par le code ISO 8859-1 (cf. Annexe), ou celui disponible sur la machine hôte (cf. Manuel Utilisateur). Dans ce manuel nous décrirons uniquement le jeu ISO. Le choix du jeu hôte aura pour effet de diminuer ou de grossir les sous-ensembles *letter* et *graphic_char* avec des caractères n'existant pas dans le jeu ISO (cf. U3-2).

Voici une description simplifiée (la description complète se trouve au paragraphe 1.9) du jeu de caractères nécessaires à Prolog :

letter =	"A" | … | "Z" | "a" | … | "z" | "À" … "ß" - "×" | "à" … "ÿ" - "÷"
letter^{I} =	"\", accent_escape ;
digit =	"0" | … | "9" ;
alpha =	letter | digit | "_" ;
separator =	"(" | ")" | "[" | "]" | "{" | "}" | "|" | "," ;
separator^{P}=	";" | "." | "<" | ">" ;
special_char=	"%" | "'" | "" | "_" | "!" | "`" ;
special_char^{E}=	";" ;
graphic_char = ":"	"#" | "$" | "&" | "*" | "+" | "-" | "/" | | "=" | "?" | "\" | "@" | "^" | "~" | NBSP[1] … "¿" | "×" | "÷" ;
graphic_char^{E} =	"." | "<" | ">" ;
character =	letter | digit | separator | graphic_char | special_char ;

Sans changer le sens de ce qui est écrit, on peut insérer des espaces n'importe où, sauf dans les constantes et les variables. On peut enlever des espaces n'importe où sauf dans les chaînes et sauf si cette suppression provoque la création de nouvelles constantes ou variables, par juxtaposition d'anciennes.

---

1 NBSP est l'espace non sécable.

Il existe diverses manières d'introduire des commentaires dans les programmes. Du point de vue de la syntaxe, un commentaire équivaut à un blanc et peut figurer partout où un blanc est permis :

• Le caractère "%" indique le début d'un commentaire qui s'étend depuis le "%" jusqu'à la fin de la ligne où ce commentaire apparaît.

• Les symboles "|*", "*|" et "/*", "*/" servent aussi à définir des commentaires, constitués par ces symboles et la suite des caractères qu'ils encadrent.

On ne doit pas mélanger ces symboles : un commentaire commençant par "|*" doit se terminer par "*|" et un commentaire commençant par "/*" doit se terminer par "*/". D'autre part, on peut imbriquer de tels commentaires; par exemple, le texte suivant sera vu comme un unique commentaire :

|* *second com-* |* *premier commentaire* *| *mentaire* *|

• Une chaîne de caractères écrite au niveau supérieur, c'est-à-dire là où une règle est attendue, possède aussi la valeur d'un commentaire.

1.3. Les variables

Les variables servent aussi bien à désigner des constantes que des entités plus complexes. En voici la syntaxe :

```
variable =          "_" , {alpha} ;
variable =          extended_var ;

extended_var P =    letter, [ (digit | "_"), {alpha} ] , { "'" } ;
extended_var E =    big_letter, [ {alpha} ] ;
```

Il y a donc, pour les variables, une syntaxe de base et une syntaxe étendue à choisir parmi deux : la syntaxe Prolog II et la syntaxe anglaise. On doit indiquer si l'on désire une syntaxe différente de la syntaxe étendue Prolog II, au moment du lancement de Prolog II+; de plus, puisque ces deux syntaxes de variables sont incompatibles[2], on ne peut pas avoir les deux extensions en même temps.

2 En effet : l'expression *x* est une variable en Prolog II et ne l'est pas dans la syntaxe anglaise. De même, le nom *Pierre* n'est pas une variable en Prolog II, mais en est une en syntaxe anglaise !

Voici quelques exemples de variables correctes :

Syntaxe Prolog II	Syntaxe Edinburgh
x	*X*
x' X_12_plus	
x' '	*Prix*
x12	*_prix*
p_rix	*X1Y2*
y33'	*_33*
y_en_a'	
_prix	
_123	

et quelques exemples d'expressions qui ne sont pas des variables correctes :

Syntaxe Prolog II	Syntaxe Edinburgh
ph	*X'*
xx	*x12*
prix	*prix*
1er_x	*y_en_a*

1.4. Constantes

Les données les plus simples sont des constantes. Elles peuvent être de quatre types : les identificateurs, les nombres entiers, les nombres réels, et les chaînes de caractères.

```
identifier =          prefix , prefix_limit , abbreviated_id ;
identifier =          abbreviated_id ;

identifierE =         prefix , prefix_limit ,
                      """, graphic_symbol, """ ;
identifierE =         graphic_symbol ;

abbreviated_id =      name - extended_var ;
abbreviated_id =      "'", { (character - "'") | "''" }, "'";

prefix_limit =        ":" ;
prefix =              [ name , { prefix_limit , name } ] ;

name =                letter , { alpha } ;

integer_number =      digits ;
integer_number =      "0b", binary_number ;
integer_number =      "0o", octal_number ;
integer_number =      "0x", hex_number ;
integer_number =      "0'", character ;

real_number =         digits, ".", digits, ("E"|"e"|"D"|"d"),
                      [ ["+"|"-"], digits];
real_numberS =        digits, ".", digits,
                      [ ("E"|"e"|"D"|"d"), [ ["+"|"-"],digits]
];

binary_number =       binary_digit , { binary_digit } ;

octal_number =        octal_digit , { octal_digit } ;

hex_number =          hex_digit , { hex_digit } ;

digits =              digit , { digit } ;

string =              "" , { quoted_char } , "" ;

quoted_char =         character - ( "" | "\" | newline) | """ ;
quoted_chari0 =       "\" ;
quoted_chari1 =       "\", format_escape ;

format_escape =       "b" | "f" | "n" | "r" | "t" | "\" | newline
                      | octal_digit, octal_digit, octal_digit
                      | ("x" | "X"), hex_digit, hex_digit ;
```

Identificateurs

Les identificateurs ont deux représentations externes : une représentation complète et une représentation abrégée. La première comporte un préfixe qui spécifie la famille dont l'identificateur fait partie; dans la représentation abrégée, le préfixe n'apparaît pas et il est supposé que certaines *conventions de lecture-écriture* précisent de manière non ambiguë quel est le préfixe de l'identificateur. Ces notions sont expliquées en détail au chapitre 3.

C'est la présence ou l'absence du caractère ":" qui distingue la représentation complète de la représentation abrégée d'un identificateur. Ce caractère peut être redéfini et remplacé par un caractère graphique, tel que le décrit le chapitre 3.

Les identificateurs complets suivants sont corrects et représentent tous des identificateurs différents:

data:peter	*grammar:singular*
x:peter	*lexicon:name*
sys:write	*sys:env:files*
:peter	*grammar:plural*

Note: La chaîne vide est un préfixe légal, l'identificateur *:peter* est donc correct syntaxiquement.

L'identificateur suivant n'est pas complet

peter

La syntaxe de la représentation abrégée des identificateurs et celle des variables sont très voisines et, ensemble, définissent ce que dans beaucoup d'autres langages de programmation on appelle «identificateur». Nous retiendrons que, dans la syntaxe Prolog II, les variables commencent par *une seule lettre* ou par le caractère "_", tandis que les représentations abrégées d'identificateurs commencent par au moins *deux* lettres. Dans la syntaxe Edinburgh, la différenciation se fait sur le premier caractère, qui doit être une lettre *majuscule* ou le caractère "_" pour représenter une variable ou bien une lettre *minuscule* pour représenter un identificateur en notation abrégée. Dans cette syntaxe, les représentations abrégées d'identificateurs peuvent également être des suites de caractères graphiques.

Voici des exemples d'identificateurs abrégés corrects :

Syntaxe Prolog II	Syntaxe Edinburgh
pomme	*i10*
pomme'	*pomme*
pomme12	***
	§

des exemples d'identificateurs incorrects :

Syntaxe Prolog II	Syntaxe Edinburgh
x	*Pomme*
1er_lapin	*1er_lapin*
y_en_a	*pomme'*
l'herbe	

Nombres

La syntaxe des nombres en Prolog II+ est sensiblement la même que dans la plupart des langages de programmation.

Les nombres entiers sont signés et permettent de représenter, a priori, des valeurs quelconques. Plusieurs syntaxes d'entiers sont acceptées.

Un entier peut être exprimé dans les bases suivantes : 2, 8, 10, 16. Pour cela, la mantisse de l'entier sera préfixée respectivement par : 0b, 0o, 0, 0x. Par défaut une mantisse non préfixée sera considérée en base décimale. Les petits entiers, inférieurs à 256, pourront également être exprimés à l'aide du préfixe 0' suivi d'un caractère, la valeur de l'entier sera alors le code du caractère[3] . Par exemple, les expressions suivantes représentent des entiers :

Expression:	Valeur:
0b110	6
0o110	72
0110	110
0x110	272
0'A	65

On notera que, contrairement à d'autres langages, les nombres réels *doivent* comporter un exposant explicite : l'expression -12.34 ne définit pas un nombre réel, mais une paire pointée formée des deux entiers -12 et 34. En revanche, -12.34e0 est un réel correct. C'est le choix par défaut en syntaxe Prolog II, il est néanmoins possible, pour se ramener à une syntaxe standard, de le modifier par une option sur la ligne de commande au lancement de Prolog (cf. § 2.3 du manuel d'utilisation).
Les nombres réels sont codés en double précision, cela correspond au type *double* du standard IEEE 64 bits. La lettre introduisant l'exposant peut être une des suivantes : e, E, d ou D. On préférera toutefois e et E.

---

3 Attention aux caractères étendus, leur valeur dépend du mode choisi : ISO 8859-1 ou code de la machine hôte.

Chaînes de caractères

Les chaînes de caractères sont encadrées par des doubles quotes "". Tous les caractères imprimables peuvent figurer dans une chaîne sans précaution particulière, sauf le caractère "" qui doit être doublé et éventuellement le caractère "\" qui doit être doublé si l'interprétation de la lecture du "\" est active (cf. les options de comportement § U 2.3.); par exemple, la chaîne:

```
"il faut traiter "" et \\ avec précaution"
```

est correcte.

De façon générale le caractère "\" ne doit pas forcément être doublé: lorsque l'interprétation de lecture du "\" est active (cf. § 2.3 du manuel d'utilisation), si avec les caractères qui le suivent, il forme une expression (séquence escape) représentant un autre caractère, il faut le doubler; dans tous les autres cas, il n'est pas utile de le doubler:

```
"Utiliser \  suivi de RETURN pour ignorer une fin de ligne"
"Les codes hexadécimaux doivent commencer par \\x"
```

Quand l'interprétation du "\" est active (règles annotées par i1), plusieurs expressions commençant par "\" permettent de représenter un caractère.

Les expressions suivantes permettent de spécifier certains caractères non imprimables dans la définition d'une chaîne :

- `\b` espace arrière (backspace)
- `\f` saut de page (form feed)
- `\n` saut à la ligne (newline). Lors de son écriture, ce caractère est, le cas échéant, remplacé par le(s) caractère(s) requis pour obtenir une nouvelle ligne sur l'unité de sortie.
- `\r` retour en début de ligne (carriage return)
- `\t` tabulation

D'autres expressions permettent de représenter des caractères qui n'existent pas sur la machine hôte (voir le paragraphe 1.9.5).

Au moyen du nombre qui est son code interne, tout caractère peut être spécifié dans une chaîne. Par exemple, les expressions "\033" et "\x1B" définissent toutes deux une chaîne composée de l'unique caractère dont le code est 27. Le caractère nul (i.e.: celui dont le codage interne est le nombre 0) ne doit pas figurer dans les chaînes de caractères.

Une chaîne de caractères peut s'étendre sur plusieurs lignes. Pour cela, le dernier caractère de chacune des lignes en question sauf la dernière doit être "\" ; le "\" et le caractère de fin de ligne seront ignorés. Par exemple, l'expression

"ceci est une chaî\
ne sur deux lignes"

définit la même chaîne que "ceci est une chaîne sur deux lignes".

En Prolog, le type chaîne de caractère existe et une chaîne de caractères représente donc un objet de ce type. En syntaxe Edinburgh, une option de démarrage permet de définir l'interprétation syntaxique de cette unité lexicale, qui peut représenter un des termes suivants: un identificateur, une liste d'identificateurs d'une lettre, une liste d'entiers, une chaîne Prolog.

N.B. : En écrivant des programmes Prolog, il arrive qu'une donnée symbolique puisse être représentée soit par une chaîne de caractères, soit par un identificateur. Les identificateurs étant eux-mêmes codés sous forme d'entités atomiques, on admet généralement qu'ils constituent une représentation des objets symboliques plus efficace que les chaînes.

1.5. Termes et arbres

Toutes les données manipulées en Prolog sont des arbres éventuellement infinis dont nous allons tout d'abord donner une description informelle. Ces arbres sont formés de nœuds étiquetés :

- soit par une constante et, dans ce cas, ils n'ont aucun fils,
- soit par le caractère "point" et, dans ce cas ils ont deux fils,
- soit par "<>" ou"<->" ou "<-->" ou "<--->" ou… et, dans ce cas, le nombre de traits d'union correspond au nombre de leurs fils.

La figure suivante présente deux exemples d'arbres finis :

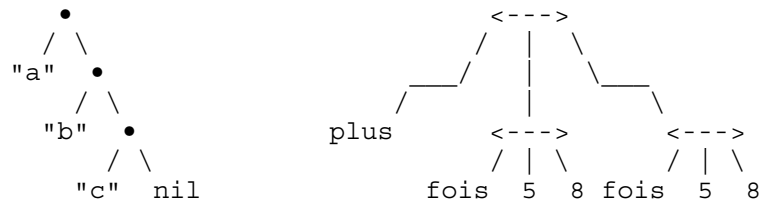


Figure 1.1

La figure 1.2 est un exemple d'arbre infini :

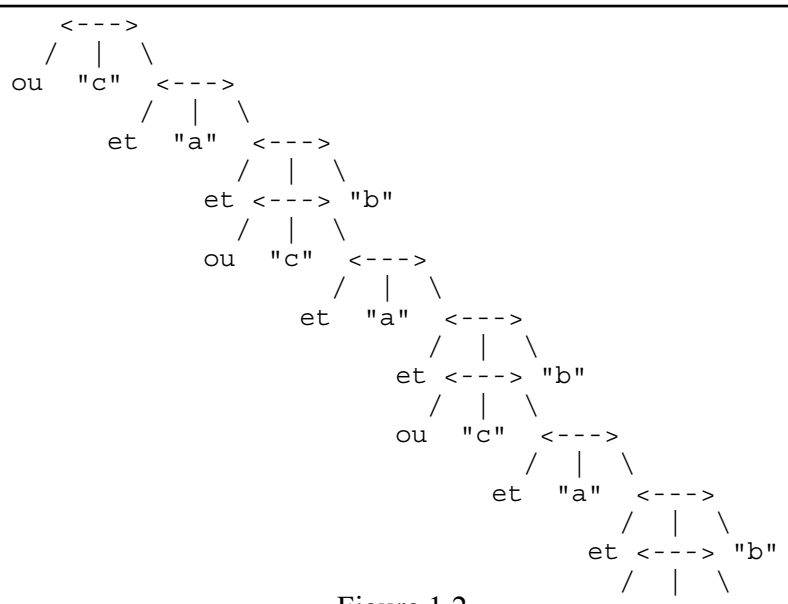


Figure 1.2

Remarquons que le dessin des arbres peut être allégé comme le montre la figure 1.3.

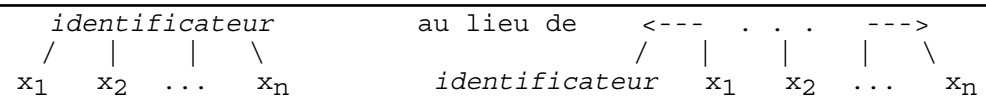


Figure 1.3

Bien entendu cette simplification présuppose que *n* ne soit pas nul. Les deux derniers arbres peuvent alors être représentés sous la forme classique de :

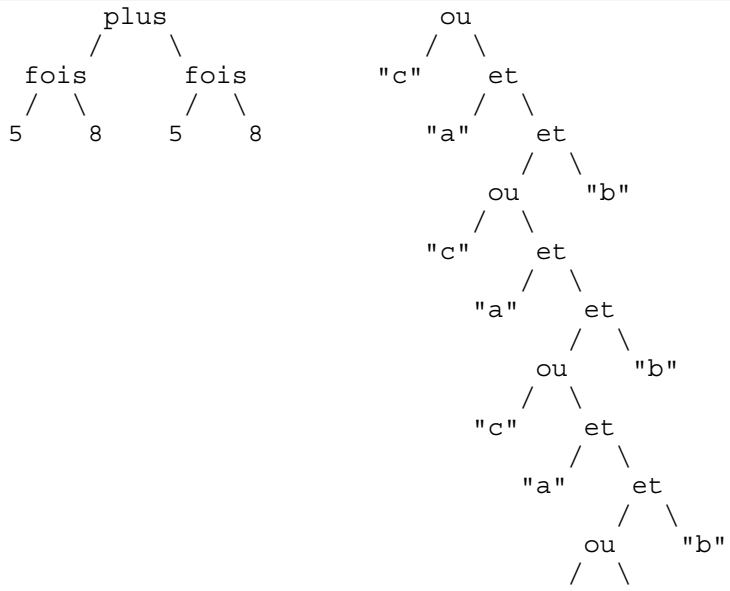


Figure 1.4

La notion d'arbre infini est suffisamment inhabituelle pour que nous nous étendions un peu dessus. Intuitivement un arbre est infini s'il possède une branche infinie. Nous nous intéresserons plus spécialement à la fraction des arbres infinis qui ensemble avec les arbres finis forme les arbres dits *rationnels* : c'est à dire les arbres qui ont un nombre fini de sous-arbres. Si nous reprenons les deux derniers exemples d'arbres l'ensemble de leurs sous-arbres est décrit dans la figure 1.5.

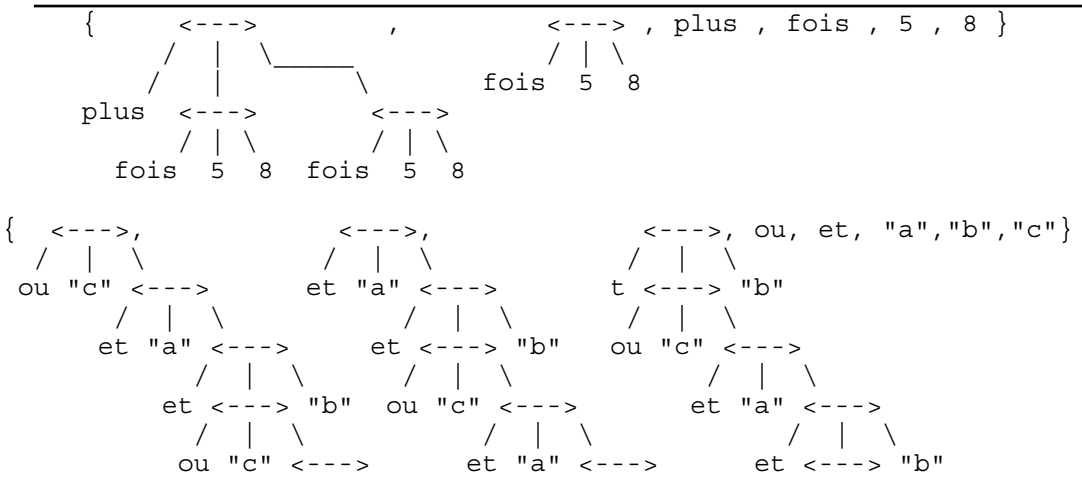


Figure 1.5

Ces ensembles étant finis, il s'agit donc d'arbres rationnels. Le fait qu'un arbre rationnel contienne un ensemble fini de sous-arbres donne un moyen immédiat de le représenter par un diagramme fini : il suffit de fusionner tous les nœuds d'où partent les mêmes sous-arbres (figure 1.6).

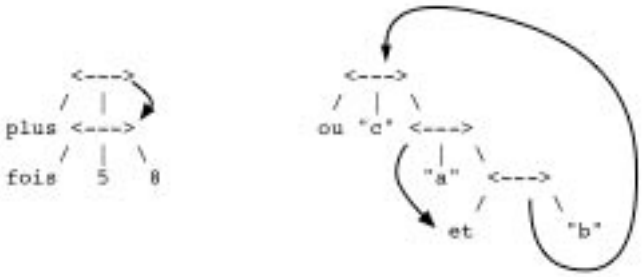


Figure 1.6

Si on ne fait pas toutes les fusions on obtient la figure 1.7.

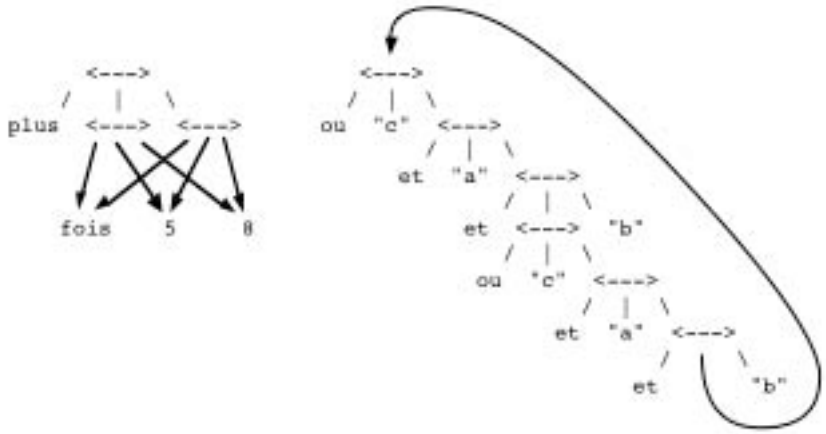


Figure 1.7

Il faut donc se méfier du fait que des diagrammes différents puissent représenter le même arbre.

Pour représenter les arbres nous utiliserons des formules appelées *termes*. Nous introduirons tout d'abord la notion de *terme strict* :

strict_term =	variable | identifier | constant | "[ ]" | "< >" ;
strict_term^P =	"(", strict_term , "." , strict_term , ")" ;
strict_term^E =	"[", strict_term , "|" , strict_term , "]" ;
strict_term^P =	"<", strict_term , { "," , strict_term } , ">" ;
strict_term^E =	"<>(", strict_term , { "," , strict_term } , ")" ;

Les termes stricts sont les "vrais" termes. Cependant pour des raisons de commodité on étend la syntaxe des termes stricts (sans en altérer le sens) en permettant :

- ^{P} d'ajouter et d'enlever des parenthèses mais en convenant que: $t_1.t_2. \ldots .t_n$ représente $(t_1.(t_2.(\ldots .t_n) \ldots)))$;

- ^{E} d'ajouter et d'enlever des crochets mais en convenant que: *[t1,t2,...,tn]* représente *[t1 | [t2 | ...[tn | []]]]* et *[t1,t2,...,tn| t]* représente *[t1 | [t2 | ...[tn | t]]];*

- d'écrire $id(t_1, t_2, \ldots , t_n)$ au lieu de $<id, t_1, t_2, \ldots , t_n>$ à condition que *id* soit un identificateur et que *n* soit différent de 0.

Ceci conduit à une notion plus générale de *terme* :

aiment(Pierre.Paul.Jean.nil, pere_de(x))

et le terme strict correspondant est :

<aiment, (Pierre.(Paul.(Jean.nil))), <pere_de, x>>

Pour transformer un terme en un arbre il faut affecter ses variables par des arbres, d'où la notion d'*affectation sylvestre*, qui est un ensemble X de la forme :

X = { x_1 := r_1, x_2 := r_2, … }

où les x_i sont des variables distinctes et les r_i des arbres. Nous introduirons aussi la notation suivante : si r_1, r_2 sont des arbres et si r_1, r_2, … , r_n est une suite de n arbres alors (r_1.r_2) et <r_1, r_2, … , r_n> représentent respectivement les arbres suivants :

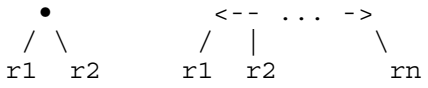


Figure 1.8

Si t est un terme strict faisant intervenir un sous-ensemble de l'ensemble des variables de l'affectation sylvestre X = { x_1 := r_1, x_2 := r_2, … } alors l'expression t/X désignera l'arbre obtenu en remplaçant les variables x_i par les arbres correspondants r_i. Plus précisément :

- $t/X = r_i$ si $t = x_i$;

- t/X = valeur de k si t est la constante k ;

- $t/X = (t_1/X \,.\, t_2/X)$ si $t = (t_1 \,.\, t_2)$;
- $t/X = [t_1/X \mid t_2/X)]$ si $t = [t_1 \mid t_2]$;

- $t/X = <t_1/X, \ldots, t_n/X>$ si $t = <t_1, \ldots, t_n>$.
- $t/X = <>(t_1/X, \ldots, t_n/X)$ si $t = <>(t_1, \ldots, t_n)$.

Si t_1 et t_2 sont des termes alors les formules $t_1 = t_2$ et $t_1 \neq t_2$ sont respectivement une *équation* et une *inéquation*. Un ensemble S de telles formules est un *système* (d'équations et d'inéquations).

L'affectation sylvestre X est appelée solution du système :

$S = \{p_1 := q_1, p_2 := q_2, \ldots\} \cup \{s_1 \neq t_1, s_2 \neq t_2, \ldots\}$

si *X* contient les mêmes variables que *S* et si *X* est telle que les arbres p_i/X sont respectivement égaux aux arbres q_i/X et que les arbres s_i/X sont respectivement différents des arbres t_i/X.
En utilisant ces notions il est possible de représenter le premier arbre de ce paragraphe par :

("a"."b"."c".nil) / {} ou ["a","b","c"] / {} suivant la syntaxe.

Le second arbre par :

plus(fois(l2, l1), fois(l2, l1)) / {} ou
plus(x, x) / X, avec X solution de {*x = fois(l2, l1)*}

et le troisième par :

x/X, avec *X* solution de { *x = ou("c", et("a", et(x, "b")))* }

1.6. Les opérateurs

Les opérateurs ont été introduits dans le compilateur Prolog II+, comme un moyen souple et clair d'exprimer certains arbres.
Par exemple, la représentation interne du terme *mul(sub(5, 3),add(5, 3))* étant la même que celle du terme *(5 - 3) * (5 + 3)*, la deuxième expression est certainement plus agréable à lire que la première.

La syntaxe des expressions écrites avec des opérateurs est donnée par:

$expr_n$ =	$prefix\_op_{n,d}$, $expr_d$;
$expr_n$ =	$expr_g$, $postfix\_op_{n,g}$;
$expr_n$ =	$expr_g$, $infix\_op_{n,g,d}$, $expr_d$;
$expr_n$ =	$expr_{n-1}$;
$expr_0$ =	pterm ;
$prefix\_op_{n,d}$ =	identifier | graphic_symbol ;
$postfix\_op_{n,g}$ =	identifier | graphic_symbol ;
$infix\_op_{n,g,d}$ =	identifier | graphic_symbol ;

Voir la description complète au paragraphe 1.9.

1.7. Les règles et les assertions

D'un point de vue théorique, un programme Prolog sert à définir un sous-ensemble A dans l'ensemble R de nos arbres. Les éléments de A sont appelés *assertions* et l'on peut généralement associer une phrase déclarative à chacun d'eux. La figure 1.9 montre quelques exemples de telles associations. L'ensemble A des assertions est généralement infini et constitue en quelque sorte une immense banque de données. Nous verrons plus loin que l'exécution d'un programme peut être vue comme une consultation d'une fraction de cette banque. Bien entendu cette banque ne peut être enregistrée sous une forme explicite. Elle doit être représentée à partir d'une information finie mais suffisante pour pouvoir déduire la totalité de l'information contenue dans la banque. Dans ce but, la définition de l'ensemble A des assertions est faite au moyen d'un ensemble fini de règles, chacune étant de la forme :

t_0 -> t_1 … t_n

où *n* peut être nul et où les t_i sont des termes.

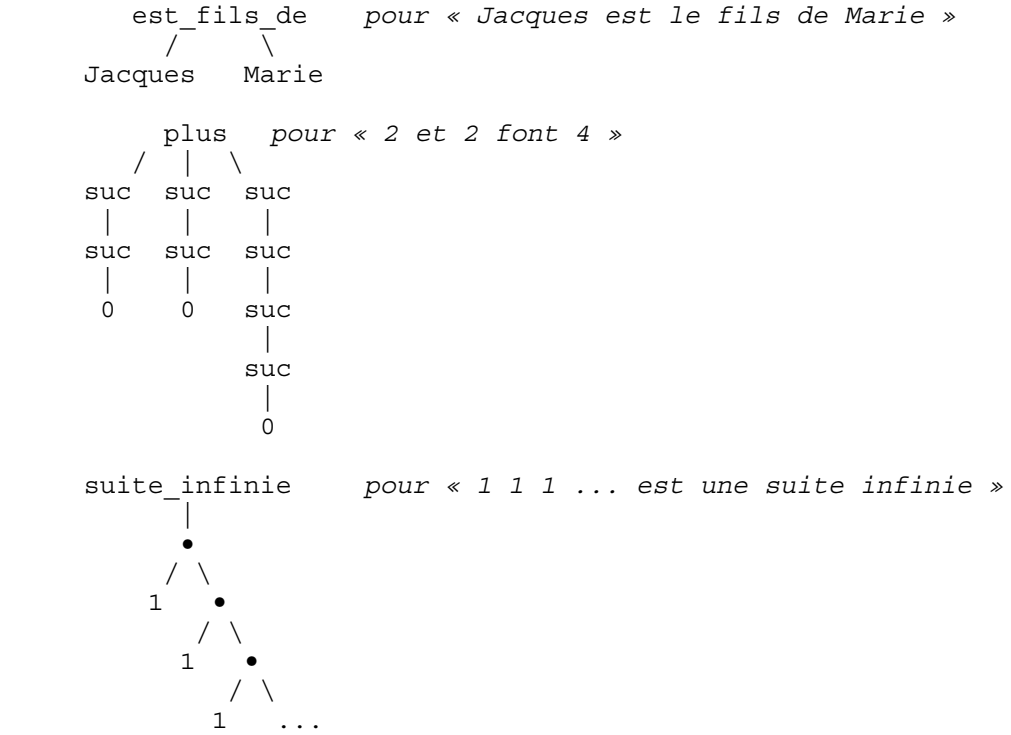


Figure 1.9

Une syntaxe simplifiée[4] des règles est la suivante :

4 On trouvera au dernier paragraphe de ce chapitre la syntaxe complète des règles.

3.1	$rule^E$ =	term , "." ;
3.2	$rule^E$ =	term , ":-" , term { "," , term }, "." ;
3.3	$rule^P$ =	term , "->", { term }, ";" ;

Avec la contrainte fondamentale suivante :

Le terme qui est le membre gauche d'une règle doit être:
- *soit un identificateur*
- *soit un tuple dont le premier argument est un identificateur*

Par exemple, les termes : *go*, *pere_de(_x,_y)* ou <*pere_de, _x, _y*> peuvent être des têtes de règle correctes, tandis que -contrairement à ce qui se passe pour Prolog II- des termes comme <<*pere_de, _x*>, *_y*> ou <*masc.sing, x, y*> ne peuvent pas l'être.

Pour le moment nous ferons abstraction de la notion de *parasite*, qui comme nous le verrons plus tard, est un moyen ad hoc d'appeler des sous-programmes non écrits en Prolog.

Les règles de la forme :

t_0 -> t_1 … t_n

induisent un ensemble, généralement infini, de règles particulières portant sur des arbres :

$t_0 / X \Rightarrow t_1 / X \ldots t_n / X$

obtenues en considérant, pour chaque règle, toutes les affectations sylvestres possibles :

$X = \{ x_1 := s_1, \ldots , x_m := s_m \}$

qui font intervenir les variables de la règle en question.

Chacune de ces règles particulières :

$r_0 \Rightarrow r_1 \ldots r_n$

peut s'interpréter de deux façons :

(1) Comme une règle de réécriture :
r_0 se réécrit dans la suite $r_1 \ldots r_n$,
et donc, lorsque n=0, comme :
r_0 s'efface.

(2) Comme une implication logique portant sur le sous-ensemble A d'arbres:
$r_1, r_2, \ldots, r_n$ éléments de A, entraîne r_0 élément de A.
Dans ce cas, lorsque $n = 0$, l'implication se résume à :
r_0 élément de A.

Suivant que l'on prend l'une ou l'autre des interprétations, les *assertions* se définissent par :

Définition 1 : les assertions sont les arbres que l'on peut *effacer*, en une ou en plusieurs étapes au moyen des règles de réécriture.

Définition 2 : les assertions forment le plus petit ensemble A d'arbres qui satisfait les implications logiques.

On peut démontrer l'existence du plus petit ensemble de la deuxième définition et l'équivalence des deux définitions.

1.8. Les mécanismes de base de Prolog

Nous venons de montrer quelle est l'information implicite contenue dans un programme Prolog, mais nous n'avons pas montré ce qu'est l'exécution d'un programme Prolog. Cette exécution vise à résoudre le problème suivant :

> Etant donné un programme qui est une définition récursive d'un ensemble A d'assertions.
>
> Etant donné une suite de termes $T_0 = t_1 \ldots t_n$ et l'ensemble de ses variables $\{ x_1, \ldots, x_m \}$.
>
> Trouver toutes les affectations sylvestres $X = \{ x_1 := r_1, \ldots, x_m = r_m \}$ qui sont telles que les arbres $t_1/X, \ldots, t_n/X$ soient des assertions.

Pour résoudre ce problème l'ordinateur doit produire toutes les dérivations de la forme :

$$(T_0, S_0) \text{ -> } (T_1, S_1) \text{ -> } (T_2, S_2) \text{ -> } \ldots$$

Les T_i étant des suites de termes appelés *buts* et les S_i étant des systèmes d'équations et d'inéquations ayant au moins une solution. S_0 est le système vide : { }. On peut expliquer simplement la dérivation (T_i, S_i) -> (T_{i+1}, S_{i+1}) au moyen des trois lignes suivantes :

(1) $(q_0\ q_1 \ldots q_n\ , S)$
(2) p_0 -> $p_1 \ldots p_m$
(3) $(\ p_1 \ldots p_m\ q_1 \ldots q_n\ , S \cup \{ q_0 = p_0 \})$

La première ligne représente la forme que doit avoir le couple (T_i , S_i), la seconde la règle utilisée et la troisième le résultat (T_{i+1}, S_{i+1}). Avant d'utiliser une règle, il est nécessaire de renommer ses variables pour qu'aucune d'entre elles n'apparaisse dans (T_i ,S_i). Il est aussi nécessaire de vérifier que le nouveau système S_{i+1} qui est obtenu en ajoutant l'équation $\{q_0 = p_0\}$ à S_i possède au moins une solution. Traditionnellement cette vérification est appelée *unification* de q_0 avec p_0.

Le but des dérivations précédentes est de trouver un couple (T_j, S_j) pour lequel la suite T_j est vide ; ce couple est donc dérivable de (T_0, { }). On peut montrer que les affectations sylvestres X qui sont solution de S_j sont les réponses à notre problème. Le résultat imprimé par l'ordinateur est alors une forme simplifiée du système S_j dans laquelle les inéquations sont omises.

On peut également montrer que Prolog II+ vérifie parfaitement si un système d'équations et d'inéquations a au moins une solution.

D'un point de vue plus pratique, quand on lance Prolog, on se trouve dans une boucle qui lit une suite de buts $T_0 = t_1 \ldots t_n$, cherche à les effacer de toutes les manières possibles ((T_0, { }) -> … -> (T_j, S_j) avec T_j vide), puis imprime les systèmes Sj correspondants. L'ordinateur écrit le caractère ">" quand il attend une suite de buts. De la même manière que nous avions une contrainte sur les règles, nous avons la contrainte suivante sur les buts :

> *A chaque étape, les arbres représentant le but qui va être effacé doivent avoir leur branche de gauche représentée par un identificateur.*

Les *parasites* permettent l'utilisation de sous-programmes externes. Pour expliquer comment ces sous-programmes sont appelés, il faut se référer à la première des trois lignes qui décrit le mécanisme de base de Prolog : si q_0 est un parasite, alors au lieu d'essayer d'utiliser une règle, Prolog exécute le sous-programme correspondant. Certains parasites apparaissent dans les règles prédéfinies qui font l'interface entre des sous-programmes externes et des règles Prolog. Cet ensemble de règles prédéfinies constitue un environnement de programmation très complet permettant notamment :

- de contrôler et de modifier le déroulement d'un programme (chapitre 2 : “Le contrôle de l'effacement des buts”) ;
- de structurer et modifier l'ensemble de règles qui constituent le programme Prolog courant (chapitre 3 : “Structuration et modification des règles”) ;
- d'avoir accès aux fonctions classiques d'arithmétique et de traitement de chaînes (chapitre 4 : “Opérations prédéfinies sur les données”) ;
- de gérer les entrées-sorties (chapitre 5 : “Les entrées / sorties”);
- de communiquer avec le système hôte (chapitre 6 : “L'environnement”).

Il est également possible au programmeur averti d'introduire de nouvelles règles prédéfinies qui font référence à de nouveaux parasites qui seront écrits dans un autre langage que Prolog (C, FORTRAN, Pascal…). (Voir chapitre 7 : "Extensions avec des langages externes" ou Annexe D : "Ajout de règles externes (méthode des parasites)").

Deux règles prédéfinies ont un lien étroit avec le mécanisme de base de Prolog II+: *dif* et *eq*.

La règle *dif* est la plus importante. C'est elle qui permet d'introduire les inéquations dans les systèmes d'équations et d'inéquations. Plus précisément, l'exécution de *dif(x,y)* ajoute l'inéquation $x \neq y$ au système S_i et vérifie que le nouveau système S_{i+1} a au moins une solution.

De la même manière la règle *eq* introduit les équations.

```
eq(x, x) -> ;
```

Les règles suivantes ne sont pas exactement des règles prédéfinies, mais tout se comporte comme si elles existaient. Elles sont utilisées pour transformer une liste de buts en un seul but.

```
p.q ->  p q ;
nil -> ;
```

1.9. La syntaxe complète de Prolog II+

Nous rassemblons ici la syntaxe complète de Prolog II+ ainsi qu'un certain nombre d'exemples et de remarques additionnelles. Les notations utilisées sont celles de la commission de normalisation de Prolog (voir le premier paragraphe de ce chapitre).

1.9.1. Le niveau syntaxique

1	program =	{ rule | directive } ;
2.1[0]	directive[P] =[7]	"->", { pterm} , ";" ;
2.2[1]	directive[E] =[7]	":-", $expr_{1199}$, "." ;
3.1[0]	rule[P] =	term , "->", { pterm }, ";" ;
3.2[1]	rule[E] =	term, "." ;
4.1	term[P] =	$expr_{1000}$, ["." , term] ;
4.2	term[E] =	$expr_{1200}$;
5	termlist =	$expr_{999}$, { "," , $expr_{999}$ } ;
6.1[2]	$expr_n$ =	$prefix\_op_{n,d}$, $expr_d$;
6.2	$expr_n$ =	$expr_g$, $postfix\_op_{n,g}$;
6.3	$expr_n$ =	$expr_g$, $infix\_op_{n,g,d}$, $expr_d$;
6.4[2]	$expr_n$ =	$expr_{n-1}$;
6.5	$expr_0$ =	pterm ;
7.1	pterm =	(identifier | variable) , ["(" , termlist , ")"];
7.2[3]	pterm =	(identifier | variable) , "[" , term , "]";
7.3	pterm[P] =	"<", termlist , ">" ;
7.4[4]	pterm =	"< >", ["(", termlist ,")"];
7.5[5]	pterm =	"{", termlist , "}" ;
7.6[6]	pterm =	"[" , listexpr , "]" | "[ ]" ;
7.7	pterm =	constant | "!" | "/?", integer_number ;
7.8	pterm =	"(" , term , ")" ;
8.1	listexpr =	$expr_{999}$, ["," , listexpr] ;
8.2	listexpr =	$expr_{999}$, "|" , $expr_{999}$;

Quand Prolog est lancé, la machine est prête à exécuter un programme, elle attend un *term*. Quand Prolog est passé dans un mode de compilation, c'est à dire un mode d'insertion de programme (cf. chapitre 3 de ce manuel), Prolog attend alors un *program*.

Notes:

0. Définit la syntaxe en mode Prolog II.

1. Définit la syntaxe en mode Edinburgh.

2. $expr_n$ représente la suite de règles $expr_1$,...,$expr_{1000}$ en syntaxe Prolog II, et $expr_1$,...,$expr_{1200}$ en syntaxe Edinburgh.

3. La règle 7.2 exprime une syntaxe alternative pour les références aux composantes des tableaux, qui permet un accès optimisé. Par exemple, si *table* est le nom d'un tableau (défini par l'emploi de la règle prédéfinie *def_array*) alors les deux expressions *table(10)* et *table[10]* sont équivalentes. Cependant, le compilateur traduit la deuxième forme de manière plus efficace.

4. Les règles 7.3 et 7.4 définissent deux syntaxes alternatives pour les tuples. En syntaxe Edinburgh seule la deuxième alternative est possible. On a donc les équivalences:

   < x , y > ⇔ < > (x , y)

5. La règle 7.5 permet de décrire des grammaires non contextuelles (en vue d'écrire en Prolog des analyseurs syntaxiques).

6. En syntaxe Prolog II, les règles 4 d'une part, et 7.6, 8.1 et 8.2 d'autre part définissent deux syntaxes alternatives pour la notation des listes. On a les équivalences suivantes :

[a | b]	⇔	a . b
[a , b]	⇔	a . b . nil
[a , b , c , d]	⇔	a . b . c . d . nil
[a , b , c | d]	⇔	a . b . c . d
[ ]	⇔	nil

   En syntaxe Prolog II, les deux syntaxes de liste peuvent toujours être mélangées dans les programmes :

   [a , b | c.d.nil] ⇔ a . [b , c , d]

7. Certains termes admis au titre de directives, ne sont pas des règles prédéfinies mais simplement des déclarations (par exemple *module*, *end_module*). Ils ne doivent pas dans ce cas être précédés de "->" ou ":-" comme il est dit dans les règles 2.1 et 2.2.

1.9.2. Les opérateurs

Les opérateurs permettent d'étendre dynamiquement la syntaxe des termes. On distingue les opérateurs préfixés, postfixés, et infixés.

10	$prefix\_op_{n,d}$ =	identifier | graphic_symbol ;
11	$postfix\_op_{n,g}$ =	identifier | graphic_symbol ;
12	$infix\_op_{n,g,d}$ =	identifier | graphic_symbol ;

On notera que dans la syntaxe Prolog II, les opérateurs ne sont pas autorisés au premier niveau des termes de la queue de règle, il faut parenthéser l'expression dans ce cas. En syntaxe Edinburgh, il n'y a pas de restriction.

Le type d'opérateur est indiqué par une convention permettant de définir la précédence des opérandes en fonction de la précédence *n* de l'opérateur:

type	précédence	type opérateur	préc.opérande(s)	exemple
fx	n	$op_{n,d}$	d:= n-1	- (- 1)
fy	n	$op_{n,d}$	d:= n	
xf	n	$op_{n,g}$	g:= n-1	
yf	n	$op_{n,g}$	g:= n	
xfx	n	$op_{n,g,d}$	g:= n-1, d:= n-1	val(1'<'2,x)
xfy	n	$op_{n,g,d}$	g:= n-1, d:= n	a,b,c[E]
yfx	n	$op_{n,g,d}$	g:= n, d:= n-1	val(1/2/3,x)

Le tableau suivant décrit les opérateurs en syntaxe prolog II+. Le tableau indique pour chaque opérateur le terme syntaxique construit.

opérateur	précédence	type	terme construit
'<'	700	xfx	sys:inf(t1,t2)
'=<'	700	xfx	sys:infe(t1,t2)
'>'	700	xfx	sys:sup(t1,t2)
'>='	700	xfx	sys:supe(t1,t2)
=\=	700	xfx	sys:'=\='(t1,t2)
=:=	700	xfx	sys:eql(t1,t2)
+	500	yfx	sys:add(t1,t2)
-	500	yfx	sys:sub(t1,t2)
'/\'	500	yfx	sys:'/\'(t1,t2)
'\/'	500	yfx	sys:'\/'(t1,t2)
*	400	yfx	sys:mul(t1,t2)
/	400	yfx	sys:div(t1,t2)
mod	400	yfx	sys:mod(t1,t2)
rem	400	yfx	sys:rem(t1,t2)
'<<'	400	yfx	sys:'<<'(t1,t2)
'>>'	400	yfx	sys:'>>'(t1,t2)
^	200	xfy	sys:'^'(t1,t2)
**	200	xfx	sys:'**'(t1,t2)
+[1]	200	fx	sys:add(t1)
-[1]	200	fx	sys:sub(t1)

Note 1 : Les arbres correspondant aux opérateurs unaires + et - sont évalués au moment de l'analyse si leur argument est une constante entière.

Note 2: Les opérateurs peuvent être écrits avec des quotes simples. Celles-ci n'ont donc pas d'autre fonction, en Prolog II+, que d'étendre la syntaxe des identificateurs. Il n'est donc pas possible d'utiliser une notation fonctionnelle autre que celles des tuples pour un foncteur déclaré en opérateur. Lorsque l'on a un doute sur le terme construit, on peut toujours le tester en décomposant le terme:

```
> eq(F(X,Y),1'<'2);
{F=inf, X=1, Y=2}
```

1.9.3. Le niveau lexical

Cette syntaxe définit les mêmes unités que celles reconnues par la primitive `read_unit(x,y)`.

Notes :

1. La règle L4 définit la syntaxe de base des variables; deux extensions de cette syntaxe sont données par les règles L5.P (syntaxe Prolog II) et L5.E (syntaxe Edinburgh). Dans les deux cas, le principe de l'extension est le même : un certain sous-ensemble des noms qui auraient été des représentations abrégées d'identificateurs pour la syntaxe de base, est ajouté à l'ensemble des variables.

Ces deux extensions sont facultatives et *incompatibles entre elles*. C'est l'utilisateur qui, au démarrage de la session Prolog, choisit la syntaxe qu'il souhaite employer: se référer au Manuel d'Utilisation.

2. Certaines combinaisons sont interdites comme "/*", "*/", "|*", "*|".

3. Une option sur la ligne de commande permet de rendre l'exposant facultatif, au prix d'une ambiguïté avec les listes en syntaxe Prolog II: 1.2 est lu comme un réel en entrée. Cette règle n'est valide que si cette option est choisie. Voir le manuel d'utilisation, paragraphe 2.3.

4. La règles L2.1 et L2.2 donnent la syntaxe de base des identificateurs. Une extension pour la syntaxe Edinburgh est donnée par les règles L2.3 et L2.4.

5. La règle L2.3 est nécessaire pour exprimer le passage de l'identificateur abrégé représenté par un graphic_symbol à sa représentation complète. En effet le caractère qui délimite le préfixe de l'identificateur abrégé, étant lui-même un caractère graphique, une ambiguïté apparaîtrait si les quotes n'étaient pas ajoutées. Par exemple, *sys::-* est vu par l'analyseur d'unités lexicales comme: l'identificateur *sys*, immédiatement suivi de l'identificateur *::-* et *sys:':-'* est vu comme: l'identificateur prédéfini *:-* de la famille *sys*.

L1	unit =	identifier | separator | variable | constant | graphic_symbol ;
L2.1	identifier =	prefix , prefix_limit , abbreviated_id ;
L2.2	identifier =	abbreviated_id ;
L2.3[E]	identifier[5] =	prefix , prefix_limit , "'", graphic_symbol, "'" ;
L2.4[E]	identifier[4] =	graphic_symbol ;
L3.1	abbreviated_id =	name - extended_var ;
L3.2	abbreviated_id =	"'", { (character - "'") | "' '" }, "'" ;
L4	variable[1] =	"_" , {alpha} | extended_var ;
L5.[P]	extended_var =	letter, [(digit | "_"), {alpha}], { "'" } ;
L5.[E]	extended_var =	big_letter, [{alpha}] ;
L6	prefix =	[name , { prefix_limit , name }] ;
L7	constant =	integer_number | real_number | string ;
L8.1	integer_number =	digits ;
L8.2	integer_number =	"0b", binary_number ;
L8.3	integer_number =	"0o", octal_number ;
L8.4	integer_number =	"0x", hex_number ;
L8.5	integer_number =	"0'", character ;
L9	real_number =	digits, ".", digits, ("E"|"e"|"D"|"d"), [["+"|"-"],digits];
L9.1[S]	real_number[3] =	digits, ".", digits, [("E"|"e"|"D"|"d"), [["+"|"-"],digits]];
L10	string =	"'" , { string_char } , "'" ;
L11	name =	letter , { alpha } ;
L12	digits =	digit , { digit } ;
L13.[P]	graphic_symbol[2] =	"->" | graphic_c, { graphic_c | "." };
L13.[E]	graphic_symbol[2] =	{ graphic_c };
L14.1	comment =	"|*", { character } , "*|" ;
L14.2	comment =	"/*", { character } , "*/" ;
L14.3	comment =	"%", { character } , newline ;

1.9.4. Les caractères

Les règles dont le membre gauche est annoté par I ne sont actives que lorsque le mode d'exécution de Prolog II+ est le mode ISO(cf. § U2.3). Les règles dont le membre gauche est annoté par H ne sont actives que lorsque le mode d'exécution de Prolog II+ est le mode hôte(cf. § U2.3).

Est désigné par *host_letter*, tout caractère du système hôte communément admis comme lettre et qui n'appartient pas au jeu ISO 8859-1; de la même façon, est désigné par *host_graphic_char*, tout caractère imprimable du système hôte n'étant ni un *alpha*, ni un *separator*, ni un *special_char* et inconnu du jeu ISO 8859-1.

Une description adaptée au jeu de caractères de la machine hôte sera donnée dans le manuel d'utilisation au paragraphe 3.2.

La règle C6.2 et les règles C7.2 et C8.1 représentent le même ensemble de caractères, mais elles ne sont pas valides simultanément. Celle annotée par P est valide en syntaxe Marseille, celles annotées par E sont valides en syntaxe Edinburgh. En changeant de syntaxe, ces caractères ne jouent plus le même rôle.

La règle C5 est valide, si le caractère qui délimite dans les identificateurs complets, le préfixe et le suffixe, n'a pas été redéfini. Il peut valoir alors, un des caractères graphiques.

C1	big_letter =	"A" | … | "Z" ;
C2	letter =	big_letter | "a" | … | "z" | "À" … "ß" - "×" | "à" … "ÿ" - "÷";
C2.1	letter^{H}=	host_letter;
C2.2[1]	letter$^{I,\ i1}$ =	"\", accent_escape ;
C3.1	binary_digit =	"0" | "1" ;
C3.2	octal_digit =	"0" | … | "7" ;
C3.3	digit =	"0" | … | "9" ;
C3.4	hex_digit =	digit | "a" | "b" | "c" | "d" | "e" | "f" | "A" | "B" | "C" | "D" | "E" | "F";
C4	alpha =	letter | digit | "_";
C5	prefix_limit =	":" ;
C6.1	separator =	"(" | ")" | "[" | "]" | "{" | "}" | "|" | "," ;
C6.2	separator^{P}=	";" | "." | "<" | ">" ;
C7.1	special_char=	"%" | "'" | "" | "_" | "!" | "`" ;
C7.2	special_char^{E}=	";" ;
C8	graphic_c =	graphic_char ;

C8.1	graphic_char^{E} =	"." | "<" | ">" ;
C8.2	graphic_char =	"#" | "$" | "&" | "*" | "+" | "-" | "/" | ":" | "=" | "?" | "\" | "@" | "^" | "~" | NBSP … "¿" | "×" | "÷" ;
C8.3	graphic_char^{H} =	host_graphic_char ;
C9	character =	letter | digit | separator | graphic_char | special_char ;
C10	string_char =	character - ("'" | "\") | """;
C10.1	string_char^{i0} =	"\";
C10.2	string_char^{i1} =	"\", format_escape ;
C10.3[1]	string_char$^{I, i1}$ =	"\", accent_escape ;

C11[1]	accent_escape =	accent , accent_letter | "~a" | "~A" | "~n" | "~N" | "~o" | "~O" | "cc"| "CC" | "ae" | "AE" | "BB" | "/o" | "/O" | "'y"| "'y"| "'Y" | "-d"|"-D" | "pp" | "PP" | "oa"| "oA" ;
C12[3]	accent =	"`" | "'" | "^" | ":" ;
C13	accent_letter =	"a" | "e" | "i" | "o" | "u" | "A" | "E" | "I" | "O" | "U" ;
C14[2]	format_escape =	"b" | "f" | "n" | "r" | "t" | "\" | newline | octal_digit, octal_digit, octal_digit | ("x" |"X"), hex_digit, hex_digit ;

1.9.5. Les caractères accentués

Une option de comportement (cf. §2.3. du manuel d'utilisation) définit le mode de lecture du caractère "\". Lorsque l'interprétation du "\" est active, les règles sont annotées par i1, lorsqu'elle ne l'est pas les règles sont annotées par i0. Les règles ainsi annotées sont exclusives. Leur validité dépend de l'option choisie.

Les notes ci-après sont valides uniquement lorsque l'interprétation du "\" est active.

Notes :

1. Il existe un mode d'exécution Prolog (cf. § U2.3.) dans lequel les *accent_escape* ne sont pas permis et sont remplacés par les *format_escape*; dans ce mode, ces règles(C2.2, C10.3, C11) ne sont pas valides. Sinon les *accent_escape* peuvent toujours être utilisés en entrée pour spécifier des caractères accentués. En sortie

Prolog utilise un *accent_escape* pour les caractères accentués du jeu ISO 8859-1 n'existant pas dans le jeu de caractère du système hôte.

2. De la même manière que pour les *accent_escape*, les *format_escape* peuvent toujours être utilisés en entrée pour spécifier un caractère. En sortie Prolog utilise un *format_escape* pour les caractères n'existant pas dans le jeu de caractère du système hôte, et ne pouvant être représentés par un *accent_escape* (en mode hôte par exemple).

3. Le caractère ":" dans un *accent_escape* représente le diacritique tréma:
cano\:e <=> canoë

	128	144	160	176	192	208	224	240
0					À	-,D	à	∂
1					Á	Ñ	á	ñ
2					Â	Ò	â	ò
3					Ã	Ó	ã	ó
4					Ä	Ô	ä	ô
5					Å	Õ	å	õ
6					Æ	Ö	æ	ö
7					Ç	×	ç	÷
8					È	Ø	è	ø
9					É	Ù	é	ù
10					Ê	Ú	ê	ú
11					Ë	Û	ë	û
12					ì	Ü	ì	ü
13					í	I, o	í	I, o
14					î	´,Y,	î	´,y
15					ï	ß	ï	ÿ

Table 1 : les caractères accentués
dans le code ISO 8859-1

La table suivante donne la correspondance entre les caractères accentués et leur expression sous la forme de *accent_escape* . On y apprend, par exemple, que les deux chaînes de caractères :

"tel maître, tel élève" et "tel ma\^itre, tel \'el\`eve"

sont équivalentes.

	128	144	160	176	192	208	224	240
0					`A	-D	`a	-d
1					'A	~N	'a	~n
2					^A	`O	^a	`o
3					~A	'O	~a	'o
4					:A	^O	:a	^o
5					oA	~O	oa	~o
6					AE	:O	ae	:o
7					CC		cc	
8					`E	/O	`e	/o
9					'E	`U	'e	`u
10					^E	'U	^e	'u
11					:E	^U	:e	^u
12					`I	:U	`i	:u
13					'I	'Y	'i	'y
14					^I	PP	^i	pp
15					:I	BB	:i	:y

Table 2 : «accent_escape» utilisés par Prolog II+

1.10. Le macroprocesseur

Il est possible d'écrire en Prolog II+ quelques macros (ou alias) très simples afin d'augmenter la lisibilité des programmes. Cette fonctionnalité est réalisée au moyen d'une directive de compilation (et non un prédicat):

set_alias(i,t)

Définition d'un alias.

A partir du lieu et moment où cette directive est rencontrée, le lecteur remplace certaines (détails ci-dessous) occurrences de l'identificateur *i* (appelé *alias*) par le terme *t* (appelé *valeur*). Cette définition est désactivée en fin de compilation de plus haut niveau d'imbrication.

La valeur *t* doit être de type entier, réel ou chaîne de caractères.

L'alias *i* doit être de type identificateur.

Seuls les identificateurs en position d'argument (pas les têtes de règle) et non préfixés explicitement dans le texte sont remplacés par la valeur de l'alias. Il est ainsi possible de conserver (en le préfixant explicitement dans le texte) un identificateur de même nom abrégé qu'un alias. Il est donc aussi logique que le préfixe de l'alias soit ignoré dans la directive. Néanmoins, dans le cas où celui-ci est déjà défini, la lecture d'une directive avec un alias explicitement préfixé dans le texte évitera sa substitution, et permettra donc une redéfinition (accompagnée d'un message d'avertissement). Le prédicat *val/1* permet la définition d'un alias au moyen d'un autre alias.

Exemples commentés:

```
> insert;
set_alias(foo,44);
set_alias(foo,55);              Ici redéfinition de 44 en 55
-> set_alias(44,55) :    ARGUMENT DE MAUVAIS TYPE

> insert;
set_alias(macro1,22);
set_alias(aa:macro1,44);        Ici le préfixage est un moyen de redéfinition
WARNING: macro1 DEJA DEFINIE, NOUVELLE VALEUR PRISE EN COMPTE
rg1(macro1)->;
rg1(aa:macro1)->;
-> insert;                      Ici, 2ème niveau d'imbrication des compilations
set_alias(macro2,val(2 * macro1)); définition à partir d'une autre macro
rg2(macro2)->;
rg2(macro1)->;
;                               Ici, on revient au 1er niveau de compilation
macro1(macro2)->;               Ici protection automatique des têtes de règle
macro1(macro1)->;
;
{}
> rg1(I);
{I=44}                          Ici la nouvelle valeur est bien prise en compte
{I=aa:macro1}                   Ici pas de substitution de l'identificateur préfixé
> rg2(I);
{I=88}
{I=44}                          La macro définie au 1er niveau de compilation a
                                été prise en compte dans le second niveau
> macro1(I);
{I=88}                          La macro définie au 2ème niveau de
                                compilation a été prise en compte dans le
                                premier niveau
{I=44}
> insert;                       Ici on démarre une nouvelle compilation
rg(macro1)->;
rg(macro2)->;
;
{}
> rg(I);
{I=macro1}                      Les définitions ont bien été annulées
{I=macro2}
```

NB: Une macro apparaissant comme opérande gauche d'un opérateur infixé doit être parenthésée. Exemple:

```
?- insert.
set_alias(foo,44).
oper(X) :- foo >= X.
oper(X) :- (foo) >= X.
```

```
.
list.
oper(_345) :-
   val(foo,_348),
   _348 >= _345 .
oper(_345) :-
   44 >= _345 .
```

2. Le contrôle de l'effacement des buts

2.1. Le contrôle
2.2. Geler
2.3. A propos des arbres infinis
2.4. Quelques conseils pour la programmation récursive
2.5. Les méta-structures

2.1. Le contrôle

A chaque étape la machine Prolog doit faire deux choix :

(1) L'un pour choisir un but dans une suite de buts à effacer. C'est le premier élément de la suite qui est toujours choisi. C'est à dire que pour effacer une suite de buts $q_0\ q_1 \ldots q_n$, on efface d'abord q_0 et ensuite $q_1 \ldots q_n$.
(2) L'autre pour choisir la règle qui sert à effacer un but *b*. C'est la première règle dont la tête s'unifie avec le but *b* qui est choisie.

Cela peut se résumer par le programme Prolog suivant :

```
effacer(nil) -> ;
effacer(t.l) -> rule(t, q) effacer(q) effacer(l) ;
```

rule(t,q) est une règle prédéfinie (voir le chapitre 3) qui donne par énumération toutes les règles dont la tête s'unifie avec *t* et la queue avec *q*.

Le moyen de contrôle consiste à modifier ou à restreindre les deux choix précédents. La manière dont Prolog fait ces choix peut amener certains programmes à boucler et par conséquent à ne pas se comporter comme on pouvait l'espérer. Les deux exemples suivants illustrent ce phénomène.

Exemple 1 : Un cas typique est celui de la transitivité. Quand on cherche à effacer *plus_grand(Jo,x)* en utilisant le programme suivant, on retrouve une instance de ce même but à effacer. Le programme se met alors à boucler et se termine par un débordement[1] de pile (ou par une interruption utilisateur!). La manière correcte d'écrire ce programme consiste à enlever la récursivité à gauche.

```
plus_grand(Jo, Max) -> ;
plus_grand(Max, Fred) -> ;
plus_grand(x, y) -> plus_grand(x, z) plus_grand(z, y);
```

[1] Si le système de réallocation automatique n'est pas désactivé au démarrage de Prolog, un certain nombre de réallocations se produiront avant le débordement.

```
> plus_grand(Jo, x);
{x=Max}
{x=Fred}

DEBORDEMENT

"Une bonne solution"

plus_grand'(Jo, Max) -> ;
plus_grand'(Max, Fred) -> ;

plus_grand(x, z) -> plus_grand'(x, y) plus_grand_ou_egal(y,
z);

plus_grand_ou_egal(x, x)  ->;
plus_grand_ou_egal(x, y)  -> plus_grand(x, y);
```

Exemple 2 : Cet exemple énumère toutes les listes construites avec 1. Avec le (mauvais) programme ci-dessous, c'est d'abord la liste infinie qui devrait être produite; bien entendu, la bonne solution s'obtient en permutant l'ordre des deux règles.

```
liste_de_un(1.x) -> liste_de_un(x) ;
liste_de_un(nil) -> ;

>liste_de_un(x);

DEBORDEMENT[1]
```

La coupure "!"

Normalement Prolog essaye d'effacer une suite de buts de toutes les manières possibles. Mais si on utilise une règle contenant un «!» (ou *coupure*) pour effacer un but *q*, l'effacement de ce «!» supprimera tous les choix de règles restant à faire pour effacer ce but *q*. Cela restreint la taille de l'espace de recherche : on peut dire que «!» fait «oublier» les autres manières possibles d'effacer *q*.

Le «!» ne peut apparaître que parmi les termes qui constituent le membre droit d'une règle. Les choix qui restent à examiner et que l'effacement du «!» fait «oublier» sont :

- les autres règles ayant la même tête que celle où le «!» figure
- les autres règles qui auraient pu être utilisées pour effacer les termes compris entre le début de la queue et le «!»

Cette question est illustrée par les exemples suivants :

```
couleur(rouge) ->;
couleur(bleu) ->;

taille(grand) ->;
taille(petit) ->;

choix1(x.y) -> couleur(x) taille(y);
```

[1]Si le système de réallocation automatique n'est pas désactivé au démarrage de Prolog, un certain nombre de réallocations se produiront avant le débordement.

```
choix1("c'est tout") ->;

choix2(x.y) -> ! couleur(x) taille(y);
choix2("c'est tout") ->;

choix3(x.y) -> couleur(x) ! taille(y);
choix3("c'est tout") ->;

choix4(x.y) -> couleur(x) taille(y) !;
choix4("c'est tout") ->;

>choix1(u);
{u=rouge.grand}
{u=rouge.petit}
{u=bleu.grand}
{u=bleu.petit}
{u="c'est tout"}
>choix2(u);
{u=rouge.grand}
{u=rouge.petit}
{u=bleu.grand}
{u=bleu.petit}
>choix3(u);
{u=rouge.grand}
{u=rouge.petit}
>choix4(u);
{u=rouge.grand}
>choix1(u) !;
{u=rouge.grand}
```

On peut considérer le «!» comme une annotation que l'on fait à un programme pour le rendre plus efficace; bien entendu, cela ne se justifie que si on ne s'intéresse qu'à la *première solution* fournie par ce programme.

Des utilisations classiques du «!» sont :

```
" Première solution uniquement "
   premiere_solution_uniquement(b)  -> b !;

" Si Alors Sinon "
   si_alors_sinon(p,a,b) -> p ! a;
   si_alors_sinon(p,a,b) -> b;

" non "
   non(p)  -> p ! fail;
   non(p)  -> ;
```

Dans le cas de *non* montré ci-dessus, il faut remarquer que l'on peut avoir des résultats inattendus si *p* contient des variables libres. C'est ce que montre le petit exemple suivant :

```
homme(Abélard) -> ;

femme(x) -> non(homme(x));

>femme(Eloïse);
{}
>femme(Abélard);
>femme(x) eq(x, Eloïse);
>
```

^(X,Y)

X doit être une variable et Y un terme quelconque.
^(X, Y) signifie: il existe *X* tel que *Y* soit vrai, et est équivalent à un appel de *Y*.
L'utilisation de ce prédicat (qui est aussi un opérateur) n'a de sens que dans les prédicats *bagof/3* et *setof/3* , pour indiquer les variables existentielles et les retirer de l'ensemble des variables libres.

bagof(x, p, l)

Pour chaque instantiation différente de l'ensemble des variables libres du but *p,* non existentielles et n'apparaissant pas dans le terme *x*, unifie *l* avec la liste de toutes les solutions *x* lorsqu'on efface *p*. Chaque liste *l* est construite suivant l'ordre des solutions trouvées. Exemple:

```
aa(2,1) -> ;
aa(1,2) -> ;
aa(1,1) -> ;
aa(2,2) -> ;
aa(2,1) -> ;
>bagof(X, aa(X,Y),L);
{Y=1, L= 2.1.2.nil}
{Y=2, L= 1.2.nil}
> bagof(X,Y^aa(X,Y),L);
{L=2.1.1.2.2.nil}
>
```

block(e, b), block_exit(e)

block est une règle prédéfinie qui permet de terminer brutalement l'effacement d'un but *b*. Cette primitive est faite essentiellement pour la récupération des erreurs. On peut considérer que :

- Pour effacer *block(e, b)* on efface *b* en ayant auparavant créé une paire de parenthèses fictives, étiquetées par *e*, autour du but *b*.

- *block_exit(e)* provoque l'abandon immédiat de l'effacement de tous les buts inclus entre les parenthèses étiquetées par *e* et supprime tous les choix éventuels en attente pour ces buts. L'effacement continue ensuite normalement, après les parenthèses étiquetées par *e*.

De plus :

- Une étiquette est un terme Prolog quelconque, et on considère que deux parenthèses étiquetées sont identiques si leurs étiquettes respectives sont unifiables.

- S'il y a plusieurs parenthèses étiquetées par *e*, *block_exit(e)* s'arrête au couple de parenthèses le plus interne.

- Si *block_exit(e)* ne rencontre pas de parenthèses *e*, alors on revient au niveau de commande avec le message d'erreur correspondant à *e*, si *e* est un entier, ou le message *'block_exit' SANS 'block' CORRESPONDANT*, si *e* n'est pas un entier.

C'est ce même mécanisme qui est utilisé pour la gestion des erreurs dans Prolog. Une erreur rencontrée dans l'exécution de Prolog provoque l'effacement du but *block_exit(i)* où *i* est un entier correspondant au numéro de l'erreur. De cette manière l'utilisateur peut récupérer toutes les erreurs de Prolog, pour pouvoir les traiter dans son application.

Lorsque les optimisations de compilation sont actives (option -f o1 au lancement de Prolog), la compilation de block est optimisée et la décompilation d'un tel but ne sera pas identique mais équivalente au but d'origine. Le but *block(e,b)* apparaissant dans une queue de règles, sera décompilé en *block(e',_,b')*.

La règle prédéfinie *quit* génère un *block_exit(14)*. Elle est définie ainsi :

```
quit -> block_exit(14);
quit(i) -> block_exit(14,i);
```

A titre d'illustration voici un programme qui lit une suite de commandes :

```
executer_commande ->

   block(fin_commande, toujours(lire_et_exec));

lire_et_exec -> outm("?") in_char'(k) exec(k);

exec("q")  -> block_exit(fin_commande) ;
exec("f")  -> block_exit(16) ;   /* Interruption */
exec("n")  -> ;

toujours(p)  -> repeter p fail ;

repeter -> ;
repeter -> repeter ;
```

Dans cet exemple, la commande "q" utilise *block_exit* pour retourner au niveau supérieur de *executer_commande*. La commande "f" simule une erreur Prolog et rend le contrôle au *block* qui traite les erreurs de Prolog (puisque 16 ne peut s'unifier avec *fin_commande*). S'il n'y a pas de *block* englobant, on revient au niveau supérieur de Prolog.

block(e, c, b), block_exit(e, c)

Fonctionnement analogue aux formes précédentes, avec deux «étiquettes» *e* et *c* à la place d'une seule. *b* est le but à effacer. *block(e,c,b)* lance l'effacement de *b*; l'effacement ultérieur de *block_exit* à deux arguments produira la recherche d'un *block* dont les deux premiers arguments s'unifient avec les deux arguments de *block_exit*.
Lorsque les optimisations de compilation sont actives (option -f o1 au lancement de Prolog), la compilation de block est optimisée et dans certains cas la décompilation d'un tel but ne sera pas identique mais équivalente au but d'origine. Le but *block(e,c,b)* apparaissant dans une queue de règles, sera décompilé en *block(e',c',b')*.
Ensemble, ces deux règles prédéfinies constituent une sorte de raffinement du mécanisme de récupération des erreurs; en pratique, *e* correspond au type d'erreur guetté; *c* correspond à un «complément d'information» rendu par le mécanisme de détection.

Un grand nombre de règles prédéfinies utilisent cette deuxième étiquette comme complément d'erreur. C'est souvent l'argument, cause de l'erreur, qui y est transmis. En phase de mise au point de programmes, cela peut paraître encore insuffisant. Dans le kit, des modules objets : *dbgbase.mo*, *dbggraph.mo* et *dbgedin.mo*, sont fournis. Ils permettent, après rechargement, en cas d'erreur, d'avoir des messages encore plus complets. Le deuxième argument de *block* sera unifié avec un terme de la forme :

prédicat d'appel ou < prédicat d'appel, complément d'erreur>

Lorsque *block_exit(e,c)* est effacé dans un environnement «parenthésé» par *block(e',b)*, alors *block_exit(e,c)* se comporte comme *block_exit(e)*. Inversement, lorsque *block_exit(e)* est effacé dans un environnement «parenthésé» par *block(e',c',b)* alors *block_exit(e)* se comporte comme *block_exit(e,nil)*

Erreurs et sous-sessions Prolog

Tout ce qui est dit ci-dessus concernant le mécanisme d'erreur *block / block_exit*, est vrai à l'intérieur d'une même session Prolog. Lorsqu'il s'agit de transmettre une erreur à travers une ou plusieurs sous-sessions (par exemple lorsqu'un but est lancé par menu), deux restrictions sont à mentionner :

- le deuxième argument de *block_exit* (souvent utilisé comme complément d'erreur) n'est pas transmis. Il vaudra toujours *nil*.

- le premier argument de *block_exit* est transmis uniquement s'il est de type entier. Dans le cas contraire c'est l'entier 317, pour l'erreur : 'block_exit' SANS 'block' CORRESPONDANT, qui est propagé.

Interruption

A tout instant un programme Prolog peut être interrompu au moyen d'une touche déterminée, dépendant du système utilisé (par exemple : <Ctrl-C>). Cette interruption est prise en compte par le mécanisme général des erreurs de Prolog et correspond à l'erreur 16. Cette erreur peut donc être récupérée par l'utilisateur. Sinon on revient au niveau de commande avec le message d'erreur suivant : INTERRUPTION UTILISATEUR.

bound(x)

bound(x) s'efface si *x* est lié. Une variable est considérée comme liée si elle a été unifiée contre :

- une constante (entier, réel, identificateur, chaîne),

- un terme de la forme $t_1.t_2$,

- un terme de la forme $<t_1, t_2, \ldots, t_n>$ ou *ff(t1, t2, ... , tn)*.

dif(t_1, t_2)

La règle prédéfinie *dif(t_1, t_2)* est utilisée pour contraindre t_1 et t_2 à représenter toujours des objets différents. Dès que cette contrainte ne peut plus être satisfaite, on revient en arrière (backtracking). L'implantation de *dif* utilise un mécanisme similaire à *geler*.

Voici deux primitives intéressantes que l'on peut écrire avec *dif* : *hors_de(x, l)* qui vérifie que *x* n'appartiendra jamais à la liste *l*; et *tous_differents(l)* qui vérifie que les composants de la liste *l* seront toujours différents deux à deux.

```
hors_de(x, nil) -> ;
hors_de(x, y.l) -> dif(x, y) hors_de(x, l);

tous_differents(nil) -> ;
tous_differents(x.l) -> hors_de(x, l)
 tous_differents(l);
```

Avec ces deux primitives on peut par exemple écrire un programme qui calcule des permutations : il suffit de dire que l'on a une liste d'entiers et que tous les éléments de cette liste sont différents deux à deux :

```
permutation(x1,x2,x3,x4)  ->
 tous_differents(x1.x2.x3.x4.nil)
 chiffre(x1) chiffre(x2) chiffre(x3) chiffre(x4);

chiffre(1) -> ;
chiffre(2) -> ;
chiffre(3) -> ;
chiffre(4) -> ;
```

Remarquons que dans cet exemple, Prolog met d'abord en place toutes les inégalités avant d'exécuter le programme non-déterministe classique. Ceci donne une programmation plus claire et plus efficace.

L'utilisation de *dif* permet également l' écriture de programmes qui auraient un comportement incorrect si l'on utilisait une définition utilisant la coupure. Par exemple la définition de la relation "*x* est élément de *l* à la valeur *v*" peut s'écrire :

```
élément(x,nil,false) ->;
élément(x,x.l,true) ->;
élément(x,y.l,v) -> dif(x,y) élément(x,l,v);

> élément(x,1.2.nil,v);
{x=1, v=true}
{x=2, v=true}
{x#1, x#2, v=false}
```

Si l'on définissait une telle relation en utilisant la primitive *!*, on introduirait de nombreux comportements anormaux :

```
élément(x,nil,false) ->;
élément(x,x.l,true) -> ! ;
élément(x,y.l,v) -> élément(x,l,v);

> élément(x,1.2.nil,v);
{x=1, v=true}              % une seule solution !!
> élément(2,4.2.3.nil,false);
{}                         % succès !!
```

default(t_1, t_2)

La règle prédéfinie *default* permet de réaliser le contrôle suivant: Si on peut effacer *t1*, alors on l'efface de toutes les manières possibles, sinon on efface *t2*. Il faut remarquer que contrairement à ce que l'on pourrait penser à première vue, cette primitive ne peut pas être réalisée avec '!'. Voyons sur un exemple une utilisation de cette règle:

```
répondre(p) -> default(p,outml("personne"));
homme(jean) ->;
homme(pierre) ->;

> répondre(homme(x));
{x=jean}
{x=pierre}
> répondre(femme(x));
personne
{}
```

eq(t_1, t_2)

Unification de t_1 et t_2 : correspond simplement à la règle :
eq(x,x) -> ; .

fail

Règle prédéfinie provoquant toujours un échec (*backtracking*).

findall(x, p, l)

Unifie *l* avec la liste de toutes les solutions *x* lorsqu'on efface *p*.

free(x)

S'efface uniquement si *x* n'est pas lié.

list_of(x, y, p, l)

Fournit la liste triée, sans répétition, de tous les individus qui satisfont une certaine propriété.
x est une variable.
y est une liste de variables.
p est un terme Prolog contenant au moins toutes ces variables.
Pour chaque ensemble de valeurs de *y*, unifie *l* avec la liste des valeurs de *x* pour que *p* soit vraie (c'est à dire pour que *p* s'efface).
L'ordre de la liste *l* est identique à celui défini sur les termes (cf. *term_cmp/3*)
Exemple : Prolog contenant la base de règles suivante :

```
homme(grand, michel, 184) ->;
homme(grand, alain, 183) ->;
homme(grand, henry, 192) ->;
homme(petit, nicolas, 175) ->;
homme(petit, julien, 176) ->;
homme(petit, gilles, 120) ->;

> list_of(x,t.nil,homme(t,x,h),l);
{t=grand, l=alain.henry.michel.nil}
{t=petit, l=gilles.julien.nicolas.nil}
```

not(X)

Est décrit par les règles:

```
not(X) :- X,!,fail.
not(X).
```

repeat

Est décrit par les règles :

```
repeat ->;
repeat -> repeat;
```

setof(x, p, l)

Pour chaque instantiation différente de l'ensemble des variables libres du but *p,* non existentielles et n'apparaissant pas dans le terme *x*, unifie *l* avec la liste de toutes les solutions *x* lorsqu'on efface *p*. Chaque liste *l* est triée et sans répétition. L'ordre de la liste *l* est identique à celui défini sur les termes (cf. *term_cmp/3*).

Exemple : Prolog contenant la base de règles suivante :

```
aa(2,1) -> ;
aa(1,2) -> ;
aa(1,1) -> ;
aa(2,2) -> ;
aa(2,1) -> ;
> setof(X, aa(X,Y),L);
{Y=1, L= 1.2.nil}
{Y=2, L= 1.2.nil}
> setof(X,Y^aa(X,Y),L);
{L=1.2.nil}
>
```

or(x,y)

Définit un ou logique transparent à la coupure. Exemple:

```
> op(900,xfy,or);
{}
> enum(i,4) (eq(i,1) or eq(i,2));
{i=1}
{i=2}
> enum(i,4) ([eq(i,3),'!'] or [outl(no),fail]);
no
no
{i=3}
>
```

2.2. Geler

Il s'agit de résoudre de manière efficace un problème qui se pose souvent en Prolog : retarder le plus possible certaines décisions. Ce qui revient à dire qu'il faut avoir suffisamment d'information avant de poursuivre l'exécution d'un programme, ou que certaines variables doivent avoir reçu une valeur pour pouvoir continuer. On profite alors des avantages de la programmation déclarative, tout en gardant une exécution efficace.

freeze(x, q)

Le but de cette règle prédéfinie est de retarder l'effacement de *q* tant que *x* est inconnu. Plus précisément :

(1) Si *x* est libre, alors *freeze(x, q)* s'efface et l'effacement de *q* est mis en attente (on dit que *q* est gelé). L'effacement effectif de *q* sera déclenché au moment où *x* deviendra liée.

(2) Si *x* est liée alors *freeze* lance l'effacement de *q* normalement.

C'est au programmeur de s'assurer qu'une variable *gelée* sera toujours dégelée dans le futur, pour que le but associé ne reste pas indéfiniment en attente. Ceci n'est pas vérifié par Prolog, et peut conduire à des solutions trop générales.

Remarque : Les impressions des variables d'une question sur lesquelles il reste des buts gelés se fait d'une manière spéciale :

```
> freeze(x,foo);
{x~foo.nil}
```

Voyons maintenant quelques exemples d'utilisation de *freeze*.

Exemple 1 : Evaluation de *somme(x, y, z)*. On veut résoudre l'équation $z = x + y$ seulement si les deux variables *x* et *y* sont connues.

```
somme(x, y, z) -> freeze(x, freeze(y, somme1(x, y, z))) ;

somme1(x, y, z) -> val(x+y, z);
```

Exemple 2 : Mais on peut faire mieux : On veut résoudre la même équation si deux au moins des variables *x*, *y*, *z* sont connues :

```
somme (x, y, z) ->
   freeze(x, freeze(y, ou_bien(u, somme1(x, y, z))))
   freeze(y, freeze(z, ou_bien(u, somme2(x, y, z))))
   freeze(x, freeze(z, ou_bien(u, somme3(x, y, z))));

ou_bien(u, p) -> free(u) ! eq(u, Cst) p;
ou_bien(u, p) ->;

somme1(x, y, z) -> val(x+y, z);
somme2(x, y, z) -> val(z-y, x);
somme3(x, y, z) -> val(z-x, y);
```

Ci-dessus, la variable *u*, qui devient liée dès que l'une des additions est effectuée, sert à empêcher que les autres additions, redondantes, soient calculées.

Exemple 3 : Cet exemple montre comment utiliser une variable pour contrôler un programme de l'extérieur. La règle *liste_de_un* qui a été donnée dans un exemple précédent, bouclait. Pour empêcher cette boucle on utilise simplement une condition externe : *longueur(l, n)* qui vérifie que la longueur d'une liste *l* est inférieure ou égale à *n*.

```
longueur(l,n) -> freeze(l,longueur'(l,n));

longueur'(nil, n) ->;
```

```
longueur'(e.l, n) ->
    dif(n,0) val(n-1, n') longueur(l, n');

liste_de_un(1.x) -> liste_de_un(x);
liste_de_un(nil) -> ;
> longueur(l,5) liste_de_un(l);
{l=1.1.1.1.1.nil}
{l=1.1.1.1.nil}
{l=1.1.1.nil}
{l=1.nil}
{l=nil}
```

Quand tous les buts ont été effacés, il se peut qu'il reste encore certains buts gelés sur des variables qui sont encore libres.

Remarque : Les optimisations de compilation ne permettent pas de déclencher les buts gelés dès l'unification de la tête de règle mais seulement sur le premier but de la queue non optimisé. En particulier, l'exemple suivant échouera :

```
> insert;
nombre(<i>) -> integer(i);
lier(<1>) ->;;
{}
> freeze(x, lier(x)) nombre(x);
>
```

En effet, les prédicats de test de type étant optimisés, le dégel de *lier(<i>)* ne se fera pas puisqu'il n'y a pas d'autres buts dans la queue de *nombre/1*.

2.3. A propos des arbres infinis

Ce paragraphe donne quelques informations pratiques sur l'utilisation dans Prolog II des arbres infinis. Tout d'abord, il faut remarquer qu'un certain nombre de règles prédéfinies ne travaillent que sur des arbres finis. En particulier, *in* ne peut pas lire d'arbres infinis, et *assert* ne peut pas ajouter de règles contenant des arbres infinis.

infinite

no_infinite

Ces règles servent à définir le type d'impression de résultat utilisé: il faut avoir activé l'option *infinite* si l'on désire imprimer des solutions contenant des arbres infinis.

Voici un exemple qui utilise *freeze* et qui vérifie qu'un arbre est toujours fini. Ceci est un moyen indirect de faire le test dit d'*occur_check*. Ce programme vérifie que l'on n'a pas deux fois le même sous-arbre dans chaque branche de l'arbre.

```
arbre_fini(x) -> branche_finie(x, nil);

branche_finie(x, l)  -> freeze(x, branche_finie'(x, l));

branche_finie'(x, l)  ->
   hors_de(x, l) domine (x, l') branches_finies(l', x.l);
```

```
branches_finies(nil, l) -> ;
branches_finies(x.l', l) ->
   branche_finie (x, l) branches_finies (l', l) ;

domine(x1.x2, x1.x2.nil)  -> ! ;
domine(x, x') -> tuple(x) ! split(x, x') ;
domine(x, nil) -> ;
```

infinite_flag

Symbole dont la valeur (*0* ou *1*) indique si *infinite* ou *no_infinite* est actif. Cette valeur peut être testée par l'intermédiaire de la règle prédéfinie *val*.

equations(t, t', l)

Cette règle prédéfinie sert à transformer un arbre fini ou infini *t* en une liste d'équations *l* qui a comme solution *t*. *t'* indique la variable de *l* représentant la racine de l'arbre *t*. C'est cette primitive qui est utilisée pour imprimer les solutions lorsque l'option *infinite* est active. Regardons une utilisation sur l'exemple suivant :

```
> infinite eq(x,ff(ff(x))) equations(x,t',l) out(t')
outm(" : ") outl(l) ;
v131 : eq(v131,ff(v131)).nil
{x=v131, v131=ff(v131), l=eq(t',ff(t')).nil}
> eq(x,ff(ff(x)));
{x=v120,v120=ff(v120)}
```

Mais une des utilisations principales de cette primitive est le fait de pouvoir ajouter des arbres infinis. Nous utilisons la primitive *assert* (chapitre suivant) qui permet d'ajouter une règle.

```
ajout_infini(t) ->
    equations(t,t',l)
    assert(arbre(t'),l) ;

> infinite eq(x, ff(ff(x))) ajout_infini(x) ;
{x=v131, v131=ff(v131)}
> list(arbre/1) ;
arbre(x11) -> eq(x11, ff(x11)) ;
{}
> arbre(x) ;
{x=v131, v131=ff(v131)}
```

2.4. Quelques conseils pour la programmation récursive

Le système Prolog II+ comporte un récupérateur de mémoire (*garbage collector*) automatique. Ce "ramasse-miettes" est très performant et est capable de ne garder dans les piles que les données dont vous avez effectivement besoin; ceci permet d'utiliser à fond les définitions récursives de programmes. Malgré tout, certaines techniques de programmation peuvent vous aider à récupérer encore plus d'espace. Pratiquement, il s'agit d'utiliser adroitement la *coupure des points de choix* ainsi que de tirer un maximum de profit de l'*optimisation de récursion et d'appel terminal* que fait le compilateur Prolog II+. L'utilisation de ces techniques permet l'écriture de programmes qui se rappellent eux-mêmes indéfiniment, sans jamais déborder.

Ceci sera plus clair à partir d'un exemple:

```
> insert;
répéter -> out(1) répéter ;;
{}
> répéter ;
1111111111111111111111...
```

est un programme tournant indéfiniment sans déborder, car la récursion se fait sur le *dernier* littéral de la règle. Par contre le programme suivant débordera, car la récursion accumule les littéraux à effacer (récursion *non terminale*):

```
répéter -> out(1) répéter out(2);
> répéter ;
1111111111111111111111...
-> DEBORDEMENT[1] PILE DE RECURSIVITE
>
```

de même le fait qu'il reste des choix à la règle exécutée accumule de l'information à garder. L'effacement du but avec la définition qui suit provoquera un débordement par accumulation de points de choix:

```
> insert;
répéter -> répéter ;
répéter ->;;
{}
> répéter fail;
-> DEBORDEMENT PILE DE RECURSIVITE
>
```

En revanche le programme suivant tournera indéfiniment (heureusement on peut l'interrompre avec *Control-C*!):

```
répéter -> ;
répéter -> répéter ;
```

Le coupe-choix "!" permet au ramasse-miettes de récupérer davantage d'espace, en supprimant des points de choix. Il faut cependant remarquer que si on le place en fin de règle, on fait disparaître la propriété de récursion terminale:

```
répéter -> out(1) répéter !;
répéter ->;
```

Le programme ci-dessus provoquera un débordement (le "!" est un littéral à effacer), alors que le programme ci-dessous tournera indéfiniment:

```
répéter -> out(1) ! répéter ;
répéter ->;
```

[1] Si le système de réallocation automatique n'est pas désactivé au démarrage de Prolog, un certain nombre de réallocations se produiront avant le débordement.

2.5. Les meta-structures

2.5.1. Création

Il est possible en PrologII+ d'attacher à une variable une liste de termes. On parlera alors de *méta-structure* ou *variable attribuée* et de *termes attribués*.

La particularité de ces variables est que l'unification avec un terme connu ou bien avec une autre variable attribuée est remplacée par des appels à des prédicats utilisateurs.

En effet, une tentative d'unification d'une variable attribuée avec un terme connu est remplacée par autant d'appels que de termes dans la liste attachée à cette variable. De la même manière, une tentative d'unification de deux variables attribuées se traduira par des appels successifs à tous les prédicats contenus dans chacune des deux listes.

Les paramètres de chacun de ces prédicats sont imposés: le premier paramètre sera la variable attribuée elle même, le second sera le terme avec lequel on essaie de l'unifier, et le troisième sera le terme attribué correspondant dans la liste.

Le lien d'un terme avec une variable peut s'effectuer au moyen de deux prédicats prédéfinis: *new_tlv/3* et *add_tlv/3* (TLV: Term Linked to Variable).

new_tlv(v, t, i)

Attache à la variable *v* le terme *t*. La liste de termes attribués à la variable *v* sera donc réduite à un seul élément: le terme *t*. Si *v* était déja une variable attribuée, l'ancienne liste est perdue. Si *v* n'est pas une variable, une erreur est générée. Le troisième argument *i* doit être un identificateur: il s'agit du nom de la règle utilisateur décrite plus haut (d'arité 3) qui sera appelée lors d'une tentative d'unification de la variable *v* avec un terme connu ou bien avec une autre variable attribuée.

add_tlv(v, t, i)

Ajoute à la variable *v* le terme *t*. La liste de termes attribués à la variable *v* sera donc augmentée d'un élément: le terme *t*. Si *v* n'était pas déja une variable attribuée, ce prédicat a le même effet que le précédent. Si *v* n'est pas une variable, une erreur est générée. Le troisième argument *i* a la même signification que précédemment.

L'ordre dans lequel les prédicats utilisateurs seront appelés n'est pas spécifié.

Exemple:

```
> insert;
rule1(V,T,A) -> outm("Rule1 ") out(T) outm(" ") outl(A);
rule2(V,T,A) -> outm("Rule2 ") out(T) outm(" ") outl(A);
rule3(V,T,A) -> outm("Rule3 ") out(T) outm(" ") outl(A);
;
{}
> new_tlv(V,1.2.nil, rule1) add_tlv(V,3.4.nil,rule2)
```

```
  add_tlv(V,5.6.nil,rule3) eq(V,foo);
Rule1 foo 1.2.nil
Rule2 foo 3.4.nil
Rule3 foo 5.6.nil
{V@rule1(1.2.nil).rule2(3.4.nil).rule3(5.6.nil).nil}
```

Une combinaison de ces deux prédicats peut être simplement obtenue par un appel au prédicat :

set_tlv(v, l)

où *l* doit être une liste d'éléments de la forme *i(t)*, *i* et *t* ayant la même signification que dans les prédicats précédents. Si la liste *l* comporte N éléments, l'appel à ce prédicat est équivalent à un premier appel à *new_tlv/3,* suivi de N-1 appels à *add_tlv/3*.

Exemple:

```
> set_tlv(X,foo(1).aa(2).nil);
{X@foo(1).aa(2).nil}
```

2.5.2. Récupération

La liste des termes attribués à une variable peut être obtenue au moyen du prédicat

get_tlv(v, l)

Unifie *l* avec la liste formée des termes attribués à la variable *v*, chacun de ces termes étant encapsulé par le prédicat utilisateur appelé lors d'une tentative d'unification de cette variable avec une autre variable attribuée ou un terme connu. Cette liste *l* est donc de la forme: *id1(t1).id2(t2)....nil* avec l'association de chaque identificateur *id* (prédicat utilisateur) au terme correspondant. Si *v* n'est pas une variable, *l* est unifiée avec *nil*.

Exemple:

```
> new_tlv(V,1.2.nil, rule1) add_tlv(V,3.4.nil,rule2)
  add_tlv(V,5.6.nil,rule3) get_tlv(V,T);
{V@rule1(1.2.nil).rule2(3.4.nil).rule3(5.6.nil).nil,
T=rule1(1.2.nil).rule2(3.4.nil).rule3(5.6.nil).nil}
```

2.5.3. Unification forcée

Une unification classique (sans appel aux prédicats utilisateurs) peut être forcée entre une variable attribuée et un terme connu ou bien une autre variable attribuée. Cette unification s'effectue au moyen d'un prédicat spécial:

unify_tlv(t1, t2)

Réalise une unification ordinaire entre les deux termes *t1* et *t2*. Voyons tous les cas possibles:

- Si ni *t1* ni *t2* ne sont des variables attribuées, est équivalent à *eq/2*.

- Si *t1* est une variable attribuée:

- Si *t2* est une variable ordinaire, *t2*, unifié avec *t1*, deviendra donc une variable attribuée.

- Si *t2* est un terme connu, *t1,* unifié avec *t2* deviendra donc ce terme connu (la liste de termes attribués sera alors perdue).

- Si *t2* est aussi une variable attribuée, après unification *t1* et *t2* désigneront une variable attribuée ayant pour liste de termes attribués la concaténation des deux listes initiales (celle de *t1* et celle de *t2*).

Exemple:

```
> new_tlv(V,1.2.nil,foo) unify_tlv(V,3);
{V=3}
> new_tlv(V,1.2.nil,foo) unify_tlv(Z,V);
{V=Z,V@foo(1.2.nil).nil,Z@foo(1.2.nil).nil}
> new_tlv(V,1.2.nil,foo) new_tlv(X,3.4.nil,faa)
  unify_tlv(V,X);
{V=X,V@foo(1.2.nil).faa(3.4.nil).nil,
   X@foo(1.2.nil).faa(3.4.nil).nil}
```

2.5.4. Sorties

Les impressions des variables attribuées d'une question se fait d'une manière spéciale : <nom de variable>@<terme>

Le terme qui suit @ est la liste des termes attribués à la variable, présentée de manière identique au prédicat *get_tlv/2*.

Exemple:

```
> new_tlv(x,zz,foo);
{x@foo(zz).nil}
```

2.5.5. Exemple

Ce programme attache à une variable l'ensemble de ses valeurs possibles. S'il ne reste plus qu'une seule valeur, elle est affectée à la variable.

```
insert;
% Si X est une nouvelle variable, lui attache l'ensemble
% de valeurs L, sinon lui attache l'intersection de L avec
% l'ensemble déja attaché
be_in(X,L) -> get_tlv(X,L1) be_in(L1, X, L);
be_in(nil,X,L) -> ! new_set(X,L);
be_in(L1,X,L) -> get_values(L1, L2) inters(L,L2,L3)
                 attach_or_affect(X, L3);

% attache à X la liste L de ses valeurs possibles
new_set(X,L) -> new_tlv(X,L,verify_in);

% Récupère la liste des valeurs possibles
get_values([],[])->!;
get_values(t_ag(L1).L2, L)-> get_values(L2,L3)
                             conc(L1,L3,L);

% concatenation de 2 listes
conc(nil,L,L)->;
```

```
conc(X.L1,L2,X.L3) -> conc(L1,L2,L3);

attach_or_affect(X,[F]) -> !
   unify_tlv(X,F);% une seule valeur possible -> on affecte
attach_or_affect(X,L) -> dif(L,nil)
   new_set(X,L);  % nouvel ensemble possible

% Le 3ème argument est l'intersection des 2 listes
inters([],L,[])->;
inters(x.l,l1,x.l2)-> member(x,l1) ! inters(l,l1,l2);
inters(x.l,l1,l2)-> inters(l,l1,l2);

% Predicat utilisateur
verify_in(V_arlibc, T_unifie, T_att) ->
            member(T_unifie,T_att)
            unify_tlv(V_arlibc, T_unifie);
;           % End of insertion
{}
> % quelques exécutions
be_in(x,[1,2,3,4]) be_in(x,[1,3,7]) be_in(x,[1,7,8]);
{x=1}
> be_in(x,[4,5,6]) be_in(x,[7,8,9]) ; % Echec
> be_in(x,[1,2,3,4,5]) be_in(y,[1,6,7,8]) eq(x,y);
{x=1,y=1}
> be_in(x,[1,2,3,4,5]) be_in(y,[1,2,6,7,8]) eq(x,y);
{x=1,y=1}
{x=2,y=2}
```

3. Structuration des règles et modification

3.1. Introduction
3.2. Terminologie
3.3. Syntaxe des identificateurs
3.4. Contexte de Lecture et Ecriture
3.5. Modules
3.6. Résumé ou Approche simplifiée des identificateurs, contextes et modules
3.7. Ajout, suppression et recherche de règles
3.8. Manipulation des modules compilés

Les notions de module, de famille d'identificateurs, et de contexte de lecture/écriture ont été introduites pour permettre:

- La modularité et l'indépendance des programmes, en permettant de partitionner l'ensemble des identificateurs manipulés.
- La gestion de programmes Prolog comportant un très grand nombre de règles. Ceci est fait en partageant l'ensemble des règles en différents modules.

3.1. Introduction

La notion de module vient de la nécessité de regrouper les règles d'un gros programme en ensembles fonctionnels que l'on appelle modules.

3.1.1. Qualification des prédicats

Afin de permettre une certaine indépendance entre l'élaboration des différents modules, il est souhaitable de pouvoir automatiquement rendre indépendants les noms des règles des différents modules.

La méthode la plus simple que l'on puisse imaginer est d'étendre les noms de règle en les préfixant systématiquement avec le nom du module : on a ainsi implicitement construit un partitionnement des règles calqué sur celui des noms de règle, la sémantique restant rigoureusement identique à celle d'un Prolog sans module. Le préfixe est alors appelé qualificateur.

```
<nom de règle> ::=    <prefixe> : <suffixe>
<prefixe> ::=  <lettre> <alpha num>
<suffixe> ::=  <lettre> <alpha num>
```

Exemples :

```
lexicon:data(1) ->;
lexicon:data(2) ->;
lexicon:pn(X) -> ...
...

grammar:sentence(X)  -> lexicon:pn(X1) grammar:sv(X2)...
grammar:error_message(M) -> sys:write(M);
grammar:data(8) ->;
...
```

Les noms de règles de modules différents sont par définition différents, puisqu'ils ont des qualificateurs différents (Il faut insister sur le fait que la suite prefixe+suffixe n'est pas une structure, mais constitue un identificateur insécable).

La notion de base est donc celle de partition des symboles de règle (ou noms des règles), qui induit une partition dans les règles correspondantes. Chaque partie est appelée un *module*. Cette notion est parfaitement dynamique: créer une règle avec un nom de règle appartenant à une partition non encore utilisée, correspond à créer un nouveau module.

Tous les modules sont situés sur le même plan, et il n'existe pas de notion de modules imbriqués.

Note
Pour compatibilité avec l'interpréteur Prolog II, la syntaxe des préfixes a été étendue de manière à pouvoir simuler une inclusion en permettant l'utilisation du séparateur ":" à l'intérieur du préfixe. Par exemple *Base:Normal:data* est un identificateur ayant pour préfixe *"Base:Normal"*, et pour suffixe *"data"*.

3.1.2. Qualification des foncteurs

Pour garder la symétrie entre les données et les programmes, le même partitionnement est utilisé pour les symboles de règles et les symboles de foncteurs. La qualification des foncteurs permet également de garantir que certains noms n'interféreront pas avec d'autres modules (désignation d'un fichier temporaire, type de donnée opaque, ...).

Pour clarifier les notations, il a été créé une notation spécifique pour certains foncteurs dits *génériques*. Ces foncteurs peuvent être utilisés comme noms symboliques "globaux" (au sens de identiques dans tous les modules).

```
<nom de foncteur> ::=  <nom de règle> |
                       <nom générique>

<nom générique> ::=     : <suffixe>
```

Exemples de noms génériques :

```
:john, :data, :sentence, :singular
```

Les identificateurs génériques correspondent à ceux qui sont utilisés par défaut pour les noms utilisateurs lorsque l'on ne fait pas de déclaration de module.

3.1.3. Simplification des notations

La spécification systématique des qualificateurs est quelque chose de relativement lourd qui peut facilement être évité dans les textes sources en indiquant le début et la fin des règles d'un module. Les qualifications peuvent alors être réalisées automatiquement lors de la lecture du module.

L'exemple précédent peut ainsi se noter dans le texte source :

```
module("lexicon");
data(1) ->;
data(2) ->;
pn(X) -> ...
end_module("lexicon");

module("grammar");
sentence(X) -> lexicon:pn(X1) sv(X2)...
error_message(M) -> sys:write(M);
data(8)...
end_module("grammar");
```

Tout identificateur noté sans le caractère ":" est dit écrit en notation simplifiée (ou non qualifiée). Il est alors représenté, dans le texte source, par son seul suffixe.

Avec cette seule convention, si le qualificateur est différent de celui du module, la forme complète (i.e. qualifiée) de l'identificateur de règle doit être spécifiée. Nous verrons dans les sections suivantes d'autres déclarations permettant de simplifier encore les notations.

La règle de qualification est la même pour les noms des règles et pour les noms des foncteurs.

Exemple en syntaxe Edinburgh:

```
module("test").
myprint(X) :- ...
data(1).
data(:john).
do_print1(L) :-
   C =.. [myprint|L],
   call(C).
do_print2(L) :-
   C =.. [write|L],
   call(C).
do_list_data :-
   listing(data/1).
end_module("test").
```

```
/* -- programme défini -- */
test:myprint(X) :- ...
test:data(1).
test:data(:john).
test:do_print1(L) :-
   C =.. [test:myprint|L],
   sys:call(C).
test:do_print2(L) :-
   C =.. [sys:write|L],
   sys:call(C).
test:do_list_data :-
   sys:listing(test:data/1).
```

3.1.4. Quelques règles simples d'utilisation

Une manière très simple d'utiliser les modules consiste à noter systématiquement avec des noms génériques les données que l'on veut noter dans plusieurs modules, et de préfixer systématiquement tous les noms d'appels de règles dans d'autres modules que le module courant et le module système. Exemple:

```
> insert;
module("lexicon");
```

```
np(:john) ->;
np(:marie) ->;
....
end_module("lexicon");

module("grammar");
nom_propre(X) -> lexicon:np(X);
...
end_module("grammar");
;
> grammar:nom_propre(X);
{X=john}
....
```

Lorsque l'on a un doute sur la qualification qui a été créée pour un identificateur, utiliser la règle prédéfinie *string_ident/3* vous permet toujours de voir quel est le préfixe associé.

Pour éviter d'avoir à qualifier des noms externes, on peut les spécifier dans l'en-tête du module, l'effet étant alors similaire à une déclaration d'importation dans un langage classique. Cela n'exclut cependant pas d'utiliser d'autres noms, pourvu qu'ils soient explicitement qualifiés.
L'exemple ci-dessus peut ainsi s'écrire:

```
module("grammar",["lexicon",["np"]]);
nom_propre(X) -> np(X);
...
end_module("grammar");
```

3.1.5. Modules et préfixes standard

Quand Prolog est lancé sans spécifier d'état initial, l'état initial standard de la mémoire de travail (fichier *initial.po*) est chargé.

L'état initial comprend ‘trois’ modules:

- Le module qui contient les règles prédéfinies, correspondant au préfixe "sys".
- Les modules qui forment le superviseur, dont les préfixes sont de la forme réservée "sys: …".
- Le module par défaut pour les programmes de l'utilisateur, correspondant au préfixe vide "".

3.2. Terminologie

De nombreuses incompréhensions ayant résulté de l'emploi des mêmes mots pour désigner à la fois une entité Prolog et la chaîne de caractères la représentant, nous essayerons de donner ici une terminologie précise. Si en Prolog normal l'ambiguïté n'est pas très gênante, elle amène cependant une extrême confusion dans le cas des modules: une séquence de caractères alphanumériques située dans un module M1, peut ne pas représenter la même entité que celle désignée par la même séquence de caractères située dans un module M2 (c'est à dire les entités correspondantes ne sont pas unifiables).

Identificateur
Un élément de l'ensemble des entités non structurées, dont la représentation externe répond à la syntaxe des identificateurs.

Représentation complète d'un identificateur
La séquence de caractères qui représente de manière unique et non ambiguë une entité de type identificateur, indépendamment des conventions de lecture / écriture. Par extension on parlera d'*identificateur complet* pour désigner cette séquence de caractères. Un identificateur complet est formé par la concaténation des caractères d'un *préfixe*, du caractère ":", et d'une abréviation d'identificateur appelée *identificateur abrégé*.

Famille d'identificateurs
On appellera *famille "p"* l'ensemble de tous les identificateurs possédant le préfixe *"p"*. Une famille pourra être désignée par la chaîne correspondant au préfixe des identificateurs qu'elle contient.
Ainsi *sys:out*, *sys:in* appartiennent à la famille "*sys*".
De même, *sys:env:files* appartient à la famille "*sys:env*", *:peter* appartient à la famille "".

Opération de lecture/écriture
Opération associant une entité Prolog (représentation interne) à une séquence de caractères (représentation externe) ou réciproquement.

Contexte de lecture et d'écriture
Règles de passage d'un identificateur abrégé à un identificateur complet (et vice-versa) lors d'une opération de lecture/écriture.

Représentation abrégée d'un identificateur
C'est une représentation externe sans préfixe d'un identificateur. La représentation complète peut être déterminée de manière non ambiguë à partir des conventions de lecture / écriture (ou contexte de lecture / écriture). Deux représentations abrégées identiques peuvent représenter des entités différentes lorsqu'elles sont prises dans des contextes différents, mais représentent toujours la même entité lorsqu'elles sont prises dans le même contexte de lecture / écriture.

Module "p"

Dans un programme, c'est à tout moment l'ensemble des règles et des faits dont l'identificateur d'accès a le préfixe "p" (les tableaux Prolog sont ici assimilés à des faits). Il s'agit donc d'une notion dynamique: les modules évoluent avec l'état du programme. On peut voir la notion de module ainsi définie, comme un regroupement des règles par famille des identificateurs d'accès.

3.3. Syntaxe des identificateurs

La syntaxe des identificateurs est étendue de manière à pouvoir les regrouper par leur *préfixe*. La syntaxe d'un *identificateur complet* est donc:

```
full_identifier =          prefix , prefix_limit , abbreviated_identifier ;
prefix_limit =             ":";
prefix =                   [ name, { prefix_limit, name }  ];
abbreviated_identifier =   name   ;
name =                     letter, { letter | digit | "_" } ;
```

Ainsi les identificateurs complets suivants sont corrects et représentent tous des identificateurs différents:

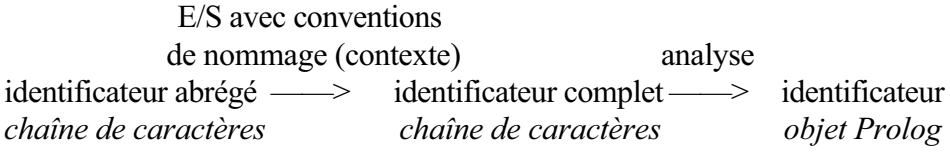
```
:peter                  grammar:plural
data:peter              grammar:name
sys:write               lexicon:name
Base:Normal:toto
```

L'identificateur suivant n'est pas un identificateur complet:

```
peter
```

> Note: il est important de pouvoir distinguer syntaxiquement les identificateurs complets des identificateurs abrégés. La détermination du nom complet à associer à un nom abrégé est réalisée à partir du contexte de lecture et d'écriture (cette notion est développée plus loin).

Il faut également souligner que le ":" n'est PAS un opérateur: le nom ne représente pas un objet sécable, seule la chaîne de caractères qui le représente est décomposable. Lorsqu'on parle de nom abrégé, on parle donc de la chaîne de caractères correspondante, seul le contexte de lecture permet de déterminer de quel objet Prolog il s'agit:



Les Prolog classiques peuvent facilement être intégrés dans ce schéma: la syntaxe des identificateurs correspond à la seule notation abrégée, et la convention d'extension est de les préfixer avec "". C'est la convention par défaut lorsqu'on démarre Prolog II+.

3.3.1. Paramétrage de l'écriture d'un identificateur complet

Jusqu'à présent et dans la suite de ce manuel, le caractère qui délimite le préfixe du suffixe, dans la représentation d'un identificateur complet, a toujours été présenté comme étant le caractère ":". C'est le caractère qui est choisi par défaut, pour tenir ce rôle.

Il peut toutefois être remplacé par un caractère graphique, en effaçant le prédicat prédéfini *set_prefix_limit/1*.

set_prefix_limit(s)

s doit être une chaîne d'un seul caractère, ce caractère doit être un caractère graphique. Il devient alors le caractère utilisé pour délimiter le préfixe du suffixe dans l'écriture d'un identificateur complet. Exemple :

```
> set_prefix_limit("$");
{}
> ident(sys$ident);
{}
> string_ident(x,y,a$a$ident);
{x="a$a",y="ident"}
```

prefix_limit(s)

s est unifié avec le caractère qui sert à délimiter le préfixe du suffixe dans l'écriture d'un identificateur complet. Exemple :

```
> prefix_limit(x) set_prefix_limit("$") prefix_limit(y);
{x=":", y="$"}
```

Le changement de l'écriture des identificateurs complets doit se faire en connaissance de cause et avec des précautions. En effet, si la définition du caractère de délimitation se fait dynamiquement, il faut prêter attention aux représentations des identificateurs qui apparaissent de manière statique, dans les données des programmes Prolog ou C. D'autre part, l'utilisation de la notation de préfixes emboîtés ("Base:Normal:Data") pour un module, peut faire changer le nom du module après utilisation de la primitive.

Une fonction C est fournie pour connaître le caractère en vigueur. Elle est décrite dans le chapitre 7 de ce manuel.

3.4. Contexte de Lecture et Ecriture

Un contexte de lecture/écriture définit les conventions de passage d'un identificateur abrégé à un identificateur complet lors d'une opération de lecture (et vice-versa lors d'une opération d'écriture). A un moment donné, un et un seul contexte est utilisé: c'est le *contexte courant de lecture/écriture.*

Un contexte est défini de manière abstraite par un couple:

(suite d'identificateurs complets, préfixe par défaut)

3.4.1. Lecture

Lors d'une opération de lecture d'un identificateur abrégé, on l'identifie avec le premier identificateur de la suite ayant la même représentation abrégée. S'il n'y en a aucun, le préfixe par défaut est appliqué.

Soit par exemple le contexte[1]:

("sys:out" "sys:outm" "m:out", "m1")

la lecture des séquences suivantes donne:

identificateur abrégé		identificateur complet
`out`	`-->`	`sys:out`
`mastic`	`-->`	`m1:mastic`
`outm`	`-->`	`sys:outm`

3.4.2. Ecriture

On écrit les noms avec la forme la plus abrégée que permet le contexte. Lors d'une opération d'écriture d'un identificateur de préfixe *p* et d'abréviation *a*, on écrit la représentation abrégée *a*

1. si l'identificateur figure dans la suite et est le premier de la suite à avoir l'abréviation *a* :
2. si le préfixe par défaut est *p* et aucun identificateur de la suite ne possède l'abréviation *a.*

Dans tous les autres cas on écrit la représentation complète.

Ces deux conditions assurent la réversibilité des opérations de lecture/écriture réalisées avec le même contexte. Par exemple, avec le contexte précédent, l'écriture des séquences suivantes donne:

---

1 La manière précise de noter ces spécifications en Prolog II+ sera donnée plus loin : cf. paragraphe 3.4.4.

identificateur complet		représentation externe
`sys:out`	`-->`	`out`
`m:out`	`-->`	`m:out`
`m1:mastic`	`-->`	`mastic`
`sys:outm`	`-->`	`outm`
`b:c:toto`	`-->`	`b:c:toto`

3.4.3. Notation simplifiée de la suite d'identificateurs d'un contexte

La plupart des programmes manipulent des règles prédéfinies, et nécessitent donc, pour pouvoir utiliser la représentation abrégée de leurs noms, que tous ceux-ci soient listés explicitement lorsqu'on définit un contexte les utilisant. C'est pour pouvoir simplifier la définition d'un contexte, qu'ont été introduites les notions de *suite explicite* et de *suite implicite*. La notation de la suite d'identificateurs du contexte est allégée en permettant de désigner *implicitement* un ensemble d'identificateurs, en spécifiant leur préfixe.

La suite d'identificateurs complets d'un contexte peut alors être elle-même représentée par un couple:

(suite explicite d'identificateurs, suite implicite d'identificateurs)

où la suite implicite représente des familles d'identificateurs du programme, désignées par leur préfixe. Un contexte est maintenant décrit abstraitement par une structure de la forme:

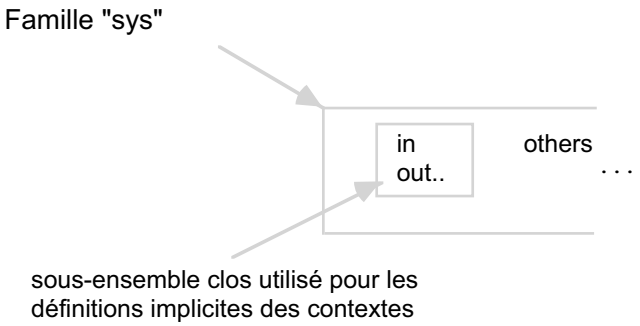
((suite explicite, suite implicite), préfixe par défaut)

Voici un exemple d'une telle description, pour la suite suivante: *"m1:out", "m2:peter", tous les identificateurs de la famille "m4", tous les identificateurs de la famille "sys"* :

(("m1:out" "m2:peter", "m4" "sys"), "a:b")

Il y a cependant un danger à confondre la notion de suite d'identificateurs pour un contexte de lecture/écriture, et celle de famille d'identificateurs d'un programme: l'ensemble des identificateurs ayant été utilisés dans un programme augmente avec le temps, et donc les opérations de lecture/écriture peuvent devenir non réversibles.

Pour résoudre ce problème, on introduit la notion de famille d'identificateurs *fermée pour le contexte* . C'est un sous-ensemble d'une famille de noms donnée, clos par une directive spéciale (*close_context_dictionary/1*), et qui ne peut être modifié que par des primitives ad hoc (*add_implicit/2*, *remove_implicit/2*).



Dans toute opération de contexte concernant une suite d'identificateurs désignés implicitement par leur préfixe, l'ensemble concerné est:

- le sous-ensemble clos, si la famille est fermée pour le contexte.
- l'ensemble de tous les identificateurs de la famille sinon. Cet ensemble est constitué de tous les identificateurs ayant apparu dans l'historique du programme avec le préfixe en question.

Un exemple de famille d'identificateurs fermée pour le contexte est la famille *"sys"*. Le sous-ensemble clos contient l'ensemble des identificateurs des règles et des fonctions prédéfinies de Prolog II.

Nous appellerons *contexte sain*, un contexte où toutes les suites implicites sont fermées pour le contexte. Un contexte sain garantit que les opérations de lecture et d'écriture sont répétables et réversibles.

Un exemple permettra de mieux comprendre ces notions. Soit le contexte abstrait suivant:

(("m1:out" "m2:peter", "sys"), "a:b")

La lecture de l'identificateur abrégé *out* donnera *m1:out* puisque le premier identificateur de la suite (explicite) a la même abréviation.

La lecture de l'identificateur abrégé *paul* donnera *a:b:paul* puisque cette abréviation ne correspond à aucun élément de la suite explicite, ni à aucun identificateur abrégé du sous-ensemble clos de la famille "sys".

La lecture de l'identificateur abrégé *nil* donnera *sys:nil* puisque cet identificateur fait partie du sous-ensemble clos de la famille fermée pour le contexte "sys".

L'écriture de *sys:out* donnera *sys:out* puisque le premier identificateur de la suite ayant l'abréviation *out* possède un préfixe différent.

3.4.4. Notation en Prolog

La suite explicite d'identificateurs d'un contexte est représentée en Prolog sous forme d'une liste alternée permettant de regrouper des noms de même préfixe:

```
Préfixe1.ListeDeNomsAbrégés1 …
Préfixen.ListeDeNomsAbrégésn.nil
```

Ainsi la suite explicite *"m1:out" "m1:outm" "jf:init" "m2:toto" "m2:peter"* sera représentée dans la notation infixée:

```
 "m1".("out"."outm".nil)
."jf".("init".nil)
."m2".("toto"."peter".nil)
.nil
```

en notation avec des crochets:

```
["m1",["out","outm"],
 "jf",["init"],
 "m2",["toto","peter"] ]
```

3.4.5. Primitives pour les contextes et les familles d'identificateurs

set_context(Identification, SuiteExplicite, SuiteImplicite, Défaut)

Cette commande permet à la fois de décrire un contexte, et de l'activer comme le contexte courant de lecture/écriture. Lors de l'effacement de cette commande, le contexte décrit est mémorisé dans le module d'environnement sous la forme:

```
sys:env:context(identification,
       suite_explicite,
       suite_implicite,
       préfixe_par_défaut) ->;
```

Lorsqu'un nouveau contexte est activé, l'ancien contexte n'est plus pris en compte.

L'argument *identification* doit être un identificateur ou une chaîne. Le deuxième argument doit être une liste alternée telle que décrite au paragraphe précédent. Le troisième argument est une liste de chaînes représentant des préfixes, et le quatrième argument est une chaîne définissant le préfixe par défaut.

Lorsque Prolog est lancé, le contexte courant de lecture écriture est le contexte sain défini par la commande:

```
set_context("user", [], ["sys"], "")
```

Pour un autre exemple, le deuxième contexte abstrait défini au paragraphe précédent, sera défini et activé par la commande:

```
> set_context(No2, ["m1",["out"],"m2",["peter"]],
["sys"], "a:b");
```

set_context(Identification)

Permet de réactiver un contexte en le désignant par son identification, une fois qu'il a déjà été défini.

Note 1:
Lorsqu'on définit des contextes ne contenant pas la famille "sys" *dans la suite implicite, il faut faire attention au fait que* nil *ne représente pas en général la marque de fin de liste* sys:nil *: cela dépend du préfixe par défaut et de la suite explicite. Des erreurs peuvent en résulter lors de l'utilisation de règles prédéfinies auxquelles on passe une telle valeur en argument. Il est conseillé dans ce cas d'utiliser le nom complet* sys:nil*, ou la notation équivalente [ ].*

Note 2:
Lorsqu'une règle contient une déclaration de contexte, celui-ci devient effectif au moment où la règle est exécutée (et non au moment ou la règle est lue). Dans l'exemple suivant, le contexte n'est pas modifié, et l'identificateur abrégé "peche" *est identifié à* ":peche"*, puisque c'est le contexte* "user" *qui est le contexte de lecture au moment de la lecture de la règle.*

```
Prolog II+, ...
>insert;
pomme ->
 set_context("new",["other",["peche"]],[],"def");
 peche;
peche ->;;
{}
>
```

current_context(t_0*)*
current_context(t_0*,*t_1*,*t_2*,*t_3*)*

Unifie t_0 avec le terme identifiant le contexte courant de lecture écriture, et t_1,t_2,t_3 avec, respectivement, le terme définissant la suite explicite, le terme définissant la suite implicite, et le préfixe par défaut de ce contexte.

close_context_dictionary(s)

Ferme la famille d'identificateurs *s* pour les opérations de contexte, dans son état au moment où la commande est exécutée. C'est à dire que tout nouvel identificateur de cette famille ne sera pas pris en compte dans les opérations de contexte.

add_implicit(s_1*,*s_2*)*

Rajoute au sous-ensemble clos de la famille fermée pour le contexte s_1, l'identificateur de nom abrégé s_2. Si *s1* n'est pas le préfixe d'une famille fermée pour le contexte, on a une erreur.

remove_implicit(s_1*,*s_2*)*

Opération inverse: enlève de l'ensemble d'identificateurs pris en compte pour le contexte de la famille fermée s_1, l'identificateur de nom abrégé s_2. Lorsqu'on applique cette commande à la famille *"sys"*, elle a le même effet, pour les identificateurs abrégés, que celui de banaliser des noms réservés.
Supposons que quelqu'un désire supprimer l'identificateur *sys:dictionary* de l'ensemble des identificateurs pris en compte pour les opérations de contexte concernant la famille "sys". L'exemple suivant montre le résultat obtenu :

```
> string_ident(p,a,dictionary)
  outl(sys:dictionary.dictionary);
dictionary.dictionary
{p="sys", a="dictionary"}
> remove_implicit("sys","dictionary");
{}
> string_ident(p,a,dictionary)
  outl(sys:dictionary.dictionary);
sys:dictionary.dictionary
{p="", a="dictionary"}
```

dictionary

Ecrit sur la sortie courante la forme abrégée des accès de la famille ayant le même préfixe que le préfixe par défaut du contexte de lecture/écriture.

dictionary(s1)

Unifie *s1* avec la liste des préfixes des modules utilisateur présents en mémoire.

dictionary(s1, t0)

Unifie t_0 avec la liste des formes *identificateur_d'accès/arité* des règles du module *s1*. Exemple :

```
> insert;
aa:bb(1) ->;
aa:cc(1,2) ->;;
> dictionary("aa",L);
{L=aa:bb/1.aa:cc/2.nil}
```

file_dictionary(f, l_pref)

Unifie *l_pref* avec la liste des préfixes des modules contenant des règles, présents dans le fichier *f*. Ce fichier doit contenir du code prolog compilé, *f* est donc un module objet ou un état binaire de démarrage, il est obtenu grâce à l'un des prédicats de sauvegarde *exit*, *save_state*, *save*.

3.5. Modules

La définition de modules en Prolog nécessite de pouvoir traiter commodément des cas aussi différents que ceux que l'on peut rencontrer, par exemple, dans un système de langue naturelle. On y trouve essentiellement deux modules: le premier est le lexique qui contient beaucoup de noms de la base de données et de la grammaire, et peu de noms de règles. Le deuxième est la grammaire qui contient essentiellement des noms qui lui sont propres et des appels au lexique. Considérons l'exemple suivant:

```
lexicon:pn(data:Peter, grammar:singular ) -> ;
lexicon:pn(data:John, grammar:singular ) -> ;
...
lexicon:verbø(data:smiles, grammar:singular ) -> ;
...
lexicon:adj(data:plural, grammar:plural ) -> ;
...
```

```
grammar:sentence(x,z) -> grammar:sn(x,y,g) grammar:vp
(y,z,g);
grammar:sn(x.l,l,g) -> lexicon:pn (x,g)
...
```

Définir des conventions qui permettent de simplifier l'écriture des identificateurs dans des cas aussi extrêmes est la raison profonde de l'élaboration du système de contexte de lecture décrit dans les chapitres précédents.

Un contexte permettant d'écrire le module *lexicon* avec seulement des identificateurs abrégés est facile à construire, il suffit de prendre comme préfixe par défaut celui couvrant le plus grand nombre d'identificateurs, et de représenter les quelques noms de règles dans la suite explicite:

```
set_context( :lex
           , ["grammar", ["singular", "plural"],
              "lexicon", ["pn", "verbø", "adj"]   ]
           , []
           , "data" )
```

3.5.1. Définition d'un module

Nous appellerons *module "p"*, dans un programme donné, et à un moment donné, le regroupement de tous les faits et règles, les fonctions évaluables, les identificateurs affectés (par *assign*), et les tableaux dont l'identificateur d'accès appartient à la famille *"p"* (c'est-à-dire possède le préfixe *"p"*). Les identificateurs de la famille *"p"* peuvent apparaître dans d'autres modules, mais pas en position identificateur d'accès (c'est à dire nom de prédicat de tête). La notion de module est dynamique: lorsqu'une nouvelle règle est créée ou supprimée, le module correspondant à son identificateur d'accès est modifié en conséquence.

On peut dire plus simplement, qu'un module est un regroupement de toutes les règles ou les faits dont l'identificateur d'accès a le même préfixe.

Un module peut exister sous forme de texte (avant lecture), ou sous forme interne (après lecture) on parlera de module source et de module objet pour désigner l'un et l'autre cas.

3.5.2. Module source

La définition d'un module source consiste à définir :
- un contexte de lecture/écriture pour le module,
- une règle, facultative, d'initialisation du module,
- les règles du module proprement dites.

Le contexte sert pour la lecture du module source, lorsqu'il est compilé. Il sert également pour l'écriture, lors de la décompilation du module (par les primitives *list* ou *editm*).

La règle d'initialisation du module, si elle existe[2], est exécutée automatiquement dès la fin de compilation du module avec le contexte courant d'exécution et non celui de lecture du module. Cette règle, ou ce paquet de règles, est sans argument et a pour identificateur d'accès : l'identificateur *ini_module* préfixé par le préfixe du module.

Un module source se présente donc comme une suite de règles comprises entre les deux déclarations:

```
sys:module( préfixe_des_accès, ... );
....
sys:end_module( préfixe_des_accès );
```

Remarque : si le module source n'a pas été construit de cette façon (les règles ont été compilées par des *insert* ou *assert,* dans un ordre quelconque par exemple), la primitive *editm* permet de le reconstruire sous cette forme.

Les déclarations *module* et *end_module* sont destinées au mode insertion de règles de Prolog (cf. *insert* paragraphe 3.6). Par conséquent, les éléments Prolog compris entre ces deux déclarations doivent être des règles (ou des commentaires, qui sont ignorés). La déclaration *module* annonce la définition d'un module et la déclaration *end_module* termine cette définition.

L'argument (chaîne Prolog) désignant le préfixe des accès dans la déclaration *module* doit être le même que celui figurant dans la déclaration *end_module*, et toutes les règles comprises entre ces deux déclarations doivent avoir un identificateur d'accès avec ce préfixe; cette contrainte permet de contrôler des erreurs de nommage à la lecture.

Naïvement, on peut dire que la déclaration module signifie *« les règles qui suivent, sont à insérer avec le contexte de lecture défini ici, et en vérifiant que les règles lues possèdent (ou acquièrent) bien le préfixe voulu »*

La forme complète de la déclaration *module* inclut la définition du contexte de lecture du module (voir la description des contextes §3.4.):

```
sys:module( préfixe_des_accès,
      suite_explicite,
      suite_implicite,
      préfixe_par_défaut)
```

Lorsque tous les arguments ne sont pas donnés, on a les défauts suivants (on supposera dans ce qui suit que le contexte de lecture de la déclaration *sys:module* permet de l'abréger):

```
module(p)      <=>    module( p, [], ["sys"], p )
module(p,s)    <=>    module( p,  s, ["sys"], p )
```

[2]Dans le cas où des modifications sont apportées au module, c'est à dire une partie du module est recompilée (par *reinsert* ou *insertz*) ou rechargée (par *reload*), la règle d'initialisation n'est exécutée que si elle fait partie des règles modifiées.

```
module(p,s,i)  <=>    module( p,  s,    i,    p )
```

La directive *sys:omodule*(*context*) en lieu et place d'une directive *module* utilise comme contexte de lecture du module, le contexte défini par la chaîne de caractères *context* . Ce contexte a été défini au préalable par un appel à la primitive *set_context* ou bien par une précédente directive *module*. Si ce contexte n'existe pas, la directive *omodule(context)* est équivalente à la directive *module(context)*, où les règles de défauts s'appliquent.

Le contexte de lecture défini par l'en-tête du module est utilisé pour la lecture de tout le reste du module, y compris la déclaration *sys:end_module*. Le contexte existant avant la lecture est ensuite rétabli.

Si au moment de la lecture du module, une règle ou un fait est créé avec un identificateur d'accès dont le préfixe est différent de celui du module, une erreur est générée.

Les deux modules de l'exemple listé en tête du chapitre peuvent ainsi s'écrire de différentes manières:

1er exemple

```
module("grammar");
sentence(x,z) -> sn(x,y,g) vp(y,z,g);
sn(x.l,l,g) -> lexicon:pn(x,g);
...
end_module("grammar");
```

2ème exemple

```
module("grammar", ["lexicon",["pn","verbø", ...]]);
sentence(x,z) -> sn(x,y,g) vp(y,z,g);
sn(x.l,l,g) -> pn(x,g);
...
end_module("grammar");
```

3ème exemple

```
module( "lexicon"
      , ["lexicon", ["pn", "verbø", "adj"],
         "grammar", ["singular",  "plural" ]]
      , [ ]
      , "data");
pn(Peter,singular) ->;
pn(John,singular) ->;
...
verbø(smiles,singular ) ->
...
adj(data:plural,plural) ->;
...
end_module("lexicon");
```

Les mots entrés par l'utilisateur seront lus avec le contexte défini par:

```
set_context( :my_entry, [ ], [ ], "data" ) ;
```

On peut, avec la notion de contexte de lecture du module, redéfinir de manière claire certaines des primitives utilisées à l'intérieur d'un module:

```
module("example", ["example", ["out"]] ) ;
out(x) -> nicer(x,y) sys:out(y);
...
explain(x) ->  out(x) TellSomethingAbout(x);
end_module("example");
```

Dans l'exemple suivant, la partie initialisation du module permet de rajouter la règle pertinente du module, en tant que règle prédéfinie Prolog, et permettra de l'appeler en utilisant l'identificateur abrégé, si le contexte courant contient bien le module *sys* des règles prédéfinies :

```
module("parser");
ini_module ->
   assert(sys:parse(x),parse(x,nil).nil)
   add_implicit("sys","parse");
parse(l1,l2) -> sentence(l1,l2);
...
end_module("parser");
```

3.5.3. Module Objet

La manipulation de modules objets concerne des représentations internes, la notion de contexte n'intervient donc pas dans ces opérations.

Le chargement et la sauvegarde d'un ou plusieurs modules objets se font par les primitives *load* et *save*.

Si un état binaire de démarrage ou bien un fichier de modules objets contient un module *"m"* qui possède une règle avec le nom *m:ini_module,* alors celle-ci est automatiquement exécutée après le chargement de l'état binaire ou du fichier objet.

Lorsqu'un module objet est sauvé sur un fichier, il est sauvé avec un dictionnaire permettant de renommer les préfixes au chargement. Il est ainsi possible de résoudre les problèmes de conflits de noms quels qu'ils soient, et de construire des programmes n'interférant pas avec les données qu'ils manipulent, ou au contraire de définir des modules de type base de données et lexique, dont les noms peuvent être fusionnés au chargement.

3.6. Résumé ou Approche simplifiée des identificateurs, contextes et modules

3.6.1. Identificateurs

Les identificateurs sont des objets Prolog qui peuvent servir à la construction d'autres objets Prolog. On distingue deux modes d'utilisation des identificateurs :

- comme prédicat : sert à définir une règle (§3.1.1.)
- comme foncteur : sert de donnée (§3.1.2.)

Dans tous les cas, la syntaxe d'un identificateur Prolog (§3.3.) est :

prefixe**:**suffixe

prefixe et suffixe sont des suites de caractères en accord avec la syntaxe choisie (Prolog II ou Edinburgh), : peut être remplacé par un caractère graphique (cf. set_prefix_limit).

Cette syntaxe a l'avantage de pouvoir désigner un gros ensemble d'identificateurs simplement d'après une particularité de leur nom.

On appellera *famille d'identificateurs de préfixe "untel"*, tous les identificateurs (prédicats ou foncteurs) dont le nom s'écrit **untel:**suffixe (§3.2.).

On appellera *module de préfixe "untel"*, ou par extension *module "untel"*, toutes les règles et/ou tableaux et/ou variables statiques dont l'identificateur d'accès s'écrit **untel:**suffixe (§3.2.).

Astuce : si vous voulez un identificateur invariant pour tous les programmes, simple à écrire et facile à retenir, choisissez pour cet identificateur le préfixe vide. Il est appelé dans les paragraphes précédents (§3.1.2.): foncteur générique.

3.6.2. Notation abrégée et contextes

On rencontre très fréquemment des situations où la grande majorité des identificateurs que l'on doit traiter, s'écrit toujours avec le même préfixe. On voudrait donc se passer de répéter ce préfixe et avoir une représentation d'identificateur sous forme abrégée.

Dans cette optique, pour simplifier la mise en œuvre et la lisibilité des programmes, il existe des conventions pour une représentation simplifiée des identificateurs, qui déterminent de manière unique la donnée Prolog. Ces conventions sont appelées contextes de lecture/écriture (§3.4.).

La conversion entre le codage interne de la donnée et sa représentation externe est réalisée uniquement au cours des opérations de lecture et d'écriture.

Un contexte de lecture/écriture permet de dire :

sauf contre ordre, toutes les représentations d'identificateurs abrégées lues doivent prendre le préfixe par défaut *pref*, exceptées les représentations abrégées suivantes *id1*, *id2*, *id3*, … qui prennent respectivement les préfixes *pref1*, *pref2*, *pref3*, … . Et pour tout ce qui doit être écrit, c'est à dire lorsque l'on doit construire la représentation la plus concise possible, faire en sorte qu'après relecture, on obtienne l'identificateur Prolog qu'on voulait écrire.

Un contexte désigne donc quel préfixe doit être attribué à la représentation simplifiée d'un identificateur pour obtenir sa représentation complète. D'où la définition du contexte par :

- une liste d'attributions explicites de préfixes,
- un préfixe par défaut pour les autres cas.

Ce contexte est complètement défini, on dira que c'est un *contexte sain*.

Propriétés:
- La liste des attributions explicites est traitée de gauche à droite.
- Si une représentation abrégée apparaît plusieurs fois dans la liste on lui attribuera le premier préfixe associé (c'est à dire le plus à gauche).
- Toutes les chaînes déclarées dans le contexte n'ont pas besoin de correspondre à des identificateurs existants. Si Prolog doit créer ou lire un identificateur, il consultera cette liste de représentations potentielles.

Pour abréger d'avantage l'écriture (même dans la définition du contexte), si on veut faire apparaître dans la liste explicite toute une famille d'identificateurs, au lieu de la mentionner de manière exhaustive, il suffit simplement de spécifier son nom. C'est alors considéré comme une attribution implicite, équivalant à calculer la liste explicite qui en découle et l'insérer dans la liste existante, au moment de chaque opération d'attribution de préfixe.

Propriétés:
- La liste des attributions implicites est traitée après la liste explicite.
- La liste des attributions implicites est traitée de gauche à droite.

D'où la définition du contexte par :
- une liste d'attributions explicites de préfixes,
- une liste d'attributions implicites de préfixes,
- un préfixe par défaut pour les autres cas.

Et sa définition en Prolog (§3.4.4. et §3.4.5.):

```
set_context(Id, ListeExplicite, ListeImplicite, Defaut);
```

Ce contexte devient alors dépendant du temps, dans la mesure où les familles désignées pour une attribution implicite peuvent grandir avec le temps.

Pour se ramener à un contexte sain, et supprimer cette partie inconnue dans le cas d'attributions implicites, il est nécessaire de 'fermer' la famille. Cela signifie : au moment de la fermeture de la famille, la liste explicite qu'elle représente est calculée et mémorisée une fois pour toute. Le contexte est à nouveau complètement défini et invariant dans le temps.

3.6.3. Modules

Prolog étant incrémental, sans type de données, il n'y a aucune contrainte sur l'ordre d'insertion des règles.

Vous pouvez toutefois grouper toutes les règles d'un même module et les définir en même temps. Vous pouvez également définir des conventions de lecture de ce groupe de règles différentes de celles du contexte d'exécution. Ceci permet de rendre la définition du module autonome si le contexte est sain.

La définition d'un module peut donc se faire par l'énoncé de:
- un contexte de lecture/écriture du module,
- un ensemble de paquets de règles,

- éventuellement une règle d'initialisation du module, à exécuter juste après compilation ou chargement des règles.

La définition d'un module peut se faire en mode insertion, en utilisant la déclaration *module* ou *omodule* (§3.5.2):

```
insert;                              insert;
module(p,e,i,d);     équivaut à      ->set_context(p,e,i,d);
                                     ->mode_verif_prefixe(p);
regles…                              regles…
end_module(p);                       ->exec(p:ini_module);
                                     ->restaure_context;
…                                    …
;                                    ;
```

Attention : si vous utilisez dans le contexte d'un module, des conventions implicites, même pour un contexte sain, il faut que la famille soit déjà définie pour que les attributions implicites se fassent. Dans la même optique, si vous faites appel, dans la règle d'initialisation d'un module, à des règles externes au module, il faut que ces règles soient déjà définies pour que l'initialisation se termine.

Cela signifie que, bien qu'il n'y ait pas de contraintes sur l'ordre des paquets de règles, il peut y avoir des contraintes sur l'ordre de compilation ou de chargement des modules.

Il est également important de noter que les notions de fichier et de module sont complètement indépendantes. Un même fichier source peut contenir plusieurs modules, et un même nom de module peut apparaître dans plusieurs fichiers chargés dans la même session.

3.7. Ajout, suppression et recherche de règles

Le système Prolog II+ comporte un compilateur qui traduit vos programmes sources écrits en Prolog dans un langage objet dont le «niveau» (i.e. : le degré de proximité avec la machine hôte) varie d'une machine à une autre, mais qui est dans tous les cas bien plus efficace que la simple interprétation du code source Prolog. En contre partie, il apparaît une certaine difficulté à restituer exactement le texte source de certaines règles, à cause de la disparition des noms des variables.

Le compilateur Prolog II+ est *incrémental* (la traduction se fait règle par règle) et *transparent* (chaque règle est compilée et indexée dès sa saisie, sans qu'il y ait besoin de composer une quelconque commande particulière).

Par défaut Prolog II+ réalise des optimisations à la compilation de certaines règles : les expressions arithmétiques, les tests de type, ou encore les prédicats prédéfinis *val*, *assign* et *block*. Ces optimisations permettent une exécution plus efficace de ces règles mais ont des conséquences sur le fonctionnement de certains autres prédicats prédéfinis (tels que *rule*, *debug* ou *freeze*). Il existe alors une option d'activation de Prolog II+ qui permet de supprimer ces optimisations de compilation (voir à ce sujet le Manuel d'Utilisation).

Pour minimiser le temps mis pour choisir une règle qui servirait à effacer un but particulier, Prolog II+ au moment de la compilation d'un paquet de règles, crée une 'image' du paquet dans laquelle il regroupe toutes les règles dont le premier argument est de type et/ou de valeur identique. Il réalise ainsi plusieurs groupes de règles disjoints et pour un même but à effacer, les règles d'un seul des groupes seront choisies, ou bien aucunes. Ce traitement est appelé l'indexation. Supprimer l'indexation revient à laisser toutes les règles du paquet valides pour un choix.

Par convention on utilisera, dans un programme et dans le texte qui suit, le terme *identificateur/entier* pour désigner un paquet de règles dont la tête a pour identificateur d'accès *identificateur*, et *entier* comme nombre d'arguments.

Avant toute chose, il est nécessaire de citer deux conséquences très importantes de la modification de paquets de règles, qui vont influer sur le comportement du programme à l'exécution.

1. La modification (ajout ou suppression de règle(s)) d'un paquet de règles compilées supprime son indexation. Pour l'indexer à nouveau, utiliser la primitive *index*.

2. La modification d'un paquet de règles en cours d'exécution d'une de ses règles (choix encore en attente) entraîne un comportement d'exécution (pour le backtracking ou pour un prochain appel) extrêmement dépendant de la configuration des règles (dépendant de la structure du paquet, de la règle courante, de la règle modifiée).

Dans la mesure du possible, il est préférable d'épuiser les choix avant de modifier les règles ou, si ce n'est pas le cas, de supprimer l'indexation avant l'exécution et la modification des règles compilées.

Toujours par souci d'améliorer les performances, il a été introduit dans Prolog II+, un nouveau 'type' de règles, appelées 'faits non compilés'. Ceci dans le but d'optimiser les problèmes de gestion d'informations par manipulation dynamique de faits. En effet pour vérifier un fait, il est inutile d'optimiser l'exécution du corps de la règle puisqu'il est inexistant, par contre il est important d'améliorer les accès aux arguments.

Prolog II+ va installer pour les 'faits non compilés' un processus d'indexation à partir de tous les arguments du fait, pour permettre au moment de l'exécution un accès quasi-direct sur les faits adéquats. Il n'est pas possible de supprimer l'indexation installée sur des 'faits non compilés'.

assert(t, q)
asserta(t, q)

> Compiler et ajouter une règle, au début de son paquet.
> *t* doit être un terme pouvant être une tête de règle, c'est-à-dire un identificateur ou un n-uplet dont le premier argument est un identificateur (i.e.: $<ident, arg_1, \ldots arg_n>$ ou $ident(arg_1, \ldots arg_n)$, ces deux notations étant équivalentes). *q* doit être une liste de termes.

L'effacement de *assert(t, q)* compile la règle *t -> q* et l'ajoute *au-dessus* du groupe de règles ayant le même «nom» que *t*, c'est-à-dire au début du paquet correspondant à *t*. Par exemple, les deux commandes suivantes

```
assert(conc(e.x, y, e.z), conc(x, y, z).nil);
assert(conc(nil, y, y), nil);
```

tapées dans cet ordre, ont pour effet l'ajout du programme

```
conc(nil, y, y) ->;
conc(e.x, y, e.z) -> conc(x, y, z);
```

Ne permettent pas d'ajouter des règles qui contiennent des arbres infinis; cependant cela peut être fait avec la règle prédéfinie *equations* (Voir § 2.3).

assert''(t, q)
assertz(t, q)

Compiler et ajouter une règle, à la fin de son paquet.
Même fonctionnement que *assert*, mais l'ajout se fait au-dessous, et non au-dessus, du paquet correspondant à *t*. Exemple: les deux commandes suivantes, dans l'ordre indiqué, produisent l'insertion du même programme *conc* que ci-dessus :

```
assert''(conc(nil, y, y), nil);
assert''(conc(e.x, y, e.z), conc(x, y, z).nil);
```

Ne permettent pas d'ajouter des règles qui contiennent des arbres infinis; cependant cela peut être fait avec la règle prédéfinie *equations* (Voir § 2.3).

assertn(t, q, n)

Compiler et ajouter une règle, à la n-ième position dans le paquet.
Même fonctionnement que *assert*, mais l'ajout se fait à la *n*-ième position dans le paquet correspondant à *t*. Exemple:

```
assert(myrule(2),nil);
assertn(myrule(0),nil,1);
assertn(myrule(1),nil,2);
```

tapées dans cet ordre, ont pour effet l'ajout du programme

```
myrule(0) ->;
myrule(1) ->;
myrule(2) ->;
```

Ne permet pas d'ajouter des règles qui contiennent des arbres infinis; cependant cela peut être fait avec la règle prédéfinie *equations* (Voir § 2.3).

current_predicate(i/a)

Tester la présence d'une règle.
S'efface s'il existe une règle d'identificateur d'accès *i* et d'arité *a*. Si *a* est une variable, et *i* est connu, énumère successivement toutes les valeurs de *a* correspondant à une règle d'accès *i*. Si *i* n'est pas connu (i.e. est une variable), unifie l'argument successivement avec toutes les formes *i/a* des règles du module déterminé par le préfixe par défaut du contexte courant. *i* et *a* ne doivent pas être libre en même temps.

discontiguous(i/a)

Permettre la non continuité d'un paquet de règles.
Lorsqu'une règle d'identificateur d'accès *i* et d'arité *a* sera compilée au moyen du prédicat *insert*, elle sera ajoutée à la fin de son paquet. Cette directive permet de scinder les paquets de règles au sein d'un même ou de plusieurs fichiers. Noter que cette directive n'a aucun effet sur une compilation au moyen du prédicat *reinsert*. qui écrasera le paquet de règles existant par le nouveau rencontré. Ceci est une directive de compilation et non pas un prédicat, c'est à dire qu'elle peut figurer au milieu d'un source que l'on compile mais ne peut se trouver dans une queue de règle.

dynamic(i/a)

Déclarer un paquet de règles dynamiques.
Tous les paquets de règles Prolog II+ sont dynamiques, c'est à dire qu'ils peuvent subir des modifications. Par conséquent, cette directive n'a aucun effet et n'existe que par compatibilité avec la norme Prolog. Ceci est une directive de compilation et non pas un prédicat, c'est à dire qu'elle peut figurer au milieu d'un source que l'on compile mais ne peut se trouver dans une queue de règle.

ensure_loaded(f)

S'assurer d'une seule inclusion de la compilation d'un fichier.
Réalise la compilation du fichier *f* à l'endroit où se trouve la directive, si ce fichier n'a pas déjà été compilé au cours de cette même phase de compilation, sinon ne fait rien. En fin de compilation de plus haut niveau d'imbrication, et donc pour une nouvelle phase de compilation, le fichier n'est plus considéré comme ayant été compilé. La compilation du fichier *f* s'effectue suivant le même mode que celui piloté par le prédicat de compilation de niveau supérieur (*insert*, *reinsert*, *insertz*).
Ceci est une directive de compilation et non pas un prédicat, c'est à dire qu'elle peut figurer au milieu d'un source que l'on compile mais ne peut se trouver dans une queue de règle.

fasserta(t)
fassertz(t)

Ajouter un fait non compilé.
Ajoute un fait non compilé en début (*fasserta*) ou en fin (*fassertz*) de son paquet. L'assertion est extrêmement rapide et permet de gérer de très grosses bases de faits. De plus, les indexations installées sur ces faits, par un appel préalable obligatoire à la primitive *init_fassert* permettent un accès très performant. Le terme *t* doit avoir la forme d'une tête de règle, c'est à dire un n-uplet dont le premier élément est un identificateur. Pour un même paquet, on ne peut mélanger des règles compilées (*assert, insert*) et des faits non compilés (*fassert*). Exemple:

```
> fasserta(myrule(1,2,3,4.nil,<44,"abc">,6,1.2.2e0.nil));
```

freplace(t, n_arg, t1)

Remplacer un argument non indexé dans un fait non compilé.
Remplace dans les faits, créés préalablement par *fasserta* ou *fassertz*, s'unifiant avec *t* l'argument de rang *n_arg* par le terme *t1*. Cet argument ne doit pas intervenir dans une combinaison d'indexation (cf. prédicat *init_fassert*). Exemple:

```
> freplace(myrule(1,2,3,x,y,z,t), 7, 345);
```

freplace(i/a, n_reg, n_arg, t1)

Remplacer un argument non indexé dans un fait non compilé.
Remplace dans le fait de rang *n_reg* du paquet d'identificateur *i* et d'arité *a*, l'argument de rang *n_arg* par le terme *t1*. Ce fait aura dû être créé par l'un des prédicats *fasserta* ou *fassertz*. Cet argument ne doit pas intervenir dans une combinaison d'indexation (cf. prédicat *init_fassert*). Exemple:

```
> freplace(myrule/7, 2, 7, 345);
```

fretract(t)

Supprimer rapidement un fait non compilé correspondant à un modèle.
S'efface autant de fois qu'il existe de faits s'unifiant avec *t*. La recherche de tels faits est très rapide et profite des optimisations mises en place pour l'accès aux faits non compilés. Ces faits auront dû être créés par l'un des prédicats *fasserta* ou *fassertz*. Dès qu'un fait s'unifiant avec *t* est trouvé, il est supprimé.
Si un fait qui convient, est invisible à la décompilation, il est supprimé, et la primitive s'efface sans unifier les variables libres de *t*.
Exemple:

```
> fretract(myrule(1,2,3,x,y,z,t));
```

fretractall(t)

Supprimer rapidement des faits non compilés correspondant à un modèle.
Supprime, en une fois, de manière très efficace tous les faits s'unifiant avec *t*. Ces faits doivent être des faits non compilés créés par l'un des prédicats *fasserta* ou *fassertz*. t doit permettre d'identifier un index pour cette base de fait, de la manière suivante: tous les arguments non libres du but doivent être atomiques, de plus ils désignent une combinaison d'arguments qui doit correspondre à une combinaison d'indexation définie pour cette base de faits. Dans le cas contraire une erreur est générée. Exemple:

```
> init_fassert(bb/3,(1.2.3).(1.2).nil);
{}
> fasserta(bb(1,2,3));
{}
> fasserta(bb(1,2,x));
{}
> fasserta(bb(1,2,1.2));
{}
> fasserta(bb(1,1,1.1));
{}
> fretractall(bb(1,x,3));
```

```
-> <bb(1,v36,3)> : LA COMBINAISON CORRESPONDANT AUX ARGUMENTS
LIES N'EXISTE PAS

> fretractall(bb(1.2,2,x));

-> <bb(1.2,2,v59)> : UN ARGUMENT INDEXE N'EST PAS ATOMIQUE

> fretractall(bb(1,2,x));
{}
> list(bb/3);
bb(1,1,1.1) ->  ;

{}
```

hidden_rule(x)

Masquer une règle ou un module pour la décompilation.
Rend 'non visible pour *rule*'. Si *x* est de la forme *i/a*, avec *i* identificateur et *a* entier, concerne le paquet de règles d'accès *i* et d'arité *a*. Si *x* est une chaîne, concerne toutes les règles du module de nom *x*. Par 'non visible pour *rule*', on entend non décompilable, c'est à dire qui ne peut être reconstruit sous forme de source Prolog. Par conséquent toutes les primitives de décompilation ou d'impression de ces règles n'auront aucun effet.

hidden_debug(x)

Masquer les accès à une règle ou tous les accès d'un module.
Rend 'non visible pour *debug*'. Si *x* est de la forme *i/a*, avec *i* identificateur et *a* entier, concerne le paquet de règles d'accès *i* et d'arité *a*. Si *x* est une chaîne, concerne toutes les règles du module de nom *x*. Par 'non visible pour *debug'*, on entend dont l'accès est caché. Le debugger ne montrera ni les appels à ce paquet de règles, ni les appels réalisés dans la queue des règles de ce paquet. Si dans une queue de règle non visible on appelle une règle visible, seule la queue de cette dernière sera visualisée. Les règles prédéfinies qui retournent des accès de règles (*dictionary* par exemple), ignoreront ces règles 'cachées pour *debug'*. On notera que pour décompiler ces règles, il faut utiliser des règles prédéfinies de décompilation nominatives (qui attendent l'identificateur d'accès et l'arité) et non globales (qui attendent un nom de module par exemple).

hidden(x)

Masquer une règle ou un module.
Si *x* est de la forme *i/a*, avec *i* identificateur et *a* entier, concerne le paquet de règles d'accès *i* et d'arité *a*. Si *x* est une chaîne, concerne toutes les règles du module de nom *x*. Rend les règles non visibles à la décompilation et non visibles au debugger ou aux primitives qui montrent les accès. Est équivalent à la suite de buts *hidden_rule(x) hidden_debug(x)*.

include(f)

Inclure la compilation d'un fichier.

Réalise la compilation du fichier *f* à l'endroit où se trouve la directive. Tout se passe donc comme si la directive était remplacée par le contenu du fichier *f*. La compilation du fichier *f* s'effectue suivant le même mode que celui piloté par le prédicat de compilation de niveau supérieur (*insert*, *reinsert*, *insertz*).
Ceci est une directive de compilation et non pas un prédicat, c'est à dire qu'elle peut figurer au milieu d'un source que l'on compile mais ne peut se trouver dans une queue de règle.

init_fassert(i/a, l)

Décrire l'indexation d'un paquet de faits non compilés.
Initialisation d'un paquet de faits non compilés d'identificateur d'accès *i* et d'arité *a*. L'argument *l* est une liste (terminée par *nil*) indiquant les combinaisons d'indexation choisies pour les arguments de ces faits. Cette liste est ordonnée par priorité de combinaison d'indexation.
Chaque élément de cette liste indique une combinaison et doit avoir pour valeur:
- soit une liste d'entiers (terminée ou non par *nil*) qui indique la combinaison des arguments à indexer, ou bien un entier si la combinaison se réduit à un seul élément. La table de hash-code correspondante à cette combinaison aura alors une taille par défaut (512 entrées).
- soit un doublet formé :
    - de la forme précédemment décrite en premier argument.
    - d'un entier indiquant la taille de la table de hash-code correspondante à cette combinaison en deuxième argument

Ce prédicat doit être appelé avant tout *fasserta* ou *fassertz*. Il pourra être appelé plusieurs fois pour le même paquet, à condition que l'argument *l* soit le même lors des différents appels.
Exemple:

```
> init_fassert(myrule/7, (1.2.3). (4.5.nil). 5. <6,200>.
    <2.6,300>. nil);
```

où l'on choisit:
- une indexation multiple en priorité sur la combinaison des arguments 1, 2 et 3.
- une sur les arguments 4 et 5 combinés.
- une sur le cinquième argument seul.
- une sur le sixième argument seul avec une taille de 200 pour la table de hash-code associée à cette combinaison.
- une sur la combinaison des arguments 2 et 6 avec une taille de 300 pour sa table de hash-code.

Le choix de la combinaison d'index lors de l'exécution dépend du type des arguments d'appel: priorité aux entiers, aux réels, aux identificateurs et aux chaînes de caractères. Si parmi les arguments correspondant à la première combinaison, l'un d'entre eux n'a pas le bon type, c'est la deuxième combinaison qui est examinée, et ainsi de suite.

NB: - La suppression complète d'un paquet de faits non compilés implique un nouvel appel à la primitive *init_fassert*. (après *suppress(i/a)* par exemple)

- La suppression de toutes les règles d'un paquet ne supprime pas la définition de l'indexation. *init_fassert* ne doit pas être refait. (après plusieurs *suppress(i/a,n)* par exemple)
- L'appel aux prédicats *edit* et *editm* sur des faits non compilés les transforme en règles compilées (équivalent à *suppress*/*insert*).

initialization(B)

Initialiser un groupe de règles.
Inclut le but *B* dans la liste des buts qui seront exécutés dès la fin de la compilation du fichier dans lequel se trouve cette directive. Ceci est une directive de compilation et non pas un prédicat, c'est à dire qu'elle peut figurer au milieu d'un source que l'on compile mais ne peut se trouver dans une queue de règle. Cette liste de buts sera exécutée à la condition qu'aucune erreur n'ait été détectée durant la compilation.

insert
insertz
reinsert

Compiler des règles.
Ces règles prédéfinies font basculer le système dans un mode dans lequel les énoncés (règles) lus sur l'unité d'entrée courante sont ajoutés, dans l'ordre dans lequel ils sont lus. Les directives et les déclarations sont respectivement exécutées et prises en compte immédiatement lorsqu'elles sont rencontrées. Le mode *insert* se termine soit quand un énoncé vide est trouvé (rencontre de ";;" en syntaxe Prolog II), soit si l'on rencontre la fin du fichier d'entrée.
Exemple:

```
>insert;
conc(nil, y, y) ->;
conc(e.x, y, e.z) -> conc(x, y, z); ;
{}
```

Si une erreur, de syntaxe est rencontrée, un avertissement est affiché, et un certain nombre de caractères (en principe tous les caractères jusqu'à un «;», ou jusqu'à la fin de la ligne sur l'unité *console*) sont ignorés, puis l'insertion reprend. A la fin de l'insertion *insert* génère alors l'erreur 86, et le complément d'erreur indique le nombre d'erreurs trouvées pendant l'insertion.
Les règles sont automatiquement indexées sur le premier argument au fur et à mesure de leur entrée.
Selon le mode de warning choisi au lancement de Prolog, à la lecture des règles un warning est affiché quand une variable singleton apparaît. Une variable singleton est une variable qui n'apparaît qu'une seule fois dans la règle, elle est donc à priori inutile. Pour la variable muette _ le warning n'est pas affiché.
insert sert à définir des paquets de règles, *reinsert* sert à redéfinir des paquets de règles et *insertz* sert à compléter des paquets de règles en ajoutant des alternatives en fin.

Par conséquent *insert* provoque une erreur lorsqu'un paquet lu existe déjà et qu'il n'a pas été déclaré non continu (directives *discontiguous/1* ou *multifile/1*), alors que *insertz* ajoute le paquet lu en fin de paquet existant et alors que *reinsert* remplace l'ancien paquet par la nouvelle définition. Attention ceci peut être dangereux : une erreur sur le nombre d'arguments d'une règle à l'intérieur d'un paquet, provoquera avec *reinsert* l'écrasement des règles précédentes quand on continuera à lire le paquet. Dans le même esprit, l'utilisation de *insertz* permettant de définir un paquet de règles en plusieurs morceaux (c'est à dire mixé avec d'autres règles), des erreurs sur le nombre d'arguments ou des confusions sur le nom de prédicats très ressemblants ne seront pas visibles.
Lorsqu'un module *m* est lu, le contexte défini par l'en-tête du module est mémorisé dans le module sous forme d'une règle *m:module_context(A1,A2,A3,A4)* où A1, A2, A3, A4 ont les valeurs définissant le contexte de lecture du module (cette règle est utilisée par les primitives d'édition de module telles que *editm*).

insert(f)
insertz(f)
reinsert(f)

Compiler des règles à partir d'un fichier.
Même fonctionnement que le mode *insert*, mais les énoncés sont lus sur le fichier indiqué. Selon que l'on a par ailleurs activé ou non le mode écho (cf. règle *echo*) les règles sont affichées sur la console au fur et à mesure de leur lecture.
Lorsqu'un module *m* est lu, le nom du fichier source du module est mémorisé dans le module sous forme d'une règle *m:module_file(s)* où *s* est une chaîne représentant le nom du fichier (cette règle est utilisée par les primitives d'édition de module telles que *editm*).

is_uncompiled(i/a)

Fait un succès si les règles de nom *i* et d'arité *a* forment un paquet de faits non compilés, échoue sinon.

list

Liste toutes les règles du module déterminé par le préfixe par défaut du contexte courant.

list(t)

Lister un paquet de règles.
Liste sur la sortie courante le ou les paquet de règles indiqués par *t*. *t* peut être une séquence de un ou plusieurs termes de la forme suivante (*i* désigne un identificateur, *a* un entier, et *v* une variable):

- *i/a* Toutes les règles composant la paquet de nom *i* et d'arité (nombre d'arguments) *a*.
- *i/v* Toutes les règles dont l'accès est *i*, quelque soit leur arité.
- *v/a* Toutes les règles dont l'arité est *a*.
- *i* équivalent de *i/v*.

exemple:

```
> list(num.bin/2);
```

list(s)

Lister un module.
s doit être une chaîne de caractères; toutes les règles ayant le préfixe *s* sont listées sur la sortie courante.

list(i/a,n)

Lister une règle.
i doit être un identificateur et *a* un entier; la règle numéro *n* du paquet correspondant au nom *i* et à l'arité (nombre d'arguments) *a* est listée sur la sortie courante.

multifile(i/a)

Permettre la non continuité d'un paquet de règles.
Directive de compilation identique à la directive *discontiguous/1*.

not_defined(s,l)

Trouver les accès non définis d'un module.
Unifie *l* avec la liste des accès appartenant au module *s* qui ne sont pas définis. Les accès sont notés *i/a* où *i* est l'identificateur d'accès et *a* son arité.
Remarque : cette règle prédéfinie active le récupérateur de mémoire sur le dictionnaire, avant de fournir le résultat.
Exemple:

```
> insert;
module("agence");
voyage(v,d,h,s) -> transport(v,t)
                   sejour(v,h,n)
                   duree(d)
                   multiplier(n,d,p)
                   additionner(p,t,s);
transport(Rome,1200) ->;
transport(Londres,800) ->;
transport(Tunis,2000) ->;
end_module("agence");
;
{}
> dictionary("agence",l);
{l=agence:module_context / 4.agence:transport / 2.
agence:voyage / 4.nil}
> not_defined("agence",l);
{l=agence:additionner / 3.agence:duree / 1. agence:multiplier
/ 3. agence:sejour / 3.nil}
```

predefined(t)

Tester si un *modèle* correspond à une règle prédéfinie.
S'efface avec succès si *t* est un terme correspondant à l'appel d'une règle prédéfinie.

rule(t, q)

Rechercher des règles correspondant à un *modèle* donné.
Cette primitive s'efface autant de fois qu'il existe de règles dont la tête s'unifie avec *t* et la queue avec *q*. Si une telle règle n'existe pas, alors *rule(t,q)* échoue. *t* doit être soit un identificateur, soit un n-uplet dont le premier argument est un identificateur connu.
La règle *rule* utilise l'indexation des règles lorsque c'est possible (voir *no_index*). Exemple d'utilisation de *rule* (se rapportant toujours au programme *conc* donné plus haut) :

```
>rule(conc(x,y,z),q);
{x=nil,z=y,q=nil}
{x=v149.v150,z=v149.v151,q=conc(v150,y,v151).nil}
>
```

rule(n, t, q)

Rechercher des règles correspondant à un *modèle* donné.
Même fonctionnement que *rule(t,q)* mais, en plus, *n* est unifié avec le rang de la règle dans son paquet.
Exemple :

```
>rule(n,conc(x,y,z),q);
{n=1,x=nil,z=y,q=nil}
{n=2,x=v172.v173,z=v172.v174,q=conc(v173,y,v174).nil}
>rule(n,conc(x,y,z),conc(x',y',z').l);
{n=2,x=v263.x',z=v263.z',y'=y,l=nil}
>rule(1,conc(x,y,z),q);
{x=nil,z=y,q=nil}
>
```

rule(n, a, t, q)

Rechercher des règles ayant un *nom* donné.
a doit être un identificateur, ou un tuple dont le 1er argument est un identificateur. Pour chacune des règles ayant *a* pour identificateur d'accès, cette règle s'efface après unification de *n* avec le rang de la règle dans son paquet, de *t* avec la tête de la règle et de *q* avec la queue de celle-ci. Ainsi, cette forme de *rule* peut être utilisée pour chercher des règles dont on connaît le nom mais pas le modèle.
Exemple :

```
conc(nil,x2,x2) ->;
conc(e.x1,x2,e.r) -> conc(x1,x2,r);
conc(x1,x2,x3,r) -> conc(x1,x2,u) conc(u,x3,r);;
> rule(n,conc,t,q);
{n=1,t=conc(v124,v125,v126,v127),
   q=conc(v124,v125,v128).conc(v128,v126,v127).nil}
{n=1,t=conc(nil,v125,v125),q=nil}
{n=2,t=conc(v127.v128,v125,v127.v129),
   q=conc(v128,v125,v129).nil}
```

rule_nb(i/a, n)

rule_nb(i, n)

Compter les règles d'un paquet.
Au moment de l'appel *i* doit être un identificateur et *a* un entier; *n* est alors unifié avec le nombre de règles composant le paquet dont le nom est *i* et l' arité (nombre d'arguments) est *a*. Dans la deuxième forme, seul le paquet ayant l'arité la plus faible est considéré.

retract(t, q)

Rechercher et supprimer des règles correspondant à un *modèle* donné.
Cette primitive s'efface autant de fois qu'il existe de règles dont la tête s'unifie avec *t* et la queue avec *q*. Utilise l'indexation des règles si c'est possible, sinon essaie les règles les unes après les autres dans l'ordre. Si une telle règle n'existe pas, alors *retract(t,q)* échoue. *t* doit être soit un identificateur, soit un n-uplet dont le premier argument est un identificateur connu.
Si une règle qui convient, est invisible à la décompilation, la règle est supprimée, et la primitive s'efface sans unifier les variables libres de *t* et *q*.

suppress(i/a)

Supprimer tout un paquet de règles.
i doit être un identificateur et *a* un entier; toutes les règles composant le paquet de nom *i* et d'arité (nombre d'arguments) *a* sont supprimées. S'il n'y a pas de règle, le prédicat s'efface en imprimant éventuellement[3] un warning.
Exemple:

```
data(1) ->;
data(2) ->;
data(3) ->;;
> data(x);
{x=1}
{x=2}
{x=3}
> suppress(data/1);
{}
> data(x);
WARNING : APPEL A UNE REGLE NON DEFINIE
>
```

suppress(i/a, n)

Supprimer une règle.
i doit être un identificateur et *a* et *n* deux entiers; la règle numéro *n* du paquet correspondant au nom *i* et à l'arité (nombre d'arguments) *a* est supprimée. S'il n'y a pas une telle règle, le prédicat s'efface en imprimant éventuellement[4] un warning.
Exemple:

```
data(1) ->;
```

[3]Cela dépend du niveau de warning choisi par une option au lancement de Prolog.

[4]Cela dépend du niveau de warning choisi par une option au lancement de Prolog.

```
data(2) ->;
data(3) ->;;
>data(x);
{x=1}
{x=2}
{x=3}
> suppress(data/1, 2);
{}
> data(x);
{x=1}
{x=3}
>
```

3.8. Manipulation des modules compilés

index(i/a)

Indexation d'un paquet de règles compilées.
Provoque l'indexation sur le 1er argument du paquet de règles compilées dont l'identificateur d'accès est *i* et l'arité *a*. Ne fait rien si la règle est déjà indexée. Cette règle peut être utilisée pour accélérer l'exécution des règles créées par *assert*, qui ne sont pas indexées automatiquement contrairement à celles créées par *insert*. N'a pas d'effet sur les faits non compilés.

kill_module(s)

Suppression de module.
s doit être une chaîne ou une liste de chaînes. Pour chaque module désigné dans *s*, toutes les règles du module sont supprimées, les tableaux du module sont désalloués, et les assignations des identificateurs du module sont défaites. Un message est éventuellement[5] affiché quand un module n'existe pas, et le traitement se poursuit.

load(f, l)
load(f)

Chargement de modules sauvés.
f est un nom de fichier (chaîne de caractères) et *l* est une liste de substitution de préfixes. Cette commande produit le chargement des modules sauvegardés dans le fichier indiqué; celui-ci doit avoir été produit par la commande *save*. Les règles d'initialisation de module, si elle existent, sont exécutées.
l est une liste de la forme $<pref_1 . subs_1> . \ldots . <pref_k . subs_k> . nil$ qui spécifie le renommage des modules chargés : $pref_1$ sera substitué par $subs_1$, $pref_2$ par $subs_2$, etc.… Si un module à renommer n'est pas présent, la substitution est ignorée pour ce module et un message peut[6] être affiché.
Exemple:

[5]Cela dépend du niveau de warning choisi par une option au lancement de Prolog.

[6]Cela dépend du niveau de warning choisi par une option au lancement de Prolog.

```
>load("myfile.mo", <"data","donnees">.nil);
{}
```

Une tentative de chargement d'un élément (règle ou tableau) déjà connu donne lieu à une erreur («définition multiple»).
La deuxième forme, *load(f)*, équivaut à *load(f, nil)*.

no_index(i/a)

Suppression de l'indexation d'un paquet de règles compilées.
Provoque la désindexation du paquet de règles compilées dont l'identificateur d'accès est *i* et l'arité *a*. Ne fait rien si la règle n'est pas indexée. On a intérêt à utiliser cette règle avant de manipuler dynamiquement des règles lues par *insert*, de manière à ce que les modifications soient toujours prises en compte de la même manière. N'a pas d'effet sur les faits non compilés.

reload(f)
reload(f, l)

Chargement de modules sauvés.
Même fonctionnement que *load*, sauf dans le cas de redéfinition d'un élément: la version rencontrée dans le fichier remplace celle qui se trouve en mémoire, sans produire d'erreur.

save(l, f)

Sauvegarde de modules.
f est un nom de fichier (chaîne de caractères) et *l* une liste de préfixes (chaînes de caractères). Cette commande produit la sauvegarde dans le fichier indiqué de tous les éléments (règles, variables statiques et tableaux) des modules correspondant aux préfixes donnés. Si un module n'existe pas, la sauvegarde se poursuit après affichage éventuel[7] d'un message.
Exemple:

```
> save(["","data","dict"], "myfile.mo");
{}
```

Le fichier produit est un fichier de *code objet*, ce code est exploitable uniquement par une machine Prolog. La mémorisation de programmes sous cette forme permet un rechargement plus rapide qu'à partir des fichiers sources.

save_state(s)

Sauvegarde d'un état de démarrage.
Sauvegarde tout le programme (règles, tableaux, identificateurs assignés, y compris le superviseur Prolog II+) dans un fichier de nom *s* (où *s* est une chaîne) qui peut être ensuite utilisé comme fichier de démarrage. Cette primitive a le même effet que *exit(s)* mais sans sortir de Prolog.

[7]Cela dépend du niveau de warning choisi par une option au lancement de Prolog.

4. Opérations prédéfinies sur les données

4.1. Les tests de type
4.2. Les opérations arithmétiques
4.3. Affectation
4.4. Opérations sur les chaînes
4.5. Composition et décomposition d'objets
4.6. Comparaison de termes quelconques

4.1. Les tests de type

Ces règles permettent de connaître le type d'un objet. Si l'argument est une variable libre, ces règles n'ont pas de sens et échouent.

ident(t)

Vérifie que *t* est un identificateur.

integer(t)

Vérifie que *t* est un entier.

real(t)

Vérifie que *t* est un réel.

double(t)

Identique à *real(t)*.

dot(t)

Vérifie que *t* est de la forme $t_1.t_2$ ou [*t1* | *t2*].

string(t)

Vérifie que *t* est une chaîne.

tuple(t)

Vérifie que *t* est un terme de la forme *f(*t_1 ... *tn)* ou <t_1 ... *tn*>.

Les primitives suivantes concernent l'«état» des variables et ont donc un esprit voisin de celui des tests de type :

bound(x)

bound(x) s'efface si *x* est lié. Une variable est considérée comme liée si elle a été unifiée contre :

- une constante (entier, réel, identificateur, chaîne),
- un terme de la forme $t_1.t_2$,
- un terme de la forme $<t_1, t_2, \ldots, t_n>$ ou *ff(t1, t2, ... , tn)*.

free(x)

S'efface uniquement si *x* n'est pas lié.

var_time(x,n)

Date de création d'une variable.

x doit être une variable libre. Cette règle prédéfinie unifie *n* avec la «date de création» de la variable *x* : il s'agit d'un nombre entier qui identifie de manière unique la variable; deux variables ont la même date si et seulement si elles sont liées entre elles (c'est ce même nombre qui est utilisé par le compilateur pour l'impression des variables libres -- voir l'exemple ci-dessous).
Exemple:

```
>eq(x,y) var_time(x,_tx) var_time(y,_ty) var_time(z,_tz)
   out(x.y.z) line;
v35.v35.v172
{y=x, _tx=35, _ty=35, _tz=172}
>
```

La date de création d'une variable correspond à une notion du «temps» qui est sensiblement la même que celle de l'horloge Prolog : le temps avance lors de l'effacement des buts et recule avec le *backtracking*. Dans l'exemple suivant, la variable x est créée trois fois; on constate qu'elle possède bien la même date de création:

```
>enum(i,3) var_time(x,t);
{i=1,t=35}
{i=2,t=35}
{i=3,t=35}
>
```

Attention, la valeur de la date de création des variables n'est pas invariante lors des tassages des piles par le *garbage collector*.

4.2. Les opérations arithmétiques

L'évaluation d'une expression est faite au moyen de la règle prédéfinie *val*. Le résultat d'une évaluation est soit un entier, soit un réel, soit un identificateur, soit une chaîne. Les booléens sont représentés par les entiers 0 et 1.

val(t_1, t_2)

tval(t1, t2)

Evaluent l'expression t_1 et unifient le résultat avec t_2. L'expression à évaluer est construite récursivement à partir des constantes, des identificateurs, des éléments de tableau et des fonctions évaluables. Les optimisations (si elles n'ont pas été désactivées au lancement de Prolog) du code généré pour l'appel au prédicat *val* ne permettent pas, pour la décompilation ou la mise au point, de restituer le terme d'origine mais un terme équivalent. De plus elles suspendent momentanément le mécanisme de gestion automatique des espaces et des piles. Le prédicat *tval* lui, n'est pas optimisé. On pourra le préférer à *val* dans les cas de programmes consommateurs d'espace où *val* manipule fréquemment des termes complexes. On évitera ainsi d'avoir à faire une gestion "manuelle" de la mémoire.
Exemples:

```
>val(add(mul(2, add(3, 4)), 1000), x);
{x=1014}
>val(add(mul(2e0, add(3, 4e0)), 1000), x);
{x=1.014000e+03}
>val(2 * (3 + 4) + 1000, x);
{x=1014}
```

Les fonctions arithmétiques peuvent avoir des arguments de types différents, dans ce cas il y a une conversion automatique des types en type le plus général (les types sont, par ordre décroissant: réel, entier). Quand une fonction à évaluer a un nombre incorrect d'arguments ou quand certains de ses arguments sont du mauvais type, *val* produit une erreur. Cette erreur est récupérable par la règle prédéfinie *block*.

• La valeur d'un nombre ou d'une chaîne est égale à ce nombre ou à cette chaîne.

• La valeur d'un tableau (externe ou interne) indicé est égale à la valeur de l'élément correspondant de ce tableau. Exemple:

```
>def_array(tab,100) assign(tab[50],3);
{}
>val(tab[50],_x);
{_x=3}
```

• La valeur d'un identificateur *i* est définie comme suit :

(1) si un terme *t* a été affecté à *i* (au moyen de la règle *assign* : voir le paragraphe suivant) alors la valeur de *i* est *t*.

(2) si *i* n'a pas fait l'objet d'une affectation préalable, alors la valeur de *i* est *i* lui-même.

Exemple :

```
>assign(un, 1);
{}
>val(un, x) val(deux, y);
{x=1,y=deux}
```

Certaines fonctions évaluables peuvent aussi être exprimées à l'aide d'opérateurs. Les fonctions évaluables sont :

add(t_1, t_2) ou t_1 + t_2

valeur(*add*(t_1, t_2)) = valeur(t_1) + valeur(t_2).
Les valeurs de t_1 et t_2 doivent être de type entier ou réel.

sub(t_1, t_2) ou t_1 - t_2

valeur(*sub*(t_1, t_2)) = valeur(t_1) - valeur(t_2).
Les valeurs de t_1 et t_2 doivent être de type entier ou réel.

mul(t_1, t_2) ou t_1 * t_2

valeur(*mul*(t_1, t_2)) = valeur(t_1) * valeur(t_2).
Les valeurs de t_1 et t_2 doivent être de type entier ou réel.

div(t_1, t_2) ou t_1 / t_2

valeur(*div*(t_1, t_2)) = valeur(t_1) / valeur(t_2).
Les valeurs de t_1 et t_2 doivent être de type entier ou réel.

mod(t_1, t_2) ou t_1 *mod* t_2

valeur(*mod*(t_1, t_2)) = valeur(t_1) modulo valeur(t_2).
mod (modulo) se différencie de *rem* (remainder) dans le cas où les opérandes *t1* et *t2* sont de signes contraires. En effet dans ce cas, le second opérande (*t2*) est ajouté au reste de la division entière de *t1* par *t2* (remainder).
Les valeurs de t_1 et t_2 doivent être de type entier.

rem(t_1, t_2) ou t_1 *rem* t_2

valeur(*rem*(t_1, t_2)) = reste de la division entière de *t1* par *t2*.
Les valeurs de t_1 et t_2 doivent être de type entier.

eql(t_1, t_2) ou t_1 =:= t_2 ou =:=(t_1, t_2)

valeur(*eql*(t_1, t_2)) = si valeur(t_1) = valeur(t_2) alors 1 sinon 0.
Si t_1 et t_2 sont de type arithmétique, la conversion automatique des types est appliquée avant l'évaluation.

'=\='(t_1, t_2) ou t_1 =\= t_2

valeur(=\=(t_1, t_2)) = si valeur(t_1) ≠ valeur(t_2) alors 1 sinon 0.
Si t_1 et t_2 sont de type arithmétique, la conversion automatique des types est appliquée avant l'évaluation.

inf(t_1, t_2) ou t_1 '<' t_2

valeur(*inf*(t_1, t_2)) = si valeur(t_1) < valeur (t_2) alors 1 sinon 0.
Pour les entiers et les réels, on prend la relation "<" entre les nombres. Pour les chaînes, on prend l'ordre alphabétique et pour les identificateurs, on prend l'ordre alphabétique sur les chaînes associées.

infe(t_1, t_2) ou t_1 '=<' t_2

valeur(*infe*(t_1, t_2)) = si valeur(t_1) ≤ valeur (t_2) alors 1 sinon 0. Cf. *inf*.

sup(t_1, t_2) ou t_1 '>' t_2

valeur(*sup*(t_1, t_2)) = si valeur(t_1) > valeur (t_2) alors 1 sinon 0. Cf. *inf*.

supe(t_1, t_2) ou t_1 '>=' t_2

valeur(*supe*(t_1, t_2)) = si valeur(t_1) ≥ valeur (t_2) alors 1 sinon 0. Cf. *inf*.

if(t, t_1, t_2)

valeur(*if*(t, t_1, t_2)) = si (valeur(t)≠0) alors valeur(t_1) sinon valeur(t_2).

sign(t)

valeur (*sign*(*t*))= si valeur(*t*) = 0 alors 0, si valeur(*t*) < 0 alors -1, sinon 1.
La valeur de *t* doit être de type entier ou réel.

ceiling(t)

valeur (*ceiling*(*t*))= - (partie entière(-valeur(*t*)))
La valeur de *t* doit être de type entier ou réel. Le résultat est de type entier.

floor(t)

valeur (*floor*(*t*)) = partie entière(valeur(*t*)) .
La valeur de *t* doit être de type entier ou réel. Le résultat est de type entier.

round(t)

valeur*(round(t))*= partie entière(valeur(*t*)+0.5).
La valeur de *t* doit être de type entier ou réel. Le résultat est de type entier.

trunc(t)

valeur(*trunc*(*t*)) = conversion en entier de la valeur de *t*.
La valeur de *t* doit être de type entier ou réel.

float(t)

valeur(*float*(*t*)) = conversion en réel de la valeur de *t*.
La valeur de *t* doit être de type entier ou réel.

double(t)

valeur(*double*(*t*)) = conversion en réel de la valeur de *t*.
La valeur de *t* doit être de type entier ou réel.

abs(t)

valeur(*abs*(*t*)) = valeur absolue de la valeur de *t*.
La valeur de *t* doit être de type entier ou réel.

Les fonctions suivantes doivent être appliquées à des arguments de type entier. Elles donnent un résultat de type entier.

'/\'(t1,t2)

valeur(*'/\'*(*t1*,*t2*)) = 'et' bit à bit entre *t1* et *t2*.

'\/'(t1,t2)

valeur(*'\/'*(*t1*,*t2*)) = 'ou' bit à bit entre *t1* et *t2*.

'<<'(t1,t2)

valeur(*'<<'*(*t1*,*t2*)) = *t1* décalé de *t2* bits vers la gauche.

'>>'(t1,t2)

valeur(*'>>'*(*t1*,*t2*)) = *t1* décalé de *t2* bits vers la droite.

'~'(t)

valeur('~'(*t*)) = complément bit à bit de *t*.

Les fonctions suivantes doivent être appliquées à des entiers ou des réels et donnent un résultat de type réel. Les fonctions trigonométriques travaillent avec des angles exprimés en radians.

atan(t)

valeur (*atan*(*t*)) = *arc tangente*(valeur(*t*)).

cos(t)

valeur (*cos*(*t*)) = *cosinus*(valeur(*t*)).

exp(t)

valeur (*exp*(*t*)) = *exponentielle*(valeur(*t*)).

ln(t)

valeur(*ln*(*t*)) = *logarithme népérien*(valeur(*t*)).

rad(t)

valeur(*rad*(*t*)) = *conversion en radian*(valeur(*t*)).

sin(t)

valeur(*sin*(*t*)) = *sinus*(valeur(*t*)).

sqrt(t)

valeur(*sqrt*(*t*)) = *racine carrée*(valeur(*t*)).

tan(t)

valeur(*tan*(*t*)) = *tangente*(valeur(*t*)).

t_1 ** t_2

valeur(t_1 ** t_2) = valeur(t_1) à la puissance valeur(t_2).

4.3. Affectation

assign(i, t)

tassign(i, t)

cassign(i, t)

Affectent le terme *t* à l'identificateur *i*, *t* peut être un terme Prolog quelconque. Tout se passe comme si *i* devenait le nom d'une variable «globale» (ultérieurement accessible pendant l'effacement de n'importe quel but) et «statique» (résistante au *backtracking*) possédant *t* pour valeur. Il s'agit donc bien de l'affectation classique, comme elle se pratique dans FORTRAN, Pascal, etc…. Les optimisations (si elles n'ont pas été désactivées au lancement de Prolog) du code généré pour l'appel au prédicat *assign* suspendent momentanément le mécanisme de gestion automatique des espaces et des piles. Le prédicat *tassign* lui, n'est pas optimisé. On pourra le préférer à *assign* dans les cas de programmes consommateurs d'espace où *assign* manipule fréquemment des termes complexes. On évitera ainsi d'avoir à faire une gestion "manuelle" de la mémoire. Le prédicat *cassign* est similaire au prédicat *tassign* mais permet de conserver les contraintes (*dif*, *freeze*) et les attributs (variables attribuées) associés aux variables du terme *t*.

Exemple :

```
>assign(nom_fichier, "monfichier.txt");
{}
>val(nom_fichier,x)
{x="monfichier.txt"}
>
```

En Prolog, ces variables statiques peuvent être vues comme une manière particulièrement efficace de réaliser des *assertions* (ou règles sans queue). Du point de vue de l'utilisateur, on peut considérer que l'emploi de *assign* et *val* fait ci-dessus est conceptuellement équivalent à:

```
>retract(nom_fichier(x), nil);
>assert(nom_fichier("monfichier.txt"), nil);
{}
>nom_fichier(x);
{x="monfichier.txt"}
>
```

Exemple : possibilité d'affecter un terme quelconque, et non uniquement des constantes.

```
>assign(myterm, jean.<34,"rue blanche">.A_utres_infos);
{}
>val(myterm,x) val(myterm,y);
{x=jean.<34,"rue blanche">.v64,y=jean.<34,"rue
blanche">.v65}
>
```

assign(tab(i), t) ou *assign(tab[i], t)*

tassign(tab(i), t) ou *tassign(tab[i], t)*

Affectent le terme *t* à l'élément de rang *i* du tableau *tab*. Ce tableau doit avoir été défini au préalable avec *def_array*.

Pour l'accès à des éléments de tableaux (dans *val*) les notations *tab(i)* et *tab[i]* sont équivalentes; cependant, elles ne sont pas représentées par les mêmes termes Prolog, et dans le cas de l'utilisation de *tab[i]* le compilateur optimise l'accès au tableau. Pratiquement, on réservera la notation *tab[i]* pour la désignation d'éléments de tableau uniquement. Nous ferons également la même remarque que précédemment concernant les optimisations : *assign* est optimisé, *tassign* ne l'est pas.

backtrack_term

Tableau prédéfini du module *sys*, de longueur 100 (indices possibles entre 1 et 100) qui peut être manipulé à l'aide des prédicats prédéfinis *val* ou *tval* et *assign* ou *tassign*. Les valeurs qui lui sont assignées sont perdues au backtracking. Chaque élément du tableau est initialisé à 0. Exemple:

```
> insert;
test(i) ->  assign(backtrack_term[1],111)
            val(backtrack_term[1],i);
test(i) ->  val(backtrack_term[1],i);
;
{}
> test(i);
{i=111}
{i=0}
```

def_array(i, n)

Définit dynamiquement un tableau de termes Prolog de nom *i* et de taille *n*. Ce tableau se comportera comme une variable «globale» (ultérieurement accessible pendant l'effacement de n'importe quel but) et «statique» (résistante au *backtracking*). Chaque élément du tableau est initialisé avec l'entier 0. Les valeurs légales de l'indice sont incluses dans 1..*n*.

Si un tableau de même nom existe déjà :
- s'il s'agit d'un tableau de même taille, il ne se passe rien
- si les tailles diffèrent, il se produit une erreur

L'accès et l'affectation sont analogues à ceux des tableaux des autres langages de programmation : voir ci-dessus *val* pour l'accès et *assign* pour l'affectation. Le tableau est désalloué lorsqu'on tue le module auquel il appartient (c.à.d. les règles ayant le même préfixe) ou bien lorsqu'on exécute le prédicat *kill_array*.

```
"gestion d'une pile"

inc(i) -> val(i + 1, x) assign(i, x);
dec(i) -> val(i - 1, x) assign(i, x);

initialise -> assign(pointeur,0) def_array(pile,100);

empile(v) ->
   inc(pointeur)
   val(inf(pointeur,100),1)
   !
   val(pointeur,p)
   assign(pile[p], v);
empile(v) -> outml("débordement de la pile") fail;
```

```
   depile(v) ->
      val(eql(pointeur,0),0)
      !
      val(pile[pointeur],v)
      dec(pointeur);
   depile(v) -> outm("pile vide") line fail;;
>initialise;
{}
>empile(12345);
{}
>empile(23456);
{}
>depile(x) depile(y);
{x=23456, y=12345}
>depile(x);
```

redef_array(i, n)

Analogue à *def_array* mais, si le tableau de nom *i* existe déjà, redéfinit sa taille: *n*, conserve les affectations existantes pour les indices valides, et initialise s'il y a lieu les nouveaux éléments.

is_array(i,t)

Vérifie que l'identificateur *i* désigne bien un tableau et unifie *t* avec sa taille.

kill_array(i)

Détruit le tableau de nom *i*. Si le tableau n'est pas défini, le prédicat s'efface en imprimant éventuellement[1] un warning.

4.4. Opérations sur les chaînes

char_code(c, n)

Fait correspondre au caractère *c* son code interne *n* et vice-versa. Ce code peut être un code ISO[2] ou le code de la machine hôte. Si *c* est un identificateur dont la représentation (hormis les quotes) est limitée à un caractère, le code correspondant est unifié avec *n*.
Exemple :

```
>char_code("A",n) char_code('A',n);
{n=65}
>char_code(c,65);
{c="A"}
```

[1]Cela dépend du niveau de warning choisi par une option au lancement de Prolog.

[2]Il s'agit du code ISO 8859-1 (voir la table complète en annexe F). Ce jeu est le jeu ASCII américain étendu sur 8 bits avec des caractères accentués.

conc_string($s1$*,* $s2$*,* $s3$*)*

Enumère tous les triplets $s1$, $s2$, $s3$ de chaînes telles que $s3$ est la concaténation de $s1$ et $s2$. $s1$, $s2$, $s3$ doivent être suffisamment connus pour produire des ensembles finis de triplets, c'est à dire soit le troisième argument connu, soit deux des trois chaînes connues.
Exemple :

```
>conc_string("ab", "cd", s);
{s="abcd"}
>conc_string(s1, s2, "abcd");
{s1="",s2="abcd"}
{s1="a",s2="bcd"}
{s1="ab",s2="cd"}
{s1="abc",s2="d"}
{s1="abcd",s2=""}
```

substring($s1$*,* *i, j,* $s2$*)*

Extrait de la chaîne $s1$ une sous-chaîne commençant à la position *i*, de longueur *j* et essaie de l'unifier avec $s2$.
Exemple :

```
>substring("1234567890", 3, 4, x);
{x="3456"}
```

find_pattern($s1$*,* $s2$*, n)*

Unifie *n* avec la position du début de la chaîne $s2$ dans la chaîne $s1$. Si $s2$ n'est pas trouvé, alors l'effacement de *find_pattern* échoue.
Exemple :

```
>find_pattern("1234567890", "3456", p);
{p=3}
>find_pattern("1234567890", "abcd", p);
>
```

Note:

Cette primitive est définie dans le module externe *prouser.c*, la modification inconsidérée du source correspondant peut être responsable de son malfonctionnement. Dans le source livré, la longueur des chaînes est limitée à 256 caractères.

4.5. Composition et décomposition d'objets

arg2(n, $t1$*,* $t2$*)*

Unifie *t2* avec la longueur ou l'élément de rang *n* d'un n-uplet, d'une liste ou d'une chaîne.

(1) Si $t1$ est une chaîne alors si
n=0, $t2$ = longueur($t1$)
$n\neq0$, $t2$ = $n^{\text{ième}}$ caractère de la chaîne $t1$.

(2) Si t_1 est une liste alors si
n=0, t_2 = nombre d'éléments de la liste
$n \neq 0$, t_2 = $n^{\text{ième}}$ élément de la liste t_1.

(3) Si t_1 est un n-uplet alors si
n=0, t_2 = nombre d'arguments du n-uplet
$n \neq 0$, t_2 = $n^{\text{ième}}$ argument du n-uplet t_1.

Exemples :

```
>arg2(0,"abcdef",x);
{x=6}

>arg2(3,"abcdef",x);
{x="c"}

>arg2(0,aa.bb.cc.dd.ee.ff.nil,x);
{x=6}

>arg2(3,aa.bb.cc.dd.ee.ff.nil,x);
{x=cc}

>arg2(0,<aa,bb,cc,dd,ee,ff>,x);
{x=6}

>arg2(3,<aa,bb,cc,dd,ee,ff>,x);
{x=cc}
```

arg(n, t_1, t_2)

La primitive correspondante de l'interpréteur Prolog II a été renommée *arg2* dans le compilateur. *arg/3* est une règle prédéfinie du mode Edinburgh.

conc_list_string(l, s)

Concatène les chaînes de la liste *l* pour créer la chaîne *s*.
Exemple :

```
>conc_list_string("How "."are "."you "."?".nil, x);
{x="How are you ?"}
```

copy_term(t1, t2)

Fait une copie du terme *t1* en remplaçant chaque variable libre (contrainte ou pas) par une nouvelle variable sans contrainte et unifie cette copie avec *t2*. C'est à dire que les deux termes sont représentés par le même arbre, aux noms de variables près. *t1* est un terme prolog quelconque, voire infini.
Exemple :

```
>copy_term(one(2.x,y), t);
{t=one(2.v33,v31)}
```

copy_term_with_constraints(t1, t2)

Analogue au prédicat *copy_term/2* mais copie les contraintes sur les variables.

enum(i, k_1, k_2)

enum(i, k_2)

k_1 et k_2 doivent être des entiers connus. Si $k_1 > k_2$ alors on a un échec. Par définition, cette règle s'efface de k_2-k_1+*1* manières : la première consiste à unifier *i* et k_1; la deuxième consiste à unifier *i* et k_1+*1*, etc…; la dernière consiste à unifier *i* et k_2.

Exemple :

```
>enum(i,5,9);
{i=5}
{i=6}
{i=7}
{i=8}
{i=9}
```

La deuxième forme, *enum(i,k),* équivaut à *enum(i,1,k)*.

gensymbol(i)

Crée un nouvel identificateur de la forme *idn*, où *n* est un entier, et l'unifie avec *i*.

list_string(l, s)

Compose une chaîne *s* à partir d'une liste *l* de caractères.

Exemple :

```
>list_string("H"."e"."l"."l"."o".nil, x);
{x="Hello"}
```

list_tuple(l, t)

Compose un n-uplet *t* à partir de la liste d'arguments *l*.

Exemple :

```
>list_tuple(111.aaa.222."Hello".nil,x);
{x=<111,aaa,222,"Hello">}
```

member(x,l)

Unifie successivement *x* avec chaque élément de la liste *l*. Est décrit par les règles :

```
member(x, x.l) ->;
member(x, _.l) -> member(x, l);
```

setarg(n, t_1, t_2)

Remplace l'élément de rang *n* du terme *t1* par le terme *t2*. *t1* doit être une liste ou bien un n-uplet. Le terme initial est restauré sur un backtracking.

Exemple:

```
> reinsert;
newarg(55) ->;
newarg(66) ->;;
{}
> eq(x, 1.2.3.nil) newarg(y) arg2(2,x,e1) setarg(2,x,y)
arg2(2,x,e2);
{x=1.55.3.nil, y=55, e1=2, e2=55}
{x=1.66.3.nil, y=66, e1=2, e2=66}
```

split(t, l)

Décompose une chaîne ou un n-uplet en la liste de ses composants. Au moment de l'effacement de *split(t, l)*, *t* doit être connu; on a alors :

- Si *t* est une chaîne, *l* est unifié avec la liste de ses caractères.
- Si *t* est un n-uplet, *l* est unifié avec la liste des arguments de *t*.
- Sinon échec.

Exemple :

```
>split("abcde", x);
{x="a"."b"."c"."d"."e".nil}
>split(plusgrand(Jean, Pierre.Marie), x));
{x=plusgrand.Jean.(Pierre.Marie).nil}
```

string_ident(s_1, s_2, i)

string_ident(s, i)

Etablit la correspondance entre un identificateur et sa représentation sous forme de chaîne. Si l'identificateur n'existe pas, il est créé. Dans la forme à trois arguments, fait correspondre aux chaînes s_1 et s_2 l'identificateur dont le préfixe est s_1 et la représentation abrégée s_2, et vice-versa. Dans la forme à deux arguments, utilise le contexte courant de lecture/écriture pour établir la relation entre l'identificateur *i* et sa représentation *s*. *s* peut être une représentation complète ou une représentation abrégée.

Exemple :

```
>string_ident(p, i, invites:Jean);
{p="invites",i="Jean"}
>string_ident("invites","Jean",x);
{x=invites:Jean}
```

Exemple dans le contexte standard :

```
>string_ident(s,sys:outm);
{s="outm"}
```

string_integer(s, n)

Fait correspondre la chaîne *s* à l'entier *n* et vice-versa.

```
>string_integer(s,123);
{s="123"}
>string_integer("-0123",n);
{n=-123}
```

string_real(s, r)

Fait correspondre la chaîne *s* au réel *r* et vice-versa.

```
>string_real(s,12.34e5) string_real(s,x);
{s="1.234000000000000e+06",x=1.234000000000000e+06}
>string_real(s,1d110) string_real(s,x);
{s="1.000000000000000e+110",x=1.000000000000000e+110}
```

string_double(s, d)

Identique à *string_real(s,d)*.

string_term(s, t, s',n)

string_term(s, t, s')

string_term(s, t)

Essaie de lire dans la chaîne de caractères *s* un terme Prolog qui sera unifié avec *t*. *s'* est la partie de la chaîne *s* qui n'a pas pu être utilisée. Si une variable apparaît dans *s*, une variable libre Prolog est créée.
Inversement (si *s* n'est pas connue) écrit, à la façon de *out(t)*, le terme *t* dans une chaîne de taille maximale de *n* caractères (400<*n* < 32768) et l'unifie avec *s*, *s'* n'est pas utilisée dans ce cas. *t* est un terme tout à fait quelconque, pouvant même contenir des variables et des structures complexes. Si l'écriture de *t* ne tient pas sur *n* caractères une erreur est générée. Si *n* n'est pas spécifié, 400 est choisi.

```
>string_term(s,myterm(x,1.2e0),s') string_term(s,x,s');
{s="myterm(v14,1.200000000000000e+00)",x=myterm(v15,1.2000
00000000000e+00),
s'=""}
>string_term("1.2,3)",t,s') string_term(s,t,s');
{t=1.2,s'=",3)",s="1.2"}
```

term_vars(t, l)

Unifie *l* avec la liste des variables libres, éventuellement contraintes, de *t*. *t* est un terme prolog quelconque, voire infini.
Exemple :

```
>term_vars(one(2.x,y,y), t);
{t=x.y.nil}
```

4.6. Comparaison de termes quelconques

Il est possible de comparer formellement deux termes Prolog quelconques. Pour cela, l'ordre croissant suivant sur les termes a été défini:

- les variables
- les réels
- les entiers
- les identificateurs
- les chaînes de caractères
- les n-uplets de longueur < 2.
- les listes
- les n-uplets de longueur >= 2.

La comparaison de deux variables est fonction de leur date de création. La comparaison de deux entiers ou deux réels suit les règles habituelles. La comparaison de deux chaînes de caractères ou deux atomes suit l'ordre alphanumérique. La comparaison de deux listes est celle de la comparaison des deux premiers éléments trouvés différents. La comparaison de deux n-uplets s'effectue d'abord sur leur longueur, et à longueur égale sur la comparaison des deux premiers éléments trouvés différents. Le prédicat de comparaison est le suivant:

term_cmp(X, Y, V)

Compare les termes *X* et *Y*, unifie *V* avec l'entier -1 si *X* précède *Y*, 1 si *Y* précède *X*, 0 si les termes *X* et *Y* sont formellement égaux. Quelques exemples:

```
> term_cmp(1e0,1,V);
{V=-1}
> term_cmp(foo,zebra,V);
{V=-1}
> term_cmp(short,short,V);
{V=0}
> term_cmp(shorter,short,V);
{V=1}
> term_cmp(foo(a),foo(b),V);
{V=-1}
> term_cmp(foo(aa),foo(bb),V);
{V=-1}
> term_cmp(X,X,V);
{V=0}
> term_cmp(Y,X,V);
{V=-1}
> term_cmp(_,_,V);
{V=-1}
> term_cmp("foo",foo,V);
{V=1}
> term_cmp(1.2.3.nil,1.1.3.4.nil,V);
{V=1}
> term_cmp(1.2.nil,<1,2>,V);
{V=-1}
> term_cmp(1.2.nil,<X>,V);
{V=1}
>
```

term_cmpv(X, Y, V)

Compare les termes *X* et *Y* à la manière de *term_cmp*, mais sans prendre en compte les variables apparaissant dans les termes *X* et *Y*. Pour le prédicat *term_cmpv*, deux variables distinctes sont considérées comme identiques. Exemple:

```
> term_cmpv(foo(V1),foo(V2),V);
{V=0}
```

Deux prédicats de tri d'une liste sont fournis : *sort/2*, *keysort/2*. Pour effectuer ce tri, ils utilisent de manière interne le prédicat *term_cmp/3* décrit ci-dessus.

sort(L1, L2)

L2 est unifiée avec la liste des éléments de *L1* triés dans l'ordre croissant. Dans cette liste, les éléments multiples ne sont conservés qu'une seule fois. Exemple:

```
> sort(4.7.foo.3.4.1.nil,L);
{L=1.3.4.7.foo.nil}
```

keysort(L1, L2)

L2 est unifiée avec la liste des éléments de *L1* triés dans l'ordre croissant. Dans cette liste, les éléments multiples sont conservés. Exemple:

```
> keysort(4.7.foo.3.4.1.nil,L);
{L=1.3.4.4.7.foo.nil}
```

Ces mêmes prédicats admettent un argument supplémentaire qui désignera le prédicat utilisateur servant pour la comparaison des termes. Ce prédicat utilisateur devra être d'arité égale à 3. Il recevra les termes à comparer dans les 2 premiers et devra rendre dans le troisième argument le résultat de la comparaison (-1, 0 ou 1).

sort(L1, L2, C_ompar)

L2 est unifiée avec la liste des éléments de *L1* triés dans l'ordre spécifié par le prédicat *C_ompar*. Dans cette liste, les éléments multiples ne sont conservés qu'une seule fois. Le prédicat *sort/2* décrit précédemment n'est qu'un cas particulier de celui-ci, en donnant à l'argument *C_ompar* la valeur *term_cmp*. Exemple:

```
> insert;
compar_tuple(P1(X), P2(Y), R) -> term_cmp(X,Y,R);
;
> sort(aa(4).bb(7).cc(foo).dd(3).aa(4).ff(1).nil,L,
                                    compar_tuple);
{L=ff(1).dd(3).aa(4).bb(7).cc(foo).nil}
```

keysort(L1, L2, C_ompar)

L2 est unifiée avec la liste des éléments de *L1* triés dans l'ordre spécifié par le prédicat *C_ompar*. Dans cette liste, les éléments multiples sont conservés. Le prédicat *keysort/2* décrit précédemment n'est qu'un cas particulier de celui-ci, en donnant à l'argument *C_ompar* la valeur *term_cmp*. Exemple:

```
> insert;
compar_tuple(P1(X), P2(Y), R) -> term_cmp(X,Y,R);
;
> keysort(aa(4).bb(7).cc(foo).dd(3).aa(4).ff(1).nil,L,
                                    compar_tuple);
{L=ff(1).dd(3).aa(4).aa(4).bb(7).cc(foo).nil}
```

5. Les entrées / sorties

5.0. Généralités
5.1. Entrées
5.2. Sorties
5.3. Chargement et adaptation du module de dessin
5.4. Déclaration d'opérateurs

Tous les organes d'entrée / sortie (clavier, écran, fichiers, fenêtres, etc…) utilisés par un programme Prolog II+ y sont représentés par des entités appelées *unités d'entrée/sortie*. La représentation écrite d'une telle unité est une chaîne de caractères[1]; le cas échéant, cette chaîne représente aussi une information identifiant l'unité vis-à-vis du système d'exploitation : nom de fichier, etc…

A tout moment, le système Prolog II+ «connaît» un certain nombre d'unités d'entrée / sortie : ce sont les unités qui ont été précédemment ouvertes et qui n'ont pas encore été fermées. Les descripteurs de ces unités sont rangés dans une sorte de pile : l'unité qui est au sommet s'appelle l'unité courante. En principe, toutes les opérations d'entrée / sortie s'effectuent sur l'unité courante correspondante. Au lancement, une unité d'entrée (appelée *console*) et une unité de sortie (appelée également *console*) sont ouvertes automatiquement par le système et sont les unités courantes; habituellement, ces unités sont associées au clavier et à l'écran de l'ordinateur ou du terminal depuis lequel Prolog a été lancé.

Changer d'unité (i.e.: effacer *input(f)* ou *output(f)*) ne ferme pas «physiquement» l'unité courante ni ne change son état, mais ouvre -si nécessaire- une nouvelle unité et la met au sommet de la pile, de façon à pouvoir restaurer l'unité précédente, à la fermeture de cette nouvelle unité. Le nombre d'unités pouvant être ouvertes simultanément est dépendant de la machine hôte.

5.0. Généralités

Les unités d'entrée / sortie créées ont un type bien défini, ainsi qu'un mode d'ouverture. Les différents types possibles sont :

- *:console* pour l'unité créée automatiquement par Prolog au démarrage.
- *:text* pour un fichier disque au format texte.
- *:binary* pour un fichier disque au format binaire.
- *:memory_file* pour un fichier mémoire décrit ci après.
- *:unix_pipe* pour un "tube" de communication inter-processus (UNIX).
- *:graphic_area, :tty_area, :edit_area* pour des unités graphiques.

---

1 Les unités fenêtres peuvent aussi être représentées par des entiers.

Les différents modes d'ouverture possibles sont :

:read pour la lecture.

:write pour l'écriture. Si l'objet physique (un fichier disque par exemple) du même nom existe déjà, son contenu sera effacé et remplacé par les sorties qui seront demandées.

:append pour l'écriture. Ce mode est spécifique aux fichiers. Si un objet physique associé existe déjà, il est ouvert et les futures écritures y seront ajoutées en fin. Ce mode d'ouverture peut être défini par une convention de nommage de l'unité, dont le nom doit dans ce cas se terminer par "+". Le fichier associé à l'unité est alors celui de nom donné, après avoir ôté le "+". Par exemple *output("file+")* va provoquer l'ouverture du fichier de nom *"file"* en mode *append.*

Certaines unités peuvent être ouvertes à la fois en lecture et en écriture. C'est le cas des types *:console*, *:tty_area* , *:edit_area*, et *:memory_file*.

memory_file(s)

memory_file(s,n)

Crée une unité de type "fichier mémoire" de nom *s*. Son tampon de caractères a une longueur de 400 caractères pour la première forme du prédicat. En utilisant la deuxième forme, le tampon sera de longueur *n* ou de la valeur la plus proche, sachant que la taille minimale allouée est de 400 caractères et la taille maximale allouée est de 32K caractères. Cette unité peut être utilisée en entrée et en sortie. Elle peut être en même temps unité courante d'entrée et unité courante de sortie. Si le tampon est trop petit pour les opérations d'écritures qui y sont faites, il est automatiquement doublé de taille, sans dépasser les 32K caractères. L'unité sera fermée physiquement quand elle aura été fermée à la fois en lecture et en écriture. Cette unité peut être utilisée au même titre que les fichiers.

Les types et les modes d'ouverture des unités peuvent être obtenus par le prédicat :

current_file(x)

current_file(x,t,m)

Unifie successivement *x* avec les noms des unités ouvertes en lecture et/ou en écriture, *t* avec le type de cette unité, et *m* avec le mode d'ouverture.

5.1. Entrées

Le tampon de caractères standard pour une unité d'entrée fourni par Prolog est limité à 400 caractères. S'il arrive qu'une ligne soit plus longue, Prolog se comporte comme s'il y avait un espace après le 400-ième caractère. Le reste de la ligne est lu après épuisement des 400 caractères précédents.

Il est conseillé de manipuler des lignes moins longues. Se rappeler que :

- le Retour Chariot sera lu au même titre que le caractère blanc par les primitives qui commencent par lire tous les espaces,

- les espaces et les tabulations peuvent apparaître partout entre les unités lexicales,
- les chaînes, les identificateurs, les entiers et la mantisse des réels peuvent s'écrire sur plusieurs lignes, en masquant le Retour Chariot par le caractère '\', si l'option d'interprétation du '\' est active (cf. U2.3.).

5.1.1. Règles pour l'entrée.

eol

Test de fin de ligne en entrée.
Teste si une fin de ligne a été atteinte; plus précisément : si le prochain caractère qui sera lu est une marque de fin-de-ligne, alors *eol* s'efface; sinon, l'effacement de *eol* échoue.
Exemple :

```
> in_char'(c) eol;
a<CR>
{c="a"}
> in_char'(c) eol;
a<space><CR>
>
```

eof

Test de fin d'entrée.
Teste s'il existe encore des caractères en entrée sur l'unité courante. Autrement dit, teste si la prochaine lecture va provoquer l'erreur 104: FIN DE FICHIER. Si ce n'est pas le cas, l'effacement de *eof* échoue. Ne peut évidemment jamais s'effacer sur la "console" ou dans une fenêtre TTY, puisque ce sont des unités en mode terminal, qui ont toujours des caractères, éventuellement après saisie.
Exemple :

```
> input("fichier_vide") eof input("console");
{}
>
```

in_char(t)

Lecture d'un caractère.
Lit le prochain caractère, le transforme en une chaîne de longueur 1 et essaie de l'unifier avec *t*.
Remarque: le caractère fin de ligne, s'il est assimilé à un espace pour d'autres primitives qui analysent les caractères lus, vaudra bien pour cette primitive "\n".
Remarque: toute commande (suite de buts) se termine au ";". Lorsque *in_char* est lancé sur la ligne de commande, le premier caractère lu est le caractère qui suit immédiatement le ';'. Il s'agit souvent de : fin de ligne.
Exemple :

```
> in_char(c);
{c="\n"}
> clear_input in_char(c);
a<CR>
{c="a"}
```

in_char'(t)

Lecture d'un caractère non blanc.
Lit tous les caractères blancs et se comporte comme *in_char*(*t*).
Exemple:

```
>in_char'(x) in_char(y) in_char'(z) in_char(t);
      ab   c   d
{x="a",y="b",z="c",t=" "}
```

next_char(t)

Essaie d'unifier le prochain caractère à l'entrée avec *t* sans le lire vraiment.

next_char'(t)

Lit tous les caractères blancs s'il y en a et ensuite se comporte comme *next_char(t)*.
Exemple:

```
>next_char'(x) in_char(y);
   abc
{x="a",y="a"}
```

inl(s)

inl(s, n)

Lecture d'une ligne.
Lit toute la fin de la ligne courante et l'unifie, en tant que chaîne de caractères, avec s. *n* est unifié avec le rang de la ligne lue, dans son unité. *n* est beaucoup plus significatif pour les fichiers, voire les fenêtres EDIT. Dans la "console" ou les fenêtres TTY, les lignes sont divisées en deux groupes : lignes d'entrée et lignes de sortie. *n* est alors le rang à l'intérieur d'un de ces groupes.
Exemple:

```
> clear_input inl(s);
Maitre corbeau, sur un arbre perche
{s="Maitre corbeau, sur un arbre perche"}
```

in(t, c)

in(t, D, c)

Lecture d'un terme Prolog.
Lit la plus grande suite *x* de caractères qui constituent un terme ou le début d'un terme. Si *x* est l'expression d'un terme, on essaie d'unifier ce dernier avec *t*. Si *x* n'est pas un terme, une erreur se produit. Tous les caractères blancs qui suivent *x* sont lus; *c* est unifié avec la première unité lexicale lue, ne faisant pas partie de *x*. *D* est unifié avec le *dictionnaire des variables* du terme, c'est-à-dire une liste de doublets correspondant aux variables de *t*. Chaque doublet contient la chaîne de caractères qui est le nom originel de la variable et la variable en question.
Exemple:

```
>in(t, c);
ceci.est.un.terme;
{t=ceci.est.un.terme, c=";"}

>in(t, D, c);
```

```
aa(x, bb(y_n1',x));
{t=aa(v129, bb(v163,v129)),
   D=<"x", v129>.<"y_n1'",v163>.nil, c=";"}
```

Note: La lecture d'une unité en avance est ici indispensable à cause des opérateurs.

in_integer(t)

Lit d'abord tous les caractères blancs, puis essaie de lire un entier sur l'unité courante. Si l'objet lu n'a pas la syntaxe d'un entier, alors rien n'est lu et *in_integer* échoue. Sinon *t* est unifié avec l'entier qui a été lu.
Exemple:

```
>in_integer(i) in_char(c) in_integer(j);
-123+456
{i=-123,c="+",j=456}
```

ATTENTION : l'écriture des entiers et réels pouvant se faire sur plusieurs lignes, à l'aide de "\<return>" (si l'option d'interprétation du '\' est active (cf. U2.3.)), si l'objet en entrée n'est pas un entier, mais un réel écrit sur plusieurs lignes, Prolog ne peut pas restaurer les lignes précédant la dernière.

in_real(t)

Lit d'abord tous les caractères blancs, puis essaie de lire un réel sur l'unité courante. Si l'objet lu n'a pas la syntaxe (définie par l'option courante) d'un réel, alors rien n'est lu et *in_real* échoue. Sinon *t* est unifié avec le réel qui a été lu.

```
> in_real(x) in_char(c) in_real(y);
-123e+4,54.63e0
{x=-1.230000000000000e+06, c=",", y=5.463000000000000e+01}
```

ATTENTION : l'écriture des entiers et réels pouvant se faire sur plusieurs lignes, à l'aide de "\<return>" (si l'option d'interprétation du '\' est active (cf. U2.3.)), si l'objet en entrée n'est pas un réel, mais un entier écrit sur plusieurs lignes, Prolog ne peut pas restaurer les lignes précédant la dernière.

in_double(t)

Identique à *in_real(t)*.

in_string(t)

Lit d'abord tous les caractères blancs, puis essaie de lire une chaîne Prolog. Si l'objet n'a pas la syntaxe d'une chaîne, alors rien n'est lu et *in_string* échoue. Sinon *t* est unifié avec la chaîne lue. Attention : il s'agit bien d'*un terme Prolog de type chaîne* (i.e.: avec les quotes) et non de «n'importe quoi, considéré comme une chaîne».
Exemple:

```
>in_string(s) in_char(c);
"Hello !";
{s="Hello !", c=";"}
```

in_ident(t)

Lit d'abord tous les caractères blancs, puis essaie de lire un identificateur Prolog. Si l'objet lu n'a pas la syntaxe (définie par l'option courante) d'un identificateur, alors rien n'est lu et *in_ident* échoue. Sinon *t* est unifié avec l'identificateur qui a été lu.

Exemple:

```
lire(entier(n)) -> in_integer(n) !;
lire(ident(i)) -> in_ident(i) !;
lire(chaine(s)) -> in_string(s) !;
lire(caractere(c)) -> in_char(c);;
>lire(x1) lire(x2) lire(x3) lire(x4) lire(x5) lire(x6) …;
aa := bb * 123 …
{x1=ident(aa),x2=caractere(":"),x3=caractere("="),
x4=ident(bb),x5=caractere("*"),x6=entier(123), … }
```

in_word(t1, t2)

Lit tous les caractères blancs, puis lit le mot le plus long, le transforme en une chaîne et essaie d'unifier cette chaîne avec *t1*. Un mot est soit :

(1) une suite de lettres
(2) une suite de chiffres
(3) tout caractère qui n'est ni un blanc, ni une lettre, ni un chiffre.

Dans le cas (1), *t2* est l'identificateur correspondant à *t1* si cet identificateur est déjà connu, ou *nil* s'il s'agit de sa première apparition. Dans ce dernier cas, l'identificateur n'est pas créé dans le dictionnaire.
Dans le cas (2), *t2* est l'entier correspondant à *t1*.
Dans le cas (3), *t2* = *t1*.

```
>in_word(x, y);
toto
{x="toto", y=nil}
>in_word(x, y);
insert
{x="insert", y=insert}
>in_word(x, y);
012345
{x="012345", y=12345}
>in_word(x, y);
+
{x="+", y="+"}
```

in_sentence(t1, t2)

Lit une phrase qui se termine par ".", "?", "!" ou tout autre caractère qui a été défini comme terminateur et met la phrase sous la forme de deux listes. *t1* est la liste des unités lexicales constituant la phrase (les mots au sens de *in_word*) et *t2* est la liste des atomes correspondants, comme pour *in_word.*
La liste des terminateurs de phrases peut être modifiée par les prédicats *add_sentence_terminator/1* et *remove_sentence_terminator/1*.

```
>in_sentence(x, y);
ceci est un string.
{x="ceci"."est"."un"."string"."."."nil,
```

```
   y=nil.nil.nil.string.".".nil}
>in_sentence(x, y);
somme := 052345;.
{x="somme".":"."="."052345".";"."."nil,
   y=nil.":"."=".52345.";"."."nil}
```

add_sentence_terminator(C)

Ajoute le caractère *C* dans la liste des terminateurs de phrases (initialement dotée des caractères ".", "?", et "!").

remove_sentence_terminator(C)

Enlève le caractère *C* de la liste des terminateurs de phrases. Provoque un échec si ce caractère n'est pas dans cette liste.

read_unit(t, u)

Lecture d'une unité lexicale.
Lit une unité lexicale et unifie *t* avec son type (identificateur, nombre, etc…) donné par un nombre conventionnel, et *u* avec sa valeur.
Les unités lexicales de Prolog II+ sont décrites par la grammaire donnée au paragraphe 1.9. Relativement aux notations utilisées dans cette grammaire, le type *t* utilisé par *read_unit* est défini comme suit :

valeur de *t*	catégorie lexicale	valeur de *u*
1	identifier	l'identificateur
2	separator[1]	la chaîne correspondante
3	variable	la chaîne correspondante
4	constant	la constante
5	graphic_symbol	la chaîne correspondante

Exemple:

```
> read_unit(t1,u1) read_unit(t2,u2) read_unit(t3,u3)
read_unit(t4,u4);
x12 + :pi ,
{t1=3,u1="x12",t2=5,u2="+",t3=1,u3=pi,t4=2,u4=","}
> read_unit(t,u);
123
{t=4,u=123}
```

read_rule(t1,t2)

Lit sur l'unité d'entrée courante une règle Prolog, et unifie *t1* avec la tête, et *t2* avec la liste des termes de la queue.

Exemple:

```
> read_rule(t1,q1) read_rule(t2,q2);
tt(1,x) -> outl(x) fail;
tt(2,x) -> !;
{t1=tt(1,v118),q1=outl(v118).fail.nil,
 t2=tt(1,v228),q2='!'.nil}
```

---

1 *separator* est un séparateur syntaxique: voir § 1.2 de ce manuel.

```
>
```

sscanf

La fonction C *sscanf* est accessible depuis Prolog avec la primitive *callC* (voir § 7.7. de ce manuel).

```
> callC(sscanf("123","%lf",<"R",y>));
{y=1.230000000000000e+02}
> eq(f,"%x %o") callC(sscanf("12 12",f,<"I",x>,<"I",y>));
{x=18, y=10}
```

5.1.2. Modification de l'unité d'entrée.

Le nom des unités d'entrée, peut être soit une chaîne Prolog, soit un identificateur Prolog, soit un entier. Toutefois, les fichiers et les fenêtres prédéfinies sont toujours désignés par des chaînes.

input(u)

input(u,n)

L'unité dont le nom est *u*, devient l'unité d'entrée courante. Si cette unité ne figure pas parmi les unités ouvertes, alors son descripteur est créé et ajouté dans la pile des unités d'entrée ouvertes. Si l'unité est nouvelle, un fichier est recherché et ouvert. Un fichier de commandes ouvert par *input* peut lui-même contenir une commande *input*, la nouvelle unité courante est empilée au-dessus de la précédente. Dans la forme à deux arguments, *n* précise la taille, en nombre de caractères, du tampon de l'unité. Elle doit être comprise entre 400 et 32K. Si *n* n'est pas dans cet intervalle, la taille du tampon sera de la valeur la plus proche.

input_is(u)

Unifie *u* avec le nom de l'unité d'entrée courante.

close_input

L'unité courante est fermée (sauf s'il s'agit de la console ou d'une fenêtre) et son descripteur est enlevé du sommet de la pile des unités: l'unité précédente redevient unité courante.

close_input(u)

L'unité de nom *u*, qui doit figurer dans la liste des unités d'entrée ouvertes, est fermée (sauf s'il s'agit de la console ou d'une fenêtre) et son descripteur est enlevé de cette liste.

clear_input

Saute les caractères qui restent non lus sur la ligne courante d'entrée.

5.2. Sorties

A travers les primitives *out*, *outm* et *line*, Prolog tient à jour un pointeur de ligne pour chaque unité de sortie. C'est l'effacement de la primitive *line* qui le réinitialise. Ce pointeur permet de maintenir en interne, une représentation de ce qui se passe sur la ligne courante de l'unité. Il sert en particulier, à déterminer la place qui reste sur la ligne (la longueur de ligne est définie par *set_line_width*) et par conséquent sert à générer un passage à la ligne forcé.

L'utilisation de *set_cursor* n'agit pas sur ce pointeur de ligne, par conséquent la représentation de la ligne que se fait Prolog n'est plus exacte. Il peut arriver que des passages à la ligne forcés (\<return>) soient malvenus. Il est donc conseillé, pour éviter cela, d'utiliser conjointement à *set_cursor*, les primitives *outl* et *outml* plutôt que les primitives *out* et *outm*. Ainsi le pointeur de ligne est constamment réinitialisé et la gestion de la présentation est totalement laissée à *set_cursor*. On notera que le comportement est le même, en faisant appel à des règles externes (C, Pascal, …) qui gèrent la présentation de l'écran.

5.2.1. Règles pour la sortie

beep

Génère une tonalité d'avertissement.

flush

Vide le tampon de caractères de l'unité courante de sortie, en envoyant tous les caractères en attente d'écriture sur l'unité physique associée.

out(t)

outl(t)

Ecrit le terme *t* sur l'unité courante. Tout terme écrit avec *out* peut être relu avec *in* si l'option d'interprétation du caractère spécial «\» est active (cf. §2.3. du manuel d'utilisation). Si le terme est plus long que la longueur restant sur la ligne courante, un retour chariot est inséré en respectant les principes suivants:

(1) Lorsqu'une chaîne ne peut être imprimée sur une seule ligne, elle est coupée à l'endroit requis par insertion d'un retour-chariot *masqué* (c'est-à-dire précédé par le caractère «\»).
(2) Un nombre n'est pas coupé sauf si, tout seul, il est plus long que toute une ligne; dans ce cas il est coupé comme une chaîne.
(3) Un identificateur n'est pas coupé sauf si, tout seul, il est plus long que toute une ligne; dans ce cas il est coupé comme une chaîne.

L'écriture du terme se fait avec les conventions suivantes:

(1) Les listes sont imprimées en notation pointée pour la syntaxe Prolog II, en notation avec des crochets carrés pour la syntaxe Edinburgh ou si l'option "notation standard des réels" est activée.

(2) Si une chaîne contient un caractère non imprimable, celui-ci est écrit sous forme d'escape séquence imprimable (voir *outm*).
(3) Tout identificateur ne répondant pas à la syntaxe d'un identificateur est quoté.
(4) Le contexte courant d'exécution détermine les abréviations possibles pour les identificateurs.

outl(t) équivaut à la suite de buts *out(t) line* : lorsque le terme *t* a été écrit on produit le passage à la ligne suivante.
Exemple:

```
> out(1.Pierre."Salut!".nil);
1.Pierre."Salut!".nil{}
>
```

N.B. : Les accolades imprimées à la suite du terme indiquent la réussite de l'effacement du but *out(1.Pierre."Salut!".nil)*. Une manière d'en empêcher l'impression consiste à faire échouer artificiellement cet effacement :

```
>out(1.Pierre."Salut!".nil) line fail;
1.Pierre."Salut!".nil
>
```

outm(s)

outml(s)

Ecrit la chaîne *s* sur l'unité active de sortie, sans écrire les quotes et en interprétant correctement les caractères de formatage que *s* peut contenir (par exemple: \n). Si la chaîne est plus longue que la place restante sur la ligne en cours, alors elle est coupée par un retour chariot «masqué». Les caractères correspondant à un code ISO 8859-1 non imprimable sont envoyés sur l'unité courante de sortie sans transformation.
Exemple:

```
> out("\tBonjour!") line fail;
"\tBonjour!"
> outm("\tBonjour!") line fail;
        Bonjour!
>
```

outml(s) équivaut à *outm(s) line*.

outm(s, n)

Ecrit *n* fois la chaîne *s* avec les mêmes conventions que ci-dessus.
Exemple :

```
>outm("-",40) line fail;
----------------------------------------
>
```

out_equ(t)

Ecrit le système minimal d'équations du terme *t*. Cette règle est beaucoup plus lente que *out*, mais factorise les sous-arbres et donne donc, entre accolades, une représentation plus compacte quand c'est possible.

line

Va à la ligne.

page

Va à la page. Sur l'unité "console", l'écran est effacé et le curseur positionné en haut à gauche de l'écran. N'a pas d'effet dans l'environnement graphique.

paper

Provoque la copie de ce qui se passe à la console dans un fichier journal (*prolog.log* par défaut).

no_paper

Annule l'effet de *paper*. Le fichier *journal* est fermé uniquement en fin de session. Un appel ultérieur à *paper* viendra ajouter des informations à la fin du fichier.

set_cursor(n1, n2)

Le curseur est positionné en (*n1*,*n2*) sur l'écran, *n1* étant la coordonnée de la colonne et *n2* celle de la ligne. (*1*,*1*) correspond au coin en haut à gauche de l'écran. On doit avoir :
1 ≤ *n1* ≤ largeur de la ligne et 1 ≤ *n2* ≤ nombre de lignes de l'écran.
N'a pas d'effet dans l'environnement graphique.

set_line_cursor(n)

Positionne le pointeur courant de caractères à la position *n* sur la ligne courante. Cela fournit une sorte de tabulation. *set_line_cursor* ne revient pas en arrière à partir de la position courante du pointeur. La position du premier caractère est 1.

sprintf

La fonction C *sprintf* est accessible depuis Prolog avec la primitive *callC* (voir § 7.7. de ce manuel).

```
> callC(sprintf(<"",x,80>,"valeur: %ld",200));
{x="valeur: 200"}
```

L'utilisation des primitives suivantes nécessite le chargement préalable du module de dessin d'arbres (voir § 5.3).

draw_equ(t)

Dessine le système minimal d'équations représentant un arbre fini ou infini *t* . Pour utiliser cette primitive, il faut que le module de dessin d'arbres soit chargé.

```
> ... draw_equ(aa(bb(cc,dd),bb(cc,dd))) ...
    aa
 +--^--+
v1026  v1026

v1026 = bb
       +^-+
      cc  dd
{}
```

draw_tree(t)

Dessine l'arbre fini *t* sur l'écran (si le terminal a des possibilités graphiques, les symboles semi-graphiques sont utilisés). Pour utiliser cette primitive, il faut que le module de dessin d'arbres soit chargé.

```
> ... draw_tree(aa(bb(cc,dd),bb(cc,dd))) ...
       aa
    +--^---+
    bb     bb
   +^-+   +^-+
  cc  dd cc  dd
{}
```

draw_mode(x)

Cette primitive, ainsi que la suivante concernent la manière dont les arbres sont dessinés. En effet, suivant le type d'écran certains caractères semi-graphiques sont utilisés pour obtenir des dessins d'arbres plus jolis. *draw_mode(x)* fournit dans *x* une chaîne correspondant au type d'écran utilisé. Les valeurs possibles de *x* sont dépendantes de la machine: "VT100", "GRAPHICS", "TTY". "TTY" qui correspond à un écran sans possibilité graphique, existe sur toutes les machines. Pour utiliser cette primitive, il faut que le module de dessin d'arbres soit chargé.

set_draw_mode(x)

Permet de choisir le type d'écran utilisé pour le dessin des arbres. Les valeurs de *x* sont les mêmes que précédemment. Si vous voulez faire imprimer des arbres sur une imprimante classique, il faut se mettre en mode "*TTY*". Pour utiliser cette primitive, il faut que le module de dessin d'arbres soit chargé.

5.2.2. Modification de l'unité de sortie

Le nom des unités de sortie, peut être soit une chaîne Prolog, soit un identificateur Prolog, soit un entier. Toutefois, les fichiers et les fenêtres prédéfinies sont toujours désignés par des chaînes.

output(u)

output(u,n)

L'unité de nom *u* devient l'unité courante de sortie. Si l'unité désignée par *u* ne figure pas déjà dans la liste des unités de sortie ouvertes, alors un descripteur pour l'unité est alloué et éventuellement un fichier est créé ayant le nom indiqué par *u*. Dans la forme à deux arguments, *n* précise la taille, en nombre de caractères, du tampon de l'unité. Elle doit être comprise entre 400 et 32K. Si *n* n'est pas dans cet intervalle, la taille du tampon sera de la valeur la plus proche. Dans la forme à un seul argument, le tampon aura une taille de 400 caractères.

En aucun cas ces primitives ne créent une fenêtre ou un fichier mémoire, il faut utiliser pour cela les primitives spécifiques. Si *u* est une chaîne de caractères terminée par le signe "+", le fichier dont le nom est la chaîne privée de ce signe est alors ouvert en mode *append*, c'est à dire que si ce fichier existe déjà sur disque, les sorties qui suivront vont y être **ajoutées en fin**.

output_is(u)

Fournit dans *u* le nom de l'unité de sortie courante.

close_output

L'unité courante est enlevée du sommet de la pile des unités de sortie ouvertes. Si elle correspond à un fichier, alors celui-ci est fermé.

close_output(u)

u est le nom d'une unité qui doit figurer dans la liste des unités de sortie ouvertes. S'il s'agit d'un fichier, il est fermé. Le descripteur de l'unité est enlevé de cette liste et détruit. En aucun cas cette primitive ne ferme une fenêtre, il faut utiliser pour cela les primitives spécifiques.

line_width(n)

Fournit dans *n* la longueur maximum actuelle des lignes de l'unité courante, définie par *set_line_width.*

set_line_width(n)

Permet de définir la longueur maximale des lignes de l'unité courante de sortie. *n* devient la nouvelle longueur de ligne de l'unité. Par défaut, la longueur maximale de la ligne est de 80 caractères. Dans tous les cas, elle est limitée à 400 caractères. Si aucun passage à la ligne suivante n'a été demandé, avant que la longueur maximum de la ligne ait été atteinte, Prolog le force, en insérant un <retour chariot> entre deux unités ou bien la séquence <\><retour chariot> à l'intérieur d'une unité.

echo

Active l'option qui provoque l'affichage sur l'unité *"console"* des caractères lus ou écrits sur une autre unité, de type fichier.

no_echo

Annule l'effet de *echo*.

5.3. Chargement et adaptation du module de dessin

Le module *Dessin* ne figure pas dans l'état initial qui vous est livré. Vous pouvez l'inclure en chargeant le fichier binaire:

```
> load("dessin.mo");
```

Pour charger à partir du répertoire Prolog, taper la commande:

```
> getenv("PrologDir2",S)
  conc_string(S,"dessin.mo",S1)
  load(S1);
```

Vous pouvez également recompiler le module source:

```
> insert("dessin.m2");
```

Ce fichier crée un nouveau module, *Dessin*, contenant le programme de tracé d'arbres et d'équations. Ceci rend disponible les règles suivantes (voir §5.2.1.):

draw_equ
draw_tree
draw_mode
set_draw_mode
gr_tree_click si le mode graphique est activé.

S'agissant d'un module de dessin il faut, bien entendu, y incorporer certains renseignements sur les possibilités semi-graphiques des terminaux utilisés. A cette fin, un "mode de dessin" (*draw_mode*) est associé à chaque protocole de gestion de terminal.

Quelques configurations sont déjà définies : vt100, tty, …. Il vous appartient d'ajouter à *dessin.m2* celles qui correspondent aux terminaux que vous utilisez.

Pour définir un nouveau mode de dessin, vous devez:

1. Lui donner un nom interne, qui est un identificateur (tty, vt100, etc...) un nom externe, qui est une chaîne de caractères ("TTY", "VT100", etc...) et associer ces deux noms en ajoutant une assertion:

   termi(nom_externe,nom_interne) ->;

   Notez que l'utilisateur final n'est censé connaître que le nom externe.

2. Ajouter deux nouvelles définitions des règles *en_graphique* et *hors_graphique* qui commandent le passage du terminal du mode caractères au mode semi-graphique et réciproquement.

3. Ajouter une nouvelle règle *config* définissant les chaînes de caractères qui, en mode graphique, provoquent l'affichage des graphismes indiqués.

Remarque: Dans tous les cas, le mode "TTY" permet d'obtenir des dessins approximatifs sans utiliser aucune possibilité graphique du terminal. Ceci peut s'avérer utile, par exemple, pour imprimer des dessins d'arbres sur une imprimante classique.

5.4. Déclaration d'opérateurs

Les opérateurs permettent une notation sans parenthèses des termes fonctionnels à un ou deux arguments. Le symbole fonctionnel est remplacé par un opérateur placé en position:

- *infixé* pour les termes fonctionnels à deux arguments.

- *préfixé* ou *postfixé* pour les termes fonctionnels à un argument.

Un opérateur est défini par un symbole, une précédence, et des règles de parenthésage. Pour lever l'ambiguïté des expressions contenant plusieurs opérateurs, on parenthèse d'abord les opérateurs de plus faible précédence. Lorsque des opérateurs ont même précédence, on utilise les règles de parenthésage. La syntaxe des expressions avec opérateurs et les opérateurs prédéfinis sont donnés section 1.9.2 de ce manuel.

```
Exemple:
    /  opérateur binaire de précédence 400, parenthésage à gauche d'abord
    -  opérateur binaire de précédence 500, parenthésage à gauche d'abord
expression:                3 - 2 - 1 / 4 / 3
expression non ambiguë:    ((3 - 2) - ((1 / 4) / 3)))
arbre Prolog:              sub(sub(3,2),div(div(1,4),3))
```

Il faut remarquer que si la notation avec opérateur est plus légère, son abus peut mener à une réelle difficulté de compréhension des arbres effectivement représentés par des expressions contenant des opérateurs peu usités.

op(n,i1,s)
op(n,i1,s,i2)

Déclarent l'opérateur *s* avec la précédence *n* (0 <= *n* <= 1200) et le type de parenthésage *i1*. *s* est un identificateur, ou une chaîne représentant un symbole graphique. Dans la forme à 3 arguments, *s* peut être une liste, la déclaration s'applique alors à chaque élément de la liste. La forme à 3 arguments est aussi une directive de compilation, c'est à dire que la déclaration d'un opérateur peut aussi se faire au milieu d'un source que l'on compile. Le symbole fonctionnel représenté par *s* est *i2* si celui-ci est défini, sinon le symbole *s* quoté. L'associativité *i1* est précisée par un identificateur combinant deux ou trois des lettres *f,x,y* avec les conventions suivantes:

f représente l'opérateur.
x représente une expression de précédence inférieure à *f*.
y représente une expression de précédence inférieure ou égale à *f*.

Les combinaisons possibles sont:

fx Opérateur préfixé avec opérande de précédence inférieure.
fy Opérateur préfixé avec opérande de précédence inférieure ou égale.
xf Opérateur postfixé avec opérande de précédence inférieure.
yf Opérateur postfixé avec opérande de précédence inférieure ou égale.
xfx Opérateur infixé avec opérandes de précédence inférieure.
yfx Opérateur infixé admettant à gauche un opérande de même précédence, avec parenthésage gauche droite.
xfy Opérateur infixé admettant à droite un opérande de même précédence, avec parenthésage droite gauche.

Lorsque *n* est égal à 0, *op/3* a pour effet de supprimer la déclaration existante pour l'opérateur *s*.
Lorsque les arguments de *op/3* sont des variables libres, ils sont unifiés successivement avec les définitions des opérateurs existants.

Exemple:

```
> insert;
op(900,fy,not) op(700,xfx,"=",eq);
not x -> x ! fail;
not x ->; ;
{}
> (not 1=2)[1];
{}
> split(1=2,L);
{L=eq.1.2.nil}
```

1 Se rappeler qu'en syntaxe Prolog II les opérateurs doivent être parenthésés au premier niveau d'une queue de règle.

6. L'environnement

6.1. Comment sortir de Prolog
6.2. Démarrage automatique d'un programme Prolog
6.3. Edition de programmes
6.4. Date, temps et mesures
6.5. Lien avec le système
6.6. Utilisation du debugger
6.7. Modification et visualisation de l'état courant
6.8. Gestion automatique des espaces et des piles

Parmi les règles prédéfinies qui permettent à l'utilisateur de communiquer avec l'extérieur de Prolog, ce chapitre décrit celles qui sont indépendantes du système particulier utilisé. Celles qui, au contraire, dépendent de l'implantation considérée sont décrites dans le manuel d'utilisation.

6.1. Comment sortir de Prolog

exit

exit(s)

Fait sortir de Prolog en sauvant la totalité des programmes couramment connus, y compris ceux qui constituent le superviseur Prolog II+. La sauvegarde est faite dans le fichier de nom *s*, ou bien pour la forme sans argument dans le fichier de nom *prolog.po*.
Remarque : Il est possible de sauver uniquement les modules que vous avez créés ou modifiés pendant la session (commande *save* - cf. paragraphe 3.7) et de quitter Prolog en faisant la commande *quit*.

quit

quit(n)

Fait quitter Prolog sans rien sauver. La valeur *n* est envoyée au système d'exploitation à titre de «statut de terminaison». Quelque soit ce système, si *n* vaut 0 alors la valeur envoyée est celle qui correspond à «pas d'erreur» (cette valeur peut être non nulle).
La première forme *quit*, équivaut à *quit(0)*.

6.2. Démarrage automatique d'un programme Prolog

Si l'état binaire sur lequel Prolog est lancé, contient des modules possédant une règle *ini_module*, chacune de ces règles est effacée (dans un ordre défini par Prolog) dès la fin de l'initialisation. Ensuite, Prolog tente d'effacer le but *:to_begin*.

La règle *:to_begin* n'est pas définie dans l'état initial. Si l'utilisateur veut faire démarrer automatiquement un programme, il doit ajouter la règle:

:to_begin -> xxxx ;

où *xxxx* est le but initial à effacer de son programme d'application. Il doit ensuite sauvegarder cet état. Lorsque plus tard il appellera Prolog avec ce nouvel état, *xxxx* sera immédiatement exécuté.

6.3. Edition de programmes

Prolog II+ étant un compilateur incrémental, il vous permet d'éditer des règles sans quitter Prolog, en utilisant un éditeur que nous appellerons ici *éditeur résident*.

L'éditeur résident est activé par la commande Prolog *edit*

Il existe plusieurs variantes de cette commande pour:

- modifier un ou plusieurs paquets de règles.
- éditer et recompiler un module source.
- éditer un fichier de texte quelconque.

A propos d'édition, il faut savoir que :

- les règles sont supprimées de l'espace du code de Prolog pour être écrites dans le fichier de texte qui sert pour l'édition. A la fin de l'édition, le fichier est réinséré.
- l'édition compile et par conséquent l'édition de faits non compilés les transformera en règles compilées.
- à tout moment, les règles n'existent qu'en un seul exemplaire : soit comme code Prolog, soit comme source Prolog.
- si une interruption utilisateur survient, l'espace du code sera tout de même modifié, de façons différentes en fonction du moment de l'interruption.
- l'édition de règles non visibles provoque la suppression de ces règles, mais ne les fournit pas sous forme de source!

6.3.1. Modifier des paquets de règles

Cette commande provoque l'édition des paquets de règles listés en argument. A la fin de l'édition, les paquets édités sont réinsérés à la place des paquets originaux. Les paquets peuvent appartenir à plusieurs modules différents, c'est alors le contexte de lecture/écriture du module du premier paquet qui est utilisé.

edit(i/a)
edit(l)

i/a représente le paquet de règles dont l'accès est l'identificateur *i*, et le nombre d'arguments est *a*.
l est une liste (terminée ou non par *nil)* d'éléments *i/a*.

Au retour d'un appel de l'éditeur résident, Prolog peut déceler une erreur de syntaxe dans les règles que vous venez d'éditer. Il affiche alors un message avec le numéro et le contenu de la ligne contenant l'erreur, le diagnostic correspondant et vous demande si vous désirez recommencer le cycle d'édition et compilation afin de corriger la faute, ou abandonner.

L'abandon provoque le retour à prolog, les règles non insérées sont supprimées.

Note:

Il peut arriver que le texte que vous venez d'éditer comporte une erreur difficile à déceler.

Sachez qu'à la suite d'une utilisation de l'éditeur résident, un fichier nommé PROEDIT.PRO a été créé dans le répertoire courant: il contient les règles éditées dans leur état final.

Ce fichier peut vous aider à reconstituer l'état courant de votre programme. Pour cela, abandonnez l'édition. Vous vous retrouvez alors sous Prolog, sans les règles éditées non encore lues. Sauvez ce programme et, hors Prolog, utilisez un éditeur de textes pour analyser le morceau manquant depuis PROEDIT.PRO.

6.3.2. Editer et recompiler un module source

Cette fonctionnalité permet de modifier et recompiler le texte source d'un module déjà chargé: l'édition se fait avec la mise en forme originale.

editm(m)

m est une chaîne. Supprime toutes les règles du module *m*, et lance l'édition du fichier source correspondant. La version éditée est recompilée et vient remplacer l'ancienne version.
Si le fichier source ne peut être trouvé, alors l'édition se fait sur un source créé par décompilation du module. Ce fichier source est appelé PROEDIT.Mx où x est un numéro engendré par Prolog et de valeur différente pour des modules de nom différent.

Note:
Les règles qui auraient pu être ajoutées par *insert* ou *assert* dans le module pendant l'exécution sont effacées avant rechargement du module.

6.3.3. Editer un fichier de texte quelconque

edit(s)

s est une chaîne. Permet d'éditer le fichier de nom *s*.

6.4. Date, temps et mesures

reset_chrono, chrono(x)

chrono unifie *x* avec le temps en secondes écoulé depuis le dernier *reset_chrono*.

reset_cpu_time, cpu_time(x)

cpu_time unifie *x* avec le temps cpu en millisecondes écoulé depuis le dernier *reset_cpu_time*.

date(j,m,a,s)

Calcule les arguments *j*,*m*,*a*,*s* de la date courante. Les quatre arguments désignent respectivement le jour du mois, le mois, l'année et le rang du jour dans la semaine (dimanche=0, lundi=1,...).

date_string(s)

Calcule la date courante, exprimée en anglais, et unifie la chaîne correspondante avec *s*.

date_stringF(s)

Calcule la date courante, exprimée en français, et unifie la chaîne correspondante avec *s*.

week(n,s)

Donne dans *s* la chaîne correspondant au jour de la semaine dont le rang est *n* et vice versa.

month(n,s)

Donne dans *s* la chaîne correspondant au mois dont le rang est *n* et vice versa.

delay(n)

Temporise *n* millisecondes.

time(x)

Donne un temps absolu en secondes. L'origine de ce temps est variable d'une machine à une autre.

time(h,m,s)

Calcule les arguments *h, m, s* de l'heure courante. Ils représentent respectivement l'heure, les minutes et les secondes.

6.5. Lien avec le système

sys_command(s)

Permet d'exécuter une commande du système d'exploitation représentée par la chaîne *s*. Quand la commande appelée provoque une erreur, *sys_command* génère une erreur "ERREUR SYSTEME" avec en complément d'erreur le status retourné par le système d'exploitation.
Sous certains systèmes d'exploitation (UNIX par exemple), ce prédicat peut être grand consommateur de mémoire. On pourra alors le redéfinir à l'aide d'un coprocessus (sur ces systèmes, voir le chapitre sur les coprocessus).

getenv(s,x)

Unifie *x* avec la chaîne représentant la valeur de la variable système d'environnement dont le nom est défini dans la chaîne *s*. Si la variable n'est pas définie, *x* est unifié avec une chaîne vide.

set_import_dir(s)

Définit un répertoire de référence, qui est consulté après le répertoire courant quand les fichiers n'y ont pas été trouvés, par les primitives qui manipulent des fichiers en lecture telles que : *input*, *insert*, *load*. La chaîne de caractère *s* concaténée à un nom de fichier, doit définir un nom complet compréhensible par le système d'exploitation.

lkload(s1,s2)

Cette primitive n'est disponible que sous certains systèmes d'exploitation, voir le manuel d'utilisation. Lorsqu'elle est présente, elle permet d'effectuer un lien et un chargement dynamiques de programmes externes.

ms_err(n, s)

Unifie *s* avec le texte de l'erreur numéro *n*. Pour cela, Prolog fait une recherche dans le fichier d'erreurs *err.txt* qui doit être composé d'une suite de lignes du format suivant:

entier espace suite de caractères qui compose le texte de l'erreur.

Il est possible de rajouter des messages dans ce fichier en respectant le format indiqué. Il est conseillé d'utiliser des numéros de message au delà de 1000.

6.6. Outil de mise au point de programmes

Le debugger Prolog, ou outil de mise au point de programmes, permet une trace interactive des programmes sans modifier leur sémantique, et sans consommation supplémentaire de mémoire. Il permet, entre autres, de positionner des points d'arrêt, de pister en pas à pas avec saut éventuel de morceaux de programme, d'imprimer l'état de la démonstration en cours, et de visualiser les *backtracking*.

Le prédicat prédéfini *statistics*, après une éxécution du programme avec le debugger actif, peut aider à configurer les espaces de Prolog.

Note: Le code optimisé ne permet de reconstruire le source que de manière équivalente, et ne permet pas au debugger de visualiser toutes les informations s'y rapportant. Si l'utilisateur désire suivre une représentation exacte de son programme, il faut que les règles aient été compilées sans optimisation avec l'option adéquate, au lancement de Prolog.

6.6.1. Mode trace

L'outil de mise au point mixe deux modes de fonctionnement :

- un mode trace, où il visualise l'exécution,
- un mode interactif, où il permet de prendre le contrôle au cours de l'exécution et d'effectuer des actions.

Dans un premier temps, si l'on ne veut pas se plonger dans le détail des commandes du mode interactif, le mode trace permet d'afficher toutes les informations nécessaires à la compréhension du programme.

Le mode trace est donc une approche simple et immédiate pour se rendre compte de l'exécution d'un programme.

Chaque but effacé est affiché avec ses arguments unifiés sur la règle choisie pour l'effacement courant. Les backtracking sont annoncés en précisant le prédicat pour lequel il restait des choix et le rang dans le paquet de la nouvelle règle.

Si le programme d'exemple menu.p2 est chargé et si le mode trace est actif, la séquence suivante montre la présentation de ces informations:

```
> repas(e,Chapon_farci,d);
hors_d_oeuvre( Artichauts_Melanie)
plat( Chapon_farci)
RECALL(2): plat / 1
plat( Chapon_farci)
poisson( Chapon_farci)
dessert( Sorbet_aux_poires)
{e=Artichauts_Melanie,d=Sorbet_aux_poires}
RECALL(2): dessert / 1
dessert( Fraises_chantilly)
{e=Artichauts_Melanie,d=Fraises_chantilly}
RECALL(3): dessert / 1
dessert( Melon_en_surprise)
{e=Artichauts_Melanie,d=Melon_en_surprise}
…
```

Pour faciliter la lisibilité (pour des programmes qui font eux-mêmes des écritures sur la sortie courante), les informations fournies par la trace peuvent être redirigées sur un fichier.

Activation, désactivation

Le mode trace est actif dès l'effacement d'un des prédicats prédéfinis *trace/0* ou *trace/1*. Il est désactivé par l'effacement du prédicat prédéfini *no_trace/0*.

trace

Active la trace.

trace(f)

Active la trace. Redirigera toutes les impressions de la trace dans le fichier de nom *f* (chaîne Prolog). Le fichier est fermé à l'exécution d'un prédicat de désactivation de l'outil de mise au point. Si les impressions de mise au point étaient déjà redirigées, le précédent fichier est fermé pour être remplacé par *f*.

no_trace

Désactive la trace.

6.6.2. Mode interactif

Si la trace est simple à activer, elle peut rapidement donner un flot d'informations important, où il serait difficile de se repérer. A ce moment là, le mode interactif devient d'une grande utilité puisqu'il permet d'aller directement à l'information importante. Ses atouts sont :

- les commandes de progression dans le code,
- le paramétrage de l'information à produire,
- les commandes annexes sur l'exécution.

Le mode interactif permet de choisir les informations qu'on veut extraire du programme, dans le format approprié, sans modifier le programme, et seulement dans les portions de code désignées. Il permet surtout, de prendre le contrôle où l'on souhaite, pour lancer des actions.

6.6.2.1. Points d'arrêt

En mode interactif, le debugger va proposer sa banière d'invite (DBG) et attendre une commande sur chaque point d'arrêt. L'important pour avoir accès à l'interpréteur de commande est de bien définir les points d'arrêt. On en distingue deux types:

- les points d'arrêt fixes: ils subsistent tant que le mode debug est actif et tant qu'ils ne sont pas désactivés par commande;
- les points d'arrêt momentanés: ils ne sont valides que le temps d'une commande, qui définit en fait le prochain point de contrôle du debugger.

Un point d'arrêt se définit sur un prédicat, le debugger s'arrête ensuite dès que le programme courant fait appel à ce prédicat. L'arrêt de la machine Prolog se fera donc au moment où le but doit être effacé, avant que l'unification avec la tête de la règle choisie soit faite.

Des commandes spécifiques, ainsi que des prédicats prédéfinis, permettent de définir les points d'arrêt fixes. Les points d'arrêt momentanés sont définis implicitement par une commande de progression ou à l'activation du debugger.

Quand la machine Prolog tourne, avec le debugger actif, une interruption utilisateur poste un point d'arrêt momentané sur le but courant et le mode interactif est réactivé. Les interruptions utilisateur sont ainsi interceptées et ne produisent pas d'erreur.

Voir les commandes du debugger *+, -, b* et les règles prédéfinies *spy, no_spy, show_spy*.

6.6.2.2. Progression dans le code

La progression dans le code se fait de point d'arrêt en point d'arrêt.

Trois granularités de progression sont possibles. On les utilisera en fonction de la "proximité" de l'erreur, ou du type d'information que l'on veut extraire d'une partie du programme.

Les progressions possibles, à partir d'un point d'arrêt, sont:

1er cas: on connait l'endroit du programme qui fait défaut, on veut y aller directement et rapidement.

> Réponse: (go to spy) aller jusqu'au prochain point d'arrêt fixe, qui peut se trouver n'importe où dans le code.

2ème cas: on veut voir "de près" l'exécution de la fonction, sans descendre au niveau bas de ses "sous-programmes".

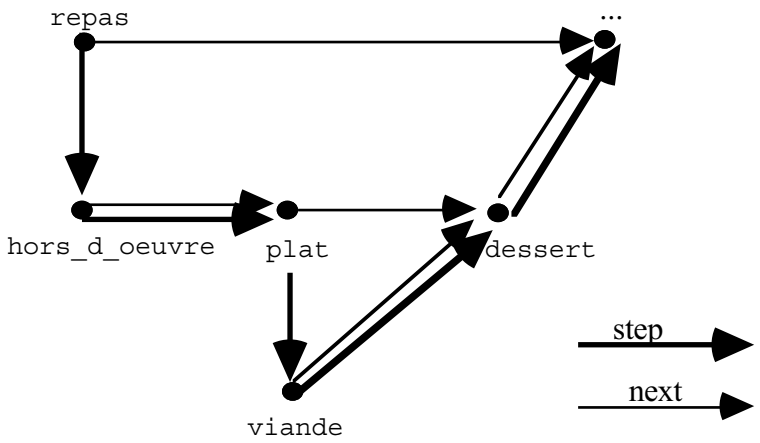
> Réponse: (next) aller jusqu'au prochain but de même niveau ou de niveau supérieur. On ne s'intéresse pas aux niveaux[1] inférieurs, c'est à dire que l'on ignore le corps du but courant.

3ème cas: on veut voir exactement toute l'exécution de la fonction, sans oublier aucune instruction.

> Réponse: (step) aller jusqu'au prochain but.

[1]Définition du niveau d'un but : on dira qu'un but de la question est de niveau 0. Si un but est de niveau n, les buts apparaissant dans la queue de sa règle sont tous du même niveau, égal à n+1.

Schématisons sur notre exemple de menu, les progressions possibles par next et step :



On remarquera que sur les assertions, step et next ont le même effet.

Les commandes de progression dans le code permettent de définir quel doit être le prochain arrêt et quelles informations doivent être obtenues entre temps. En effet, à chaque progression, on pourra choisir ou pas d'afficher les informations sur l'exécution de la portion de programme.

Voir les commandes du debugger *RC, n, N, g, t.*

6.6.2.3. Terminer l'exécution

Ayant suffisamment dégrossi une partie du programme, il faut maintenant le terminer. Il est possible de :

- (go to end) continuer jusqu'à la fin du programme : en imprimant toutes les informations, en imprimant seulement celles concernant les points d'arrêt fixes, en quittant le mode debug et continuant normalement.

- (abort) interrompre le programme.

- (quit) quitter Prolog.

Voir les commandes du debugger *a, e, G, T, q*.

6.6.2.4. Affichage des informations

Nous allons décrire maintenant à quels moments se font les affichages et sous quelle forme.

Quand?

Un Backtracking est généré

L'information est affichée. Elle indique que c'est une alternative (RECALL), le numéro de la nouvelle règle choisie et le but qu'on essaie à nouveau d'effacer. Le but à réessayer fait partie des buts à effacer, il peut donc être utilisé comme point d'arrêt. Par exemple, sur l'éxécution :

```
> plat(Chapon_farci);
CALL: plat( Chapon_farci)
DBG:
CALL: viande( Chapon_farci)
DBG:
RECALL(2): plat( Chapon_farci)
DBG:
CALL: poisson( Chapon_farci)
DBG:
{}
```

L'information sur le backtracking est :

```
RECALL(2): plat( Chapon_farci)
```

Le numéro de la règle à réessayer est noté entre parenthèse `(2)`. Un point d'arrêt est mis au nouvel appel du but, sur :

```
plat( Chapon_farci)
```

On a alors accès, sur cette autre alternative, à toutes les possibilités offertes par l'outil de mise au point.

Autre exemple, si un point d'arrêt fixe est mis sur `plat/1`, le debugger s'arrête sur toutes ses alternatives :

```
> repas(e,Chapon_farci,d);
CALL: repas( v147, Chapon_farci, v284)
DBG: +plat/1                    (ajoute le point d'arrêt)
DBG: g                          (go to spy)
CALL: plat( Chapon_farci)
DBG: g
RECALL(2): plat( Chapon_farci)
DBG: g
{e=Artichauts_Melanie,d=Sorbet_aux_poires}
{e=Artichauts_Melanie,d=Fraises_chantilly}
{e=Artichauts_Melanie,d=Melon_en_surprise}
CALL: plat( Chapon_farci)
DBG: g
RECALL(2): plat( Chapon_farci)
DBG: g
…
```

Une erreur est générée par la machine ou par un prédicat externe

Les points d'arrêt ne sont pas modifiés. Le message d'erreur est affiché, même si l'erreur est récupérée par un block, ainsi que la branche de résolution, au moment de l'erreur. Par exemple, sur l'exécution suivante :

```
> insert;
toto(x,y) -> block(x,titi(y)) outl(x);
```

```
titi(y) -> string_ident(y,z) outl(z);
;
{}
> debug toto(x,12);
CALL: toto( v156, 12)
DBG:
block
CALL: titi( 12)
DBG:
CALL: string_ident( 12, v360)
DBG:
   ARGUMENT DE MAUVAIS TYPE
^^^^^^^^^^^^^^^^TOP^^^^^^^^^^^^^^^^^^
^^^^^^^^ titi / 1, rule number 1
^^^^^^^^ toto / 2, rule number 1
^^^^^^^^^^^^^^^BOTTOM^^^^^^^^^^^^^^^^
CALL: outl( 253)
DBG:
```

L'information sur l'erreur est :

```
   ARGUMENT DE MAUVAIS TYPE
^^^^^^^^^^^^^^^^TOP^^^^^^^^^^^^^^^^^^
^^^^^^^^ titi / 1, rule number 1
^^^^^^^^ toto / 2, rule number 1
^^^^^^^^^^^^^^^BOTTOM^^^^^^^^^^^^^^^^
```

Il est possible de supprimer par commande, pour une session de mise au point, l'affichage de ces informations sur les erreurs. Voir la commande du debugger *s*.

Les erreurs produites en Prolog par *block_exit* ne sont pas mises en évidence par ce mécanisme. La règle prédéfinie *block_exit*, est traitée comme un prédicat ordinaire. Pour être averti de ce type d'erreur, il suffit de mettre un point d'arrêt sur *block_exit/1* et *block_exit/2*.

Sur un point d'arrêt, c'est à dire quand un but est appelé

L'impression des buts peut se faire à deux moments, nous les appellerons avant unification et après unification. Voir les commandes du debugger *p, P*.

Les buts peuvent être imprimés avant l'unification, c'est-à-dire l'impression montre le but tel qu'il apparait dans la résolvante avant son unification avec la tête de la règle choisie pour son effacement.

Ils peuvent être imprimés après unification, c'est-à-dire l'impression a lieu après le choix de la règle et juste avant le prochain point d'arrêt. Par conséquent, on pourra avoir des résultats différents d'une exécution à l'autre pour des progressions de pas différents. Par exemple pour un but donné, si le pas de progression sous le debugger est celui d'un step, la machine Prolog aura simplement unifié le but avec la tête de la règle. Alors qu'après un pas tel que next, la machine Prolog aura exécuté le corps de la règle où des unifications se seront passées, et les arguments du but n'auront pas forcément les mêmes valeurs que précédemment.

Voyons sur notre exemple, la valeur des arguments après unification, dans les deux cas de progression :

```
> repas(e,p,d);
CALL: repas( v156, v191, v226)
DBG: P1                          (valide l'impression après unification)
DBG:
CALL: hors_d_oeuvre( v156)
DBG:
hors_d_oeuvre( Artichauts_Melanie)
CALL: plat( v191)
DBG:
plat( v191)
CALL: viande( v191)
DBG:
viande( Grillade_de_boeuf)
CALL: dessert( v226)
DBG: g                           (go to spy)
{e=Artichauts_Melanie,p=Grillade_de_boeuf,d=Sorbet_aux_poires
}
…
> repas(e,p,d);
CALL: repas( v156, v191, v226)
DBG: P1
DBG:
CALL: hors_d_oeuvre( v156)
DBG:
hors_d_oeuvre( Artichauts_Melanie)
CALL: plat( v191)
DBG: n                           (next)
plat( Grillade_de_boeuf)
CALL: dessert( v226)
DBG: g
{e=Artichauts_Melanie,p=Grillade_de_boeuf,d=Sorbet_aux_poires
}
…
```

Comment?

Présentation

D'une manière générale pour faciliter la lisibilité de la mise au point et différencier visuellement les messages du programme et ceux du debugger, il est possible de définir une marge (en nombre de caractères) pour les impressions du debugger. Quelque soit le message transmis par le debugger, il sera indenté du nombre de caractères de la marge, pour l'affichage.

```
> repas(e,p,d);
          CALL: repas( v156, v191, v226)
          DBG:
          CALL: hors_d_oeuvre( v156)
          DBG:
          CALL: plat( v191)
          DBG: n                 (next)
          CALL: dessert( v226)
          DBG:
{e=Artichauts_Melanie,p=Grillade_de_boeuf,d=Sorbet_aux_poires
}
          RECALL(2): dessert(v190)
```

```
          DBG:
{e=Artichauts_Melanie,p=Grillade_de_boeuf,d=Fraises_chantilly
}
          RECALL(3): dessert(v190)
          DBG:
{e=Artichauts_Melanie,p=Grillade_de_boeuf,d=Melon_en_surprise
}
          RECALL(2): viande(v165)
          DBG:
          CALL: dessert( v226)
          DBG:
```

Une autre alternative est de rediriger les messages du debugger dans un fichier. Dans ce cas, seule la séquence d'invite apparaîtra dans l'unité de trace.

Voir la commande du debugger *i*, ou les règles prédéfinies *debug(f)* et *debug(n,f)*.

Précision

La précision de l'affichage des buts est paramétrable, selon la quantité d'informations que l'on souhaite obtenir. Trois modes sont disponibles :

- impression des prédicats sans leurs arguments éventuels mais précisant l'arité, sous la forme identificateur/arité.

- impression des prédicats avec leurs arguments éventuels, en notation fonctionnelle avec une limite en profondeur paramétrable. A partir d'une profondeur de 1000, la totalité du terme est imprimé. L'impression des arbres infinis en totalité utilise sa représentation en système d'équations. On ne sait pas dire à priori si l'impression restreinte d'un arbre infini est possible.

- impression des prédicats avec leurs arguments éventuels, sous forme d'arbre par le prédicat *Dessin:draw_tree*. Le module de dessin d'arbre doit être chargé pour pouvoir utiliser ce mode. Il est possible de modifier la règle *Dessin:draw_tree*, de manière à redéfinir ce qui est imprimé par le debugger.

Détail de la notation fonctionnelle en profondeur limitée :

On attribue une profondeur à chaque atome du terme à imprimer. Ne seront affichés que les atomes dont la profondeur est inférieure à celle fixée.

Le comptage se fait récursivement à partir du but à afficher, il démarre à 1. La profondeur est incrémentée lorsqu'on passe d'un terme à un sous terme de la manière suivante :

Si un terme fonctionnel f(a1, t2, …, as) apparaît dans un terme de profondeur n alors le foncteur f est de profondeur n, et ses arguments a1, …, as sont de profondeur n+1.

Si un n-uplet <t1, t2, …, ts> (t1 n'est pas un identificateur) apparaît dans un terme de profondeur n alors l'atome <> est de profondeur n, et les arguments du n-uplet t1, t2, …, ts sont de profondeur n+1.

Si une liste x.y apparaît dans un terme de profondeur n, alors x est de profondeur n et y est de profondeur n+1.

Par exemple, voici comment sont définies les profondeurs des atomes pour le terme suivant :

element_de(	12,	1.	2.	3.	4.	5.	nil,	non)
1	2	2	3	4	5	6	7	2

En limite de profondeur 3, il sera imprimé : element_de(12, [1, 2,…], non);

Voir la commande du debugger *m*.

Choix de l'information

Certaines parties du programme ont déjà été prouvées, il n'est pas nécessaire de les vérifier. En mode interactif, par la progression next, on peut éviter de développer le corps d'un prédicat. En mode trace (trace totale ou impression des informations entre deux points d'arrêt) la même possibilité est offerte.

Par exemple, si `plat/1` ne doit pas être développé, on obtient :

```
> trace repas(e,p,d) !;
hors_d_oeuvre( Artichauts_Melanie)
plat( Grillade_de_boeuf)
dessert( Sorbet_aux_poires)
{e=Artichauts_Melanie,p=Grillade_de_boeuf,d=Sorbet_aux_poires
}
…
```

au lieu de :

```
> repas(e,p,d) !;
hors_d_oeuvre( Artichauts_Melanie)
plat( v191)
viande( Grillade_de_boeuf)
dessert( Sorbet_aux_poires)
{e=Artichauts_Melanie,p=Grillade_de_boeuf,d=Sorbet_aux_poires
}
…
```

Voir les commandes du debugger >, <, *j*.

6.6.2.5. Informations annexes sur l'exécution

Pour détailler l'état du programme en cours d'exécution, sur un but particulier, on peut se poser les questions : où en est-on dans l'exécution du programme? Et quelle règle essaie-t-on d'effacer pour ça?

Sur chaque point d'arrêt, le debugger permet de situer le but appelé par rapport à l'arbre de résolution et par rapport à la base de règles.

Quelle est la branche courante de l'arbre de résolution?

```
^^^^^^^^^^^^^^^^TOP^^^^^^^^^^^^^^^^^^
^^^^^^^^ titi / 1, rule number 1
^^^^^^^^ toto / 2, rule number 1
^^^^^^^^^^^^^^^^BOTTOM^^^^^^^^^^^^^^^^
```

Quelle est la position du but dans le programme?

```
--->  repas / 3, rule number 1, goal number 3 in queue
```

En fait, la branche courante de la résolution est donnée par l'état de la pile de récursion. L'optimisation de l'appel terminal pratiquée par la machine Prolog, ne permet pas de visualiser les règles pour lesquelles le but en cours d'exécution est le dernier de la queue. Par exemple :

```
> repas(e,p,d);
CALL: repas( v156, v191, v226)
DBG: s            ici la pile est vide, puisque repas n'est pas encore installé.
^^^^^^^^^^^^^^^^TOP^^^^^^^^^^^^^^^^^^
^^^^^^^^^^^^^^^^BOTTOM^^^^^^^^^^^^^^^^
DBG:
CALL: hors_d_oeuvre( v156)
DBG: s            ici elle contient repas, puisque c'est le but appelant.
^^^^^^^^^^^^^^^^TOP^^^^^^^^^^^^^^^^^^
^^^^^^^^ repas / 3, rule number 1
^^^^^^^^^^^^^^^^BOTTOM^^^^^^^^^^^^^^^^
DBG:
CALL: plat( v191)
DBG: s            ici aussi, puisque plat et hors_d_oeuvre sont de même
                  profondeur.
^^^^^^^^^^^^^^^^TOP^^^^^^^^^^^^^^^^^^
^^^^^^^^ repas / 3, rule number 1
^^^^^^^^^^^^^^^^BOTTOM^^^^^^^^^^^^^^^^
DBG:
CALL: viande( v191)
DBG: s            ici plat n'apparait pas puisque viande est son appel terminal.
^^^^^^^^^^^^^^^^TOP^^^^^^^^^^^^^^^^^^
^^^^^^^^ repas / 3, rule number 1
^^^^^^^^^^^^^^^^BOTTOM^^^^^^^^^^^^^^^^
DBG:
CALL: dessert( v226)
DBG: s            ici repas n'apparait plus puisque dessert est son appel
                  terminal.
^^^^^^^^^^^^^^^^TOP^^^^^^^^^^^^^^^^^^
^^^^^^^^^^^^^^^^BOTTOM^^^^^^^^^^^^^^^^
DBG:
```

Enfin, pour avoir accès au plus grand nombre d'informations, il est possible d'ouvrir une session Prolog. Ainsi, il est possible de vérifier les effets de bords du programme, de relancer une partie du programme, de lancer un programme annexe, de vérifier la base de règles, de consulter l'occupation des piles, … .

A l'ouverture de la session, le prompt de la ligne de commande est affiché dans la console et l'utilisateur peut entrer n'importe quel but Prolog. A la fermeture de la session (par l'effacement de *quit* par exemple) la mise au point continue où elle en était. Il faut noter que les effets de bord de la session sont gardés: affectations, ajout/suppression de règles, ouverture/fermeture de fichier…

Par exemple, l'exécution de `repas(e,Poulet,d)` provoque un échec. Mettons au point :

```
> repas(e,Poulet,d);
CALL: repas( v156, Poulet, v313)
DBG:
CALL: hors_d_oeuvre( v156)
DBG:
CALL: plat( Poulet)
DBG:
CALL: viande( Poulet)
DBG:
RECALL(2): plat( Poulet)
DBG:E                          pourquoi viande(Poulet) a échoué?
                               on ouvre une session,
> list(viande/1);              le prompt Prolog apparait,
viande(Grillade_de_boeuf) ->  ;
viande(Poulet_au_tilleul) ->  ;

{}                             on consulte la base de règles,
> quit;                        on termine la session,
Elapsed time: 19s
Goodbye...
DBG: w                         on est de retour sous le debugger!
--->  plat / 1 , rule number 2, on backtracking
DBG: g
>
```

Voir les commandes du debugger *s, w, E*.

6.6.2.6. Configuration

Comme nous l'avons déjà vu précédemment, il est possible de définir les comportements du debugger. L'ensemble des paramètres de comportement est appelé la configuration.

Certains paramètres de la configuration peuvent être précisés au lancement de l'outil, ils seront valides durant la session de mise au point (c'est à dire tant qu'on n'a pas désactivé l'outil ou tant qu'on ne l'a pas ré-activé avec d'autres paramètres) tant qu'une commande ne les modifie pas.

Les comportements paramétrables sont :

à l'activation ou sous l'interpréteur de commande du debugger :
- la progression dans le code,
- le moment d'impression des arguments,

sous l'interpréteur de commande du debugger uniquement :
- la présentation des informations, dont :
  - un paramètre d'indentation des messages,
  - un paramètre de précision ou profondeur d'impression,
- le choix des informations : sur les erreurs, entre deux points d'arrêt, pendant la trace.

Une configuration standard est prédéfinie. Elle comprend : arrêt sur le prochain but, impression avant unification, aucune indentation, impression des termes par *out* en profondeur 4, impression d'un message et de la pile de récursion sur les erreurs.

Des commandes de l'outil de mise au point permettent de connaître sa configuration et toute autre information relative à la mise au point :

l'état des options,
la liste des points d'arrêt actifs,
la liste des prédicats non développés en trace,
la liste des commandes de l'outil.

Voir les commandes du debugger *S, b, j, h* et la règle prédéfinie *show_spy*.

6.6.3. Activation d'un mode de mise au point

L'outil de mise au point est activé dès l'effacement de l'un des prédicats prédéfinis *trace/0*, *trace/1, debug/0*, *debug/1* ou *debug/2*. Il est désactivé par l'effacement de l'un des prédicats prédéfinis *no_trace/0* ou *no_debug/0*.

Un drapeau permet de définir la configuration de démarrage. Il vaut la somme des valeurs choisies parmi :

1 arrêt sur le premier appel de but.
2 arrêt sur le premier point d'arrêt fixe.
8 Option impression après unification validée
16 Option impression avant unification validée

Un drapeau égal à 0 correspond à la configuration standard prédéfinie.

debug(n) ou
debug(n, f)

sont les **prédicats généraux** d'activation du debugger. *n* doit être un entier connu au moment de l'appel. Il précise la configuration de démarrage, telle qu'elle a été définie ci-dessus. Quand *n* vaut 0, cela correspond à la configuration prédéfinie standard. *f* est un nom de fichier dans lequel seront redirigées les impressions du debugger. Le fichier est fermé à l'exécution d'un prédicat de désactivation de l'outil de mise au point. Si les impressions de mise au point étaient déjà redirigées, le précédent fichier est fermé pour être remplacé par *f*.

debug ou
debug(f)

sont les **prédicats de réactivation** du debugger. Ils ne modifient pas la configuration en place. *f* est un nom de fichier dans lequel seront redirigées les impressions du debugger. Le fichier est fermé à l'exécution d'un prédicat de désactivation de l'outil de mise au point. Si les impressions de mise au point étaient déjà redirigées, le précédent fichier est fermé pour être remplacé par *f*.

trace ou

trace(f)

sont des **prédicats particuliers** du debugger. Ils l'activent dans une configuration précise, celle du mode trace, représentée par le drapeau 19. *f* est un nom de fichier dans lequel seront redirigées les impressions du debugger. Le fichier est fermé à l'exécution d'un prédicat de désactivation de l'outil de mise au point. Si les impressions de mise au point étaient déjà redirigées, le précédent fichier est fermé pour être remplacé par *f*.

no_trace ou

no_debug

sont des **prédicats de désactivation** de l'outil de mise au point, quel que soit son mode, sans altérer la configuration en place.

6.6.4. Gestion des points d'arrêt

Les règles prédéfinies qui permettent de manipuler les points d'arrêt sont:

spy(i/a)

Place un point d'arrêt sur l'entrée de la règle de nom *i* et d'arité *a*. Ce point d'arrêt n'est pris en compte que lorsque le debugger est activé avec un mode le permettant.

no_spy(i/a)

Enlève tout point d'arrêt placé sur la règle de nom *i* et d'arité *a*.

show_spy(l)

Unifie *l* avec la liste des primitives sur lesquelles un point d'arrêt est actif. Les primitives sont mentionnées sous la forme habituelle *i/a* avec *i* son identificateur d'accès et *a* son arité.

6.6.5. Commandes du mode interactif

A chaque point d'arrêt, le debugger imprime la séquence d'invite "DBG:" pour indiquer son attente d'une commande utilisateur. La frappe d'un retour chariot seul, noté RC dans ce chapitre, correspond à la commande par défaut *step* du mode interactif. Les autres commandes se terminent par un retour chariot.

Les commandes disponibles sont:

RC (step) Continue jusqu'au prochain appel de règle.

a (abort program) Génère une interruption utilisateur.

b (breakpoints) Affiche les points d'arrêt actifs. Voir aussi le prédicat *show_spy*.

e (end debugging and go) Provoque la terminaison du mode debug et poursuit l'exécution.

E (Execute) Empile une session Prolog.

f (fail) Provoque un échec sur le but affiché.

g (go to spy) Continue sans impression jusqu'au prochain point d'arrêt fixe.

G entier
(Go to end) Continue jusqu'à la fin du programme en imprimant seulement les points d'arrêt fixes si *entier* vaut 0. Si *entier* est différent de 0, imprime aussi les buts figurant dans la queue des points d'arrêt fixes.

h (help) Liste les commandes disponibles.

i entier
(indent) La valeur de *entier* définit la marge, en nombre de caractères pour l'impression.

j
(jump) Donne la liste des règles dont on ignore le corps, en mode *trace*.

l entier
(length) La valeur de *entier* définit la longueur de la ligne pour les sorties.

m entier
(mode) La valeur de *entier* permet de choisir le mode d'affichage des buts Prolog:

0	Afficher sous la forme: identificateur/arité.
1	Affichage par le programme de dessin d'arbres.
[2, 1000]	Affichage en profondeur limité par *entier*.
> 1000	Affichage complet du terme.

n (next) Arrêt sur le prochain but qui n'est pas un sous but.

N (Next while tracing) Arrêt sur le prochain but qui n'est pas un sous but, en imprimant les informations de l'exécution de la portion de programme, selon les options courantes choisies. L'effet de la commande > est ignoré dans ce cas.

p booléen
(print before unification) L'impression des buts avant unification est activée si *booléen* vaut 1 ou désactivée si *booléen* vaut 0. Si *booléen* n'est pas mentionné, il équivaut à 0.

P booléen
(Print after unification) L'impression des buts après unification est activée si *booléen* vaut 1 ou désactivée si *booléen* vaut 0. Si *booléen* n'est pas mentionné, il équivaut à 0.

q (quit) Quitte Prolog, et retourne au niveau système hôte.

s (stack) Affiche la pile Prolog des règles en cours d'exécution.

s 0 Permet d'éliminer l'impression du message et de la pile Prolog lors d'une erreur. *s 1* la rétablit.

S (State) Affiche les options courantes.

t (trace to spy) Continue jusqu'au prochain point d'arrêt fixe, en imprimant les informations de l'exécution de la portion de programme, selon les options courantes choisies.

T (Trace to end) Continue jusqu'à la fin du programme, en imprimant les informations d'exécution, selon les options courantes choisies.

w (where) Affiche la position du but courant dans le programme.

\+ *ident/arité*
Met un point d'arrêt sur le paquet de règles d'identificateur d'accès *ident* et d'arité *arité*. Voir aussi le prédicat *spy*.

\- *ident/arité*
Supprime tout point d'arrêt sur le paquet de règles d'identificateur d'accès *ident* et d'arité *arité*. Voir aussi le prédicat *no_spy*.

\- *a* Permet de supprimer tous les points d'arrêt en une seule commande.

\> *ident/arité*
Indique que le prédicat d'identificateur d'accès *ident* et d'arité *arité* ne devra pas être développé en trace.

< *ident/arité*
Annule l'effet de la commande inverse (>) sur le prédicat d'identificateur d'accès *ident* et d'arité *arité*.

< *a* Permet d'annuler l'effet de toutes les commandes inverses (>) qui ont eu lieu sur des prédicats.

Les points d'arrêt fixes, les prédicats à ne pas développer en trace, le mode d'impression et la définition de la marge, sont permanents. Ils ne sont pas modifiés par une activation spécifique ou par défaut de l'outil. Ils ne peuvent être modifiés ou supprimés qu'explicitement.

6.6.6. Exemple

A titre d'illustration, intéressons nous au programme d'exemple des menus livré dans le kit. Les commentaires de l'exécution sont donnés en italique.

On vérifie les règles chargées, puis on se place en mode debug. Enfin, on lance le but `dif(d, Sorbet_aux_poires) balanced_meal( Truffes_sous_le_sel, p, d).`

Ce qui est en gras est entré par l'utilisateur. Les lignes données par l'utilisateur doivent être terminées par un retour-chariot que nous n'indiquons pas, sauf dans le cas d'une ligne vide notée RC.

```
> dictionary;
DICTIONARY CONTENT OF ""
balanced_meal / 3
calories / 2
dessert / 1
fish / 1
hors_d_oeuvre / 1
main_course / 1
meal / 3
meat / 1
smaller / 2
sumof / 4
value / 4
{}
> debug  dif(d,  Sorbet_aux_poires)
balanced_meal(Truffes_sous_le_sel,p,d);
CALL: dif( v254, Sorbet_aux_poires)
DBG: RC
CALL: balanced_meal( Truffes_sous_le_sel, v623, v743)
On ajoute à la configuration par défaut l'impression après
unification et une indentation.
DBG: P1
DBG: i5
On vérifie l'état des options.
     DBG: S
     Print after unification
     Print before unification
     Print mode = out, depth: 4
     Errors are printed
     Indentation= 5
On poursuit.
     DBG: RC
     CALL: meal( Truffes_sous_le_sel, v623, v743)
C'est un prédicat important, on y met un point d'arrêt fixe
et on vérifie les points d'arrêt fixes actifs.
     DBG: +meal/3
     DBG: b
     meal/3
On poursuit.
     DBG: RC
     meal( Truffes_sous_le_sel, v623, v743)
     CALL: hors_d_oeuvre( Truffes_sous_le_sel)
     DBG: RC
     hors_d_oeuvre( Truffes_sous_le_sel)
     CALL: main_course( v623)
C'est un prédicat important, on y met un point d'arrêt fixe
et on vérifie les points d'arrêt fixes actifs.
     DBG: +main_course/1
     DBG: b
     main_course/1
```

```
     meal/3
On poursuit.
     DBG: RC
     main_course( v623)
     CALL: meat( v623)
     DBG: RC
     meat( Grillade_de_boeuf)
     CALL: dessert( v743)
     DBG: RC
     RECALL(2): dessert( v743)
     DBG: RC
     dessert( Fraises_chantilly)
     CALL: value( Truffes_sous_le_sel, Grillade_de_boeuf,
Fraises_chantilly, v817)
On ne veut pas s'arrêter dans ce prédicat, mais voir ce qui
s'y passe.
     DBG: N
     value( Truffes_sous_le_sel, Grillade_de_boeuf,
Fraises_chantilly, v817)
     CALL: calories( Truffes_sous_le_sel, v828)
     calories( Truffes_sous_le_sel, 212)
     CALL: calories( Grillade_de_boeuf, v833)
     calories( Grillade_de_boeuf, 532)
     CALL: calories( Fraises_chantilly, v838)
     calories( Fraises_chantilly, 289)
     CALL: sumof( 212, 532, 289, v817)
Les expressions arithmétiques ont été optimisées à la
compilation.
     add
     add
     sumof( 212, 532, 289, 1033)
     CALL: smaller( 1033, 800)
     DBG: N
     inf
     RECALL(3): dessert( v743)
     DBG: RC
     dessert( Melon_en_surprise)
     CALL: value( Truffes_sous_le_sel, Grillade_de_boeuf,
Melon_en_surprise, v817)
On vérifie la position du prédicat.
     DBG: w
     --->  balanced_meal / 3 , rule number 1, goal number 2
in queue
Pour la suite, le détail du prédicat n'est pas intéressant,
on supprime son développement et on vérifie la liste des
prédicats dans ce cas.
     DBG: >value/4
     DBG: j
     value/4
On poursuit.
     DBG: n
     value( Truffes_sous_le_sel, Grillade_de_boeuf,
Melon_en_surprise, 866)
     CALL: smaller( 866, 800)
     DBG: >smaller/2
     DBG: j
     smaller/2
     value/4
     DBG: n
     RECALL(2): dessert( v743)
```

```
On souhaite avoir la trace jusqu'au prochain point d'arrêt
avec seulement l'impression après unification, on désactive
donc l'impression avant unification.
     DBG: p0
     DBG: t
     dessert( Fraises_chantilly)
     value( Truffes_sous_le_sel, Poulet_au_tilleul,
Fraises_chantilly, 901)
     RECALL(3): dessert( Melon_en_surprise)
     value( Truffes_sous_le_sel, Poulet_au_tilleul,
Melon_en_surprise, 734)
     smaller( 734, 800)
{d=Melon_en_surprise,p=Poulet_au_tilleul}
     RECALL(2): main_course(v165)
.    DBG: p1
On se trouve arrêté sur un point d'arrêt fixe, on s'intéresse
aux prédicats importants (ils ont un point d'arrêt fixe)
jusqu'à la fin.
     DBG: +meat/1
     DBG: +fish/1
     DBG: b
     fish/1
     main_course/1
     meal/3
     meat/1
     DBG: G
     main_course( v623)
     CALL: fish( v623)
     fish( Bar_aux_algues)
{d=Fraises_chantilly,p=Bar_aux_algues}
{d=Melon_en_surprise,p=Bar_aux_algues}
     RECALL(2): fish( v623)
     fish( Chapon_farci)
{d=Fraises_chantilly,p=Chapon_farci}
{d=Melon_en_surprise,p=Chapon_farci}
>
```

6.7. Modification et visualisation de l'état courant

alloc
alloc/12

Décrit l'état des piles Prolog. La version sans argument écrit les valeurs courantes sur la sortie courante. Les arguments s'il sont présents, sont unifiés respectivement avec l'espace occupé et l'espace libre de:
1,2: espace du code,
3,4: pile de copie des structures (heap),
5,6: pile des environnements et variables locales (stack),
7,8: pile de restauration (trail),
9,10: espace du dictionnaire,
11,12: aucun espace.

edinburgh

Permet de passer en mode Edinburgh lorsque l'on est en mode Prolog II : change de syntaxe, ajoute des règles prédéfinies, ajoute et modifie des opérateurs. Cette primitive provoque le chargement du module *edinburg.mo*.

get_option(o,x)

Permet de connaître la valeur d'une option de comportement dans la session. *o* doit être une chaîne d'un caractère représentant l'option. *x* est unifié avec une chaîne d'un caractère représentant la valeur de l'option. La signification des caractères est celle définie par les conventions de l'option de la ligne de commande *-f* (cf. §U2.3). Par exemple pour connaître la syntaxe des réels choisie pour la session:

```
> get_option("r",x);
{x="P"}
```

prologII

Permet de passer en mode Prolog II lorsque l'on est en mode Edinburgh : change de syntaxe, modifie les opérateurs, supprime les règles prédéfinies spécifiques Edinburgh.

prologIIE

Permet de passer en mode Prolog II lorsque l'on est en mode Edinburgh, en gardant les règles prédéfinies spécifiques Edinburgh : change de syntaxe, modifie les opérateurs.

state

Affiche l'état courant des piles, et le nom des modules se trouvant en mémoire.

statistics
statistics/12

Permet de visualiser les valeurs maximales atteintes durant une session dans les espaces Prolog. La mémorisation de ces valeurs est effectuée en mode *debug* uniquement. Si *debug* n'a jamais été effacé avant l'appel à *statistics*, les valeurs affichées ne sont pas significatives. Les arguments ont la même signification que pour le prédicat *alloc*.

set_options(o)

Permet de changer les options ayant pu être définies au lancement par l'option *-f* de la ligne de commande, sans sortir de Prolog. *o* est une chaîne de caractères. Exemple pour passer en syntaxe de variable Edinburgh et en syntaxe de réels standard:

```
> set_options("vErS");
```

version(o)

Permet d'obtenir la version courante de Prolog. Par exemple, si la version courante est la version 3.1,

```
> version(o);
{o=301}
```

6.8. Gestion automatique des espaces et des piles

Le fonctionnement d'une machine suppose des piles, pour gérer l'exécution, et des espaces pour stocker les structures du programme.

Le système Prolog II+, qui est une machine virtuelle, possède trois piles :

- pile de récursion (stack) : qui contient les environnements successifs des appels de la démonstration en cours;
- pile de copie (heap) : qui contient une copie des structures de la démonstration en cours;
- pile de restauration (trail) : qui contient la mémorisation des valeurs des arguments, pour le backtracking;

et deux espaces pour les structures internes :

- espace pour le code : qui contient toutes les règles compilées et les données statiques de tous les modules;
- espace pour le dictionnaire : qui contient tous les identificateurs et les accès aux règles créés durant la session.

Le système Prolog II+ comporte un mécanisme de récupération et tassage des piles et des espaces, ainsi qu'un mécanisme de réallocation.

Pendant les opérations qui consomment les piles ou les espaces (*assert, insert, def_array,* récursivité non terminale…), ces piles ou espaces peuvent arriver à saturation. Le mécanisme de récupération d'espace et de tassage est alors déclenché automatiquement. Si malgré cela, la place libre restant dans la pile ou l'espace est toujours insuffisante, le mécanisme de réallocation est à son tour déclenché automatiquement. Il est chargé de réallouer la pile ou l'espace en cause, avec une taille supérieure de 25% à ce qu'elle était. Ce pourcentage est celui par défaut. Il est possible de le modifier à l'aide d'une option sur la ligne de commande au démarrage de Prolog (cf U2.3).

Il est possible de déclencher explicitement un des deux mécanismes sur une pile ou un espace, à l'aide des prédicats *gc* et *realloc*.

gc(x)

déclenche la récupération et le tassage de l'espace désigné par *x*. *x* peut prendre les valeurs suivantes:

:code pour l'espace du code,
:dictionary pour l'espace du dictionnaire,
:stacks pour les piles (qui ne peuvent pas être dissociées pour cette opération).

N.B. : Lorsque le récupérateur de l'espace du code est activé, toutes les règles en cours d'utilisation dans l'effacement courant sont considérées comme devant être conservées, même si elles ont été précédemment supprimées et sont donc inaccessibles pour une nouvelle exécution.

N.B. : Le récupérateur d'espace doit déterminer les objets inutiles pour pouvoir réutiliser leur place. Les structures succeptibles d'être manipulées depuis un langage externe et sur lesquelles le récupérateur d'espace peut agir, sont les identificateurs Prolog. Pour permettre à Prolog de savoir si un identificateur est toujours utile et pour éviter à l'utilisateur de continuer à manipuler une représentation éventuellement obsolète, l'utilisateur doit préciser à Prolog s'il utilise ou pas un identificateur particulier depuis l'extérieur, afin que le récupérateur d'espace ne le supprime pas. Ceci peut être fait grâce aux fonctions *set_permanent_symbol* et *reset_permanent_symbol* (cf. Chapitre 7 de ce manuel).

realloc(x,y)

déclenche la réallocation de l'espace désigné par *x*, d'une taille réajustée par le coefficient *y*. *x* peut prendre les valeurs *:code*, *:dictionary*, *:stack*, *:heap*, *trail*, selon qu'il s'agisse respectivement : du code, du dictionnaire, de la pile de récursion, de la pile de copie ou de la pile de restauration. *y* est un entier, il indique le pourcentage d'augmentation (s'il est positif) ou de diminution (s'il est négatif) de la taille précédente.

7. Extensions de Prolog avec des langages externes.

7.1. Principes des fonctions de communication de données
7.2. Fonctions de communication de données simples
7.3. Fonctions de communication de termes quelconques
7.4. Méthode des descripteurs
7.5. Données partagées
7.6. Ajout de fonctions externes
7.7. Ajout de fonctions externes à appel direct

Ce chapitre montre comment faire des liens avec des données externes ou comment ajouter de nouvelles règles prédéfinies écrites en C ou tout autre langage compatible avec C. Les exemples qui y sont décrits, se trouvent déjà dans le fichier *expredef.c* du kit. Un équivalent pour Fortran est donné dans le fichier *fprouser.eg* du kit.

Pour étendre Prolog avec des fonctions d'un langage externe, il est nécessaire de disposer d'un module de communication pour:

- créer un lien entre le prédicat prolog et la fonction externe,
- faire la transformation des données échangées, entre la structure Prolog et la structure externe.

Différentes méthodes sont proposées pour réaliser ces deux traitements:

- le lien se définit en C: c'est la méthode des descripteurs décrite dans ce chapitre,
- le lien se définit en Prolog: c'est la méthode des parasites exposée en annexe D,
- la transformation des données se fait en Prolog: c'est la méthode des descripteurs concernant les fonctions à appel direct, c'est à dire qui ne nécessitent pas d'interface C pour le passage d'arguments,
- la transformation des données se fait en C ou en Fortran: dans les autres cas.

Pour étendre Prolog avec certains types de données, qui seront alors partagées entre Prolog et C, il suffira de:

- créer un lien entre le terme Prolog et la donnée externe, par la méthode des descripteurs.

Nous débuterons ce chapitre par la description des fonctions de communication de termes, dans le cas où la récupération et l'affectation de la valeur des termes se font en C ou en Fortran. En effet ces fonctions sont utiles dans de nombreux cas. Nous exposerons ensuite, comment faire le lien avec des objets externes par la méthode des descripteurs. Nous terminerons en détaillant les possibilités offertes par cette méthode.

7.1. Principes des fonctions de communication de données

Nous allons décrire dans les paragraphes 7.2. et 7.3. suivants, les fonctions de communication qui sont utilisables lorsque dans une procédure d'un langage externe, il est nécessaire de connaître la valeur d'un argument du prédicat Prolog lié à la procédure, ou bien de lui affecter une valeur.

Nous donnons ici quelques principes de fonctionnement communs à toutes ces fonctions.

Etant donné les différences de conventions entre langages en ce qui concerne le passage d'argument, un jeu de fonctions est fourni pour le langage C, et un jeu équivalent est fourni pour le langage FORTRAN. Les routines FORTRAN ont les mêmes arguments, mais passés par adresse. Leur nom commence par la lettre "f" et ne contient pas de signe "_". Les routines FORTRAN peuvent également être utilisées pour des programmes PASCAL en déclarant tous les arguments en *var*.

Dans tous les cas, une donnée doit être transmise entre Prolog et un langage externe. Par conséquent, à l'appel de la fonction de communication, il est nécessaire de préciser **le rang** de l'argument du prédicat Prolog, et **la structure** de la donnée externe associée. Puis, selon le sens de communication demandé, il s'agira de "copier" l'argument Prolog dans la structure externe choisie, ou bien il s'agira de construire dans les piles de Prolog le terme représenté par la structure externe et de l'**unifier** avec l'argument du prédicat associé.

D'autre part, il peut arriver que la communication ne puisse pas se faire. Il est alors nécessaire d'en être informé. Les fonctions de communication ont pour cela un argument qui sert d'**indicateur**. Il est nul quand la fonction a pu être réalisée, strictement positif quand une **erreur** doit être générée, et égal à -1 quand un **backtracking** doit être provoqué.

Le système de gestion des erreurs de Prolog impose que le programme externe soit immédiatement quitté dès qu'une erreur ou un backtracking est notifié, autrement dit dès que l'indicateur devient non nul. En effet quand l'indicateur est non nul, cela reflète une situation anormale qui nécessite dans le programme Prolog un backtracking ou une erreur, qui pourront être réalisés seulement lorsque la machine Prolog sera relancée, c'est à dire lorsque le programme externe se terminera. Il peut s'agir également d'une erreur plus grave, ayant pour conséquence l'arrêt de la machine Prolog.

7.2. Fonctions de communication de données simples

Les fonctions décrites ici conviennent dans les cas simples où le type des données à transférer est connu au moment de l'appel et correspond à un type du langage externe, typiquement les entiers, les réels et les chaînes de caractères. Pour les autres cas, d'autres fonctions de communication plus générales existent, elles sont décrites au § 7.3.

Chaque type de données a une fonction[1] qui lui est associée pour transférer cette donnée depuis Prolog vers le programme externe, et une autre pour la transférer depuis la fonction externe vers Prolog.

7.2.1. Test du type d'un argument

La fonction *get_arg_type* permet de tester le type d'un argument Prolog, afin de pouvoir choisir la procédure de communication appropriée pour récupérer sa valeur. Cette fonction permet également de connaître la taille en octets qu'occuperait l'argument. Ceci est très utile en particulier pour connaître la taille à allouer pour récupérer une chaîne. La fonction renvoie une erreur uniquement lorsqu'il n'existe pas d'argument du rang demandé.

```
get_arg_type(no_arg, value, lgPtr, err)
int   no_arg;
char  *value;
int   *lgPtr;
int   *err;

integer*4 function fgetargtype(no_arg, value, lgPtr, err)
integer*4   no_arg,lgPtr,err;
character** value;
```

no_arg

Entier donnant le rang de l'argument choisi dans le prédicat Prolog. Le premier argument a le rang 1, le second a le rang 2 et ainsi de suite.

---

[1] Des fonctions supplémentaires existent pour le transfert de chaînes. Voir l'annexe E.

value

Adresse d'une variable de type caractère dans laquelle sera écrit le "code" du type de l'argument. La signification des caractères retournés est la suivante:

"code"	type
'I'	entier
'R'	réel
'S'	chaîne
'N'	identificateur
'E'	*nil*
'V'	variable libre
'D'	liste
'T'	n-uplet

Le codage des types est le même que celui choisi pour les procédures *get_term*, *put_term* (cf. §7.2.1.), excepté pour l'identificateur *nil* qui est repéré par le 'E'.

lgPtr

Adresse d'une variable entière dans laquelle sera écrite la taille en octets occupée par l'argument. Lorsque l'argument n'est pas un terme simple, autrement dit lorsque c'est une liste ou un n-uplet, la taille est non spécifiée.

err

La variable pointée est différente de 0 si une erreur s'est produite, égale à 0 sinon.

La fonction retourne la valeur correspondant à `*err`.

7.2.2. Transfert de données simples de Prolog vers un autre langage.

Ces fonctions sont appelées par le programme externe pour obtenir les valeurs effectives des arguments du prédicat Prolog.

Si le type de l'argument effectif n'est pas celui attendu, ou bien s'il n'existe pas d'argument du rang demandé, la fonction de communication notifie une erreur en affectant à l'indicateur *err* une valeur non nulle. Si, pour un argument de type entier, la valeur ne peut être représentée dans le type externe associé, une erreur est notifiée de la même manière.

> ***Attention**: Pour un comportement correct du système de gestion des erreurs, on doit immédiatement sortir du programme externe si la variable représentée par* err *est non nulle.*

Voici les fonctions disponibles pour les types simples:

Interface C:

```
int get_integer(no_arg, value, err)
int   no_arg;
long  *value;
int   *err;

int get_real(no_arg, value, err)
int   no_arg;
float *value;
int   *err;

int get_double(no_arg, value, err)
int   no_arg;
double *value;
int   *err;

int get_string(no_arg, value, err)
int   no_arg;
char  *value;
int   *err;

int get_max_string(no_arg, lg_max, value, in_external_code,
err)
int   no_arg;
int   lg_max;
char  *value;
int   in_external_code;
int   *err;
```

Interface Fortran:

```
integer*4 function fgetinteger(no_arg, value, err)
integer*4 no_arg,value,err

integer*4 function fgetreal(no_arg, value, err)
integer*4 no_arg,err
real*4 value

integer*4 function fgetdouble(no_arg, value, err)
integer*4 no_arg,err
real*8 value

integer*4 function fgetstring(no_arg, lg, value, err)
integer*4 no_arg,lg,err
character** value

integer*4 function fgetmaxstring(no_arg, lg_max, lg, value,
in_external_code, err)
integer*4 no_arg,lg_max,lg,in_external_code,err
character** value
```

no_arg

Entier donnant le rang de l'argument choisi dans le prédicat Prolog. Le premier argument a le rang 1, le second a le rang 2 et ainsi de suite.

value

Adresse de la donnée qui doit recevoir la valeur de l'argument de rang *no_arg* dans le prédicat Prolog.

lg

Entier, utilisé dans les fonctions Fortran qui manipulent des chaînes, qui doit recevoir la longueur effective de la chaîne de caractères.

lg_max

est la taille de la zone de caractères.

in_external_code

est un booléen qui indique si la chaîne doit être en code caractère externe (ceci est utile lorsque l'option caractères ISO est activée et que l'on utilise des caractères non présents dans la première moitié du jeu ISO; voir Annexe E). Si la valeur est 0 aucune transformation n'est faite sur la chaîne Prolog.

err

La variable pointée est différente de 0 si une erreur ou un *backtracking* a été demandé.

Ces fonctions retournent zéro lorsqu'une erreur s'est produite ou une valeur non nulle lorsqu'il n'y a pas eu d'erreur.

get_max_string copie la chaîne de caractères originale (depuis la mémoire de travail de Prolog) dans une zone définie dans le programme externe, pointée par *value*, d'une taille de *lg_max* caractères. Un caractère nul indique la fin de la chaîne. Une erreur est provoquée si la taille de la zone de caractères définie par *lg_max* est insuffisante pour la chaîne à récupérer.

get_string fait la même chose que *get_max_string* mais ne fait pas de test de débordement. Le tableau pointé par *value* doit donc avoir une taille suffisante pour contenir la chaîne. Pour éviter les risques d'écrasement mémoire, il est préférable d'utiliser *get_max_string*.

7.2.3. Transfert de données simples d'un langage externe vers Prolog

Ces fonctions sont appelées par le programme externe pour unifier une valeur avec un argument du prédicat Prolog associé. Si l'unification échoue, un *backtracking* est annoncé par la fonction de communication.

Si le type de l'argument effectif n'est pas celui attendu, ou bien s'il n'existe pas d'argument du rang demandé, la fonction de communication notifie une erreur en affectant à l'indicateur *err* une valeur non nulle.

***Attention**: Pour un comportement correct du système de gestion des erreurs, on doit immédiatement sortir du programme externe si la variable représentée par* err *est non nulle.*

Voici les fonctions disponibles pour les types simples:

Interface C:

```
int put_integer(no_arg, value, err)
int   no_arg;
long  value;
int   *err;

int put_real(no_arg, value, err)
int   no_arg;
float value;
int   *err;

int put_double(no_arg, value, err)
int   no_arg;
double value;
int   *err;

int put_string(no_arg, value, err)
int   no_arg;
char  *value;
int   *err;
```

Interface Fortran:

```
integer*4 function fputinteger(no_arg, value, err)
integer*4 no_arg,value,err

integer*4 function fputreal(no_arg, value, err)
integer*4 no_arg,err
real*4 value

integer*4 function fputdouble(no_arg, value, err)
integer*4 no_arg,err
real*8 value

integer*4 function fputstring(no_arg, lg, value, err)
integer*4 no_arg,lg,err
character** value
```

no_arg

Entier donnant le rang de l'argument choisi dans le prédicat Prolog. Le premier argument a le rang 1, le second a le rang 2 et ainsi de suite.

value

Valeur qui sera unifiée sur l'argument de rang *no_arg* dans le prédicat Prolog associé. Pour le transfert de données de type chaîne de caractères, *value* est l'adresse d'une chaîne de caractères terminée par zéro, définie dans le programme externe. La fonction *put_string* copie alors cette chaîne de caractères dans la mémoire de travail de Prolog avant d'unifier la valeur avec l'argument du prédicat associé.

err

La variable pointée est différente de 0 si une erreur ou un *backtracking* a été demandé.

Ces fonctions retournent zéro lorsqu'une erreur s'est produite ou une valeur non nulle lorsqu'il n'y a pas eu d'erreur.

7.3. Fonctions de communication de termes quelconques

Les termes dont il est question dans ce chapitre, sont des termes qui ne représentent pas des arbres infinis, et qui peuvent être d'un type Prolog quelconque. Par exemple, une liste de n-uplets contenant des variables est un tel terme.

Pour pouvoir communiquer ces termes, et en particulier des types de données qui sont propres à Prolog, il faut disposer d'une codification de ces données pour les identifier hors Prolog.

Pour cela, Prolog permet de choisir entre:

- des chaînes de caractères, qui existent dans tous les langages, mais qui nécessitent l'utilisation d'un analyseur pour identifier les termes représentés.

- une structure arborescente d'éléments (type, valeur) codée dans des tableaux de données communes à tous les langages. Pour identifier les termes représentés, il s'agira de parcourir cette structure.

7.3.1. Au moyen de chaînes de caractères

Les fonctions décrites ici sont simples à employer et faciles à mettre en œuvre. Elles peuvent être choisies par exemple pour une première réalisation d'interfaçage de Prolog avec des langages externes.

Elles sont beaucoup moins efficaces que la solution proposée avec les structures de tableaux qui sera décrite au paragraphe suivant. Elles offrent moins de possibilités d'exploitation des arguments. Elles comportent une restriction sur la manipulation des variables: il n'est pas possible de voir ou d'imposer que plusieurs arguments du prédicat lié à la procédure externe, utilisent la même variable libre. En effet le dictionnaire des variables est local à la procédure de communication, c'est à dire local à un argument (et non pas local à la règle).

Interface C:

```
get_strterm(no_arg, lg_max, value, in_external_code, err)
int   no_arg;
int   lg_max;
char  *value;
int   in_external_code;
int   *err;


put_strterm(no_arg, value, in_external_code, err)
int   no_arg;
char  *value;
int   in_external_code;
int   *err;
```

Interface Fortran:

```
integer*4 function fgetstrterm(no_arg, lg_max, lg, value,
in_external_code, err)
integer*4 no_arg,lg_max,lg,in_external_code,err
character** value

integer*4 function fputstrterm(no_arg, lg, value,
in_external_code, err)
integer*4 no_arg,lg_max,in_external_code,err
character** value
```

no_arg

est le rang de l'argument choisi dans le prédicat Prolog. Le premier argument a le rang 1, le second a le rang 2 et ainsi de suite. S'il n'existe pas d'argument de ce rang, une erreur est annoncée.

value

est l'adresse d'une zone de caractères.

lg_max

est la taille de la zone de caractères.

in_external_code

est un booléen qui indique si la chaîne est (*put_strterm*), ou doit être (*get_strterm*) en code caractère externe (ceci est utile lorsque l'option caractères ISO est activée et que l'on utilise des caractères non présents dans la première moitié du jeu ISO; voir Annexe E).

err

indique si une erreur s'est produite ou pas ou si un backtracking doit être généré.

Dans le cas du transfert de Prolog vers le langage externe, la zone de caractères pointée par *value* doit être suffisamment grande pour contenir la chaîne résultat; Prolog transforme (à la manière du prédicat `out`) l'argument de rang *no_arg* en une chaîne qu'il copie à l'adresse *value*. Si la chaîne à copier est plus grande que *lg_max*, une erreur est signifiée.

Dans le cas du transfert du langage externe vers Prolog, Prolog analyse la chaîne contenue dans *value*, construit le terme Prolog qu'elle représente (à la manière du prédicat `in`) et l'unifie avec l'argument de rang *no_arg*.

Par exemple, dans une règle externe, pour unifier le troisième argument avec une liste de 5 entiers, on écrira :

```
put_strterm( 3, "[1,2,3,4,5]", 0, &err);
```

Voyons ci-dessous un petit exemple d'utilisation, avec une partie en C à ajouter à Prolog et la partie Prolog qui le teste. On suppose que le lien entre le prédicat Prolog et la fonction C a été déclaré.

```
int test_ccom()
   { int err; char s[80];
   get_strterm(1,80,s,1,&err);
   if (err) return err;
   fprintf(stderr,">>>>%s\n",s);
   put_strterm(2,s,1,&err);
   if (err) return err;
   return 0;
   }

> test_ccom(term(1.2.x,"string",[]),z);
>>>>term(1.2.v64,"string",nil)
{z=term(1.2.v150,"string",nil)}
>
```

7.3.2. Au moyen de structures de tableaux

Les deux fonctions de communication de termes, *get_term* et *put_term,* décrites ici sont les fonctions de communication de données, les plus générales.

Sachant que les fonctions de communication de termes simples sont typées, cette interface peut servir également dans le cas où le type de l'argument à traiter n'est pas connu à priori.

Dans le cas de la communication du langage externe vers Prolog, il s'agit d'unifier le terme proposé avec l'argument du prédicat Prolog associé. Si l'unification échoue, un *backtracking* est annoncé par la fonction de communication.

Les structures de Prolog sont transformées par ces procédures d'interface en une structure de tableaux facilement manipulable dans des langages algorithmiques typés comme Pascal ou C. Les structures obtenues peuvent être conservées d'une exécution à une autre et peuvent être écrites ou chargées à partir d'un fichier. Pour cela les termes de Prolog sont éclatés dans quatre tableaux:

Deux tableaux pour les constantes:
- un tableau de caractères pour les chaînes.
- un tableau de doubles pour les réels.

Deux tableaux pour coder respectivement le type et la valeur d'un terme et de ses sous-termes.
- un tableau de caractères pour les types (ou 'tags') de terme/sous-termes.
- un tableau d'entiers longs pour les valeurs de terme/sous-termes. La signification de chaque élément est donnée par le contenu de l'élément de même indice dans le tableau de types.

Le terme correspondant à l'argument *no_arg* est décrit par la première case des tableaux de types et de valeurs de terme/sous-termes:
tag_tab[0], val_tab[0]

7.3.2.1. Description du codage d'un terme

Un terme est représenté par un tag et une valeur entière dont l'interprétation dépend du tag.

terme:	**(tag,**	**valeur)**
Entier:	'I'	la valeur de l'entier. Si l'entier venant de Prolog est trop grand pour être codé, une erreur est notifiée.
Réel:	'R'	un index dans le tableau des réels doubles *real_tab*.
Réel:	'X'	un index dans le tableau des réels doubles *real_tab*.
Chaîne:	'S'	un index *i* dans le tableau des caractères *str_tab*. » *i* indique le premier caractère de la chaîne. La chaîne est terminée par un caractère NULL.
Ident:	'N'	un entier représentant l'identificateur de manière unique. » Cette représentation est unique tant que l'identificateur est déclaré utilisé (cf. *set_permanent_symbol*). Sinon il peut être supprimé par le récupérateur de mémoire.
variable:	'V'	un numéro de variable libre. » Le numéro est unique pour un état donné de la machine (voir § 7.3). Si deux numéros sont différents, il s'agit de deux variables libres différentes (c'est à dire non liées entre elles). Dans un même appel de procédure externe, ces numéros sont cohérents pour différents appels des fonctions *get_term* et *put_term*. Entre deux activations ou deux états, il n'y a aucun lien entre ces numéros. De nouvelles variables peuvent être créées en appelant *put_term* avec des numéros non encore utilisés. Il est plus efficace de créer des numéros en ordre croissant par rapport aux numéros déjà attribués. Prolog attribue ces numéros par incréments de 1.
séquence:	'D'	un index dans les tableaux de terme/sous-termes vers une séquence de sous-termes. » La séquence pointée est terminée par 'E' ou 'F' suivant le cas.

terme:	**(tag,**	**valeur)**
n-uplet:	'T'	un index dans les tableaux de terme/sous-termes *tag_tab* et *val_tab*. » Pointe sur une suite de termes dont le premier est un entier représentant le nombre d'arguments *n* du n-uplet, suivi des *n* sous-termes représentant les arguments.
fin liste:	'E'	indéfinie. » Marque la fin d'une séquence se terminant par *nil*. Cette marque peut être le premier élément d'une séquence pointée par 'D'.
fin séquence:	'F'	indéfinie. » Marque la fin d'une séquence ne se terminant pas par *nil*: le terme qui suit immédiatement cette marque est le dernier élément de la séquence. Cette marque ne peut être le premier élément d'une séquence pointée par 'D'.

On a deux représentations équivalentes du terme *1.2.3.nil*, l'une sous forme vectorisée, la deuxième étant de type paire pointée (ici une paire est représentée par 3 entrées). Prolog fournit toujours les données sous la première forme qui est plus compacte, la deuxième étant surtout utile pour construire des listes dont la longueur n'est pas connue au début de la construction.

Le couple ('N', 0) représente par convention l'identificateur *nil.* (dans ce qui suit, "-" représente des valeurs indéfinies)

	forme compacte			forme type paire pointée		
00	'D'	1	⇔	'D'	1	terme: 1.2.3.nil
01	'I'	1		'I'	1	
02	'I'	2		'F'	-	
03	'I'	3		'D'	4	
04	'E'	-		'I'	2	
05				'F'	-	
06				'D'	7	
07				'I'	3	
08				'F'	-	
09				'N'	0	

exemple:

Codage du terme *1.2.(3."Hi!".x).father(x,y).nil*

indice	tag_tab	val_tab (32 bits)	
00	'D'	1	argument de type séquence
			* codage des sous-termes de l'arg.
01	'I'	1	
02	'I'	2	...
03	'D'	10	vers: (3."Hi!".x)
04	'T'	6	vers: father(x,y)
05	'E'	-	nil
			* début sous-terme: father(x,y)
06	'I'	3	
07	'N'	<entier représentant father>	
08	'V'	0	x
09	'V'	1	y
			* début sous-terme: (3."Hi!".x)
10	'I'	3	
11	'S'	0	référence vers 1er élt de str_tab
12	'F'	-	
13	'V'	0	x
14	-	-	

Tableau des caractères		Tableau des réels	
indice	caractère	indice	réel
00	'H'	00	-
01	'i'	01	-
02	'!'		
03	null (0)		
04	-		

7.3.2.2. Identificateurs

Dans la structure de tableaux, un identificateur est codé par une clé (notamment un entier) dans le dictionnaire de Prolog. Au moment de la transformation des données par les fonctions d'interfaçage (*get_term*, *put_term*), des fonctions de conversions entre la clé et la représentation de l'identificateur (chaîne de caractères) peuvent être nécessaires.

La fonction *pro_symbol*, ou son équivalent en Fortran, permet à partir d'une chaîne de caractères, de construire l'identificateur Prolog correspondant (identificateur qui pourra par exemple faire partie d'un terme qui sera transmis à Prolog par la procédure *put_term*). La chaîne de caractères doit avoir la syntaxe d'un identificateur. Si la chaîne de caractères ne comprend pas de préfixe (c'est une notation abrégée), l'identificateur créé sera fonction du contexte courant (Voir à ce propos le Chapitre 3 de ce manuel). Si une erreur s'est produite, la fonction retourne la valeur 0.

```
long pro_symbol(str)
char *str;

integer*8 function fprosymbol(lg, str)
integer*4 lg
character** str
```

La fonction *symbol_string*, ou son équivalent en Fortran, permet d'obtenir la représentation, sous forme de chaîne, d'un identificateur Prolog. Cet identificateur aura été obtenu par une des fonctions de communication de termes, *get_term* par exemple. Les arguments fournis à la fonction sont la clé `key`, l'adresse d'une zone mémoire où la chaîne pourra être copiée `str`, la longueur maximum de la chaîne de caractères copiée `lg_max`. La chaîne obtenue est la notation de l'identificateur en fonction du contexte courant, notation abrégée si possible. La fonction donnera dans le paramètre de sortie `lg`, la longueur effective de la chaîne. La fonction retourne 0 si elle a pu s'exécuter, un entier positif correspondant à un numéro d'erreur si une erreur s'est produite.
NB: `key` doit correspondre à un identificateur valide.

```
int symbol_string(key, str, lg, lg_max)
long key;
char *str;
int *lg;
int lg_max;

integer*4 function fsymbolstring(key, str, lg, lg_max)
integer*4 key,lg_max,lg
character** str
```

Le caractère choisi pour noter la séparation entre le préfixe et le suffixe de la représentation d'un identificateur complet, peut être modifié depuis Prolog. La fonction suivante permet donc de connaître le caractère courant utilisé :

```
char prefix_limit();

character*1 function fprefixlimit();
```

Les fonctions *set_permanent_symbol* et *reset_permanent_symbol*, ou leur équivalent en Fortran, permettent de prévenir le récupérateur de mémoire de Prolog, dans le cas où il serait activé sur le dictionnaire, qu'un identificateur est utilisé et respectivement non utilisé, par une procédure externe.
NB: `key` doit correspondre à un identificateur valide.

```
set_permanent_symbol(key)
long key;

reset_permanent_symbol(key)
long key;

integer*4 function fsetpermanentsymbol(key)
integer*4 key

integer*4 function fresetpermanentsymbol(key)
integer*4 key
```

7.3.2.3. Description des fonctions de communication

Les paramètres de ces fonctions sont:

- Le numéro de l'argument *no_arg*.

- La taille des tableaux: *tab_size*, *str_tab_size*, *real_tab_size*.

- Les tableaux utilisés pour coder le terme: *tag_tab*, *val_tab*, *str_tab*, *real_tab*. Lorsqu'un tableau n'est pas référencé (par exemple s'il n'y a ni chaînes, ni réels), il peut être remplacé par NULL. Une erreur est signifiée lorsqu'un des tableaux n'a pas une taille suffisante pour coder le terme.

- Un entier indiquant la dernière case utilisée pour coder le terme dans les tableaux *tag_tab* et *val_tab*: *max_used*. Dans l'exemple décrit plus haut, il vaut 13.

- L'indicateur *err*, indiquant si une erreur ou un backtracking doit être provoqué. Cet indicateur doit être testé à chaque appel, et il faut sortir de la procédure externe immédiatement s'il est non nul.

```
int get_term(no_arg, tab_size, str_tab_size, real_tab_size,
      tag_tab, val_tab, str_tab, real_tab, max_used,
      err );
   int no_arg;
   int tab_size, str_tab_size, real_tab_size;
   char tag_tab[];
   long val_tab[];
   char str_tab[];
   double real_array[];
   int * max_used;
   int *err;

int put_term(no_arg, tab_size, str_tab_size, real_tab_size,
      tag_tab, val_tab, str_tab, real_tab, max_used,
      err );
   int no_arg;
   int tab_size, str_tab_size, real_tab_size;
   char tag_tab[];
   long val_tab[];
   char str_tab[];
   double real_array[];
   int max_used;
   int *err;

integer*4 function fgetterm(no_arg, tabsize, strtabsize,
realtabsize,
      tagtab, valtab, strtab, realtab, maxused,
      err );
integer*4 function fputterm(no_arg, tabsize, strtabsize,
realtabsize,
      tagtab, valtab, strtab, realtab, maxused,
      err );
```

Des exemples d'utilisation de ces procédures d'interface sera trouvé dans le fichier *expredef.c* ou *fprouser.eg* du kit.

7.4. Principe de la méthode des descripteurs

La méthode des descripteurs permet de définir des objets relais Prolog et leur lien avec un objet externe. Elle s'applique pour les fonctions externes et les données partagées. Dans cette méthode, la déclaration des objets et des liens se fait en C.

Nous verrons d'abord quels sont les éléments à décrire en C pour réaliser cette définition et ensuite les deux manières de les déclarer: statiquement ou dynamiquement.

7.4.1. Eléments descriptifs

Pour pouvoir réaliser cette association, il est nécessaire de connaître:

- l'objet externe: dans un langage tel que C, cela se traduit par son adresse et son type.
- l'objet Prolog: c'est à dire son nom, son type et sa taille ou son arité selon qu'il s'agisse d'un tableau ou respectivement d'un prédicat.

Nous identifions donc les éléments nécessaires suivants:

name

identifie le terme Prolog associé à l'objet externe. C'est un pointeur vers une chaîne de type C contenant la représentation d'un identificateur Prolog complet.
Il a une signification différente dans le cas d'un objet de type SYMBOL_ARRAY. Il représente alors seulement le module auquel appartiennent les identificateurs du tableau. Il contient donc la représentation du préfixe de ces symboles.

type

définit le type de l'objet externe déclaré. Les valeurs possibles sont:
INT_ARRAY pour un tableau d'entiers, CHAR_ARRAY pour un tableau de caractères, STRING_ARRAY pour un tableau de chaînes, SINGLE_FLOAT_ARRAY pour un tableau de réels en simple précision, DOUBLE_ARRAY pour un tableau de réels en double précision, SYMBOL_ARRAY pour un tableau d'identificateurs, C_FUNCTION ou C_FUNCTION_PROTECTED pour une fonction externe, C_FUNCTION_BACKTRACK ou C_FUNCTION_BACKTRACK_PROTECTED pour une fonction externe non déterministe, DIRECT_C_FUNCTION pour une fonction externe à appel direct. Pour les tableaux de données, il est possible de leur ajouter la constante OFFSET_ZERO_BASED pour permettre en Prolog de les indexer à partir de zéro.

size

définit l'arité de la règle dans le cas d'une fonction externe, ou bien la taille du tableau s'il s'agit d'une donnée commune. N'est pas significatif dans le cas de SYMBOL_ARRAY.

adresse

Définit l'adresse effective de l'objet C déclaré (ou d'un sous-tableau de descripteurs pour SYMBOL_ARRAY).

7.4.2. Déclaration statique

La déclaration des objets externes et des objets Prolog associés, peut se faire statiquement par des tables de descripteurs qui sont parcourues par Prolog au moment de l'initialisation. Ces tables, au moment du chargement de Prolog, vont créer les liens et les déclarations Prolog nécessaires. Les objets Prolog ainsi créés et leur lien avec l'objet externe, sont permanents durant toute la session. En particulier, ils ne sont pas supprimés par les règles prédéfinies *kill_module*, *kill_array*, *suppress* …

Un descripteur de référence externe (voir fichier *proext.h*) est une structure composée de quatre champs qui contiennent les éléments nécessaires que nous avons identifiés:

```
typedef struct
    {
    char *name;
    int  type;
    int  size;
    POINTER adresse;
    } EXTERNAL_DESCRIPTOR;
```

Pour ajouter ces objets à Prolog, la machine Prolog doit être étendue par un ou plusieurs modules externes contenant la déclaration de ces descripteurs. Un descripteur doit être affecté à une table de descripteurs, qui doit être pointée par le tableau général PRO_EXTERNAL. Le programme *prolink* d'édition de liens, construit automatiquement ce tableau PRO_EXTERNAL dans le module *prodesc*, puis reconstruit la machine Prolog.

PRO_EXTERNAL est un tableau d'adresses de tables de descripteurs, il doit être terminé par une adresse nulle(0). Dans sa version initiale du kit, il contient la table du fichier *prouser*:

```
EXTERNAL_DESCRIPTOR *PRO_EXTERNAL[] = {
prouser_desc,
0 };
```

Table de descripteurs

Une table de descripteurs est un tableau de descripteurs, terminé par un descripteur contenant 0 dans le champ *name*. Par exemple:

```
EXTERNAL_DESCRIPTOR sample_desc[] =
 {
   {":term_vars", C_FUNCTION, 2, (POINTER) term_vars},
   {":qsort", DIRECT_C_FUNCTION, 1, (POINTER) quicksort},
   {":enumerate", C_FUNCTION_BACKTRACK,3,(POINTER)enumerate},
   { 0, 0, 0, 0}
 };
```

L'utilisateur peut créer autant de tables qu'il le désire; le plus naturel est d'en réaliser une par fichier externe.

7.4.3. Déclaration dynamique

La possibilité de créer des associations d'objets Prolog-C dynamiquement, est offerte grâce à la fonction PRO_BIND. Les objets externes doivent être connus de l'exécutable Prolog (c'est à dire que la machine Prolog doit être étendue par des modules externes contenant ces objets), par contre la création de l'objet Prolog et le lien avec cet objet sont faits à l'exécution de la fonction PRO_BIND.

Comme dans le cas de la déclaration statique, les objets Prolog ainsi créés et leur lien avec l'objet externe, sont permanents durant toute la session. Une fois la déclaration faite, il n'est pas possible d'en annuler l'effet.

La fonction PRO_BIND a 4 arguments, chaque argument représentant un élément nécessaire que nous avons identifié précédemment :

```
int PRO_BIND(name, type, size, adresse)
char *name;
int type, size;
void *adresse;
```

Par exemple, il est possible de créer dynamiquement une zone de données partagées. Dès que la fonction PRO_BIND a été appelée, le tableau est connu de Prolog.

L'exemple suivant alloue une zone de 1000 réels commune à Prolog et C, et accessible dans Prolog sous le nom *mymodule:data*

```
#include <malloc.h>
#include "proext.h"
...
double *t;

t = (double *) malloc( 1000*sizeof(double));
PRO_BIND( "mymodule:data"
        , DOUBLE_ARRAY
        , 1000
        , (POINTER) t);
```

à partir de l'exécution de cette séquence, le programme Prolog suivant s'exécute sans erreur, et référence la zone de données *t[4]*:

```
assign(mymodule:data(5), 1.5e );
```

Dans les paragraphes suivants qui exposent les différents types d'objets accessibles par la méthode des descripteurs, les exemples sont décrits avec des déclarations statiques. Ils peuvent évidemment aussi être déclarés de manière dynamique.

7.5. Données partagées

Les données qui peuvent être partagées entre Prolog et C sont des données d'un type manipulable par les deux langages. Les données qui s'y prêtent sont les entiers, les réels, les caractères, les chaînes de caractères et les identificateurs.

Prolog peut être étendu par une ou plusieurs zones de données partagées. Ces zones de données sont assimilées à des tableaux et sont manipulées comme des tableaux dans les deux langages.

La définition de ces données communes se fait par la déclaration d'un descripteur dont le champ *type* aura une des valeurs suivantes: INT_ARRAY, SINGLE_FLOAT_ARRAY, DOUBLE_ARRAY, CHAR_ARRAY, STRING_ARRAY, SYMBOL_ARRAY, ou bien une de ces valeurs augmentée de OFFSET_ZERO_BASED.

Le champ *name* sera alors la représentation Prolog du tableau. La primitive *def_array* ne doit pas être utilisée pour définir ce tableau. Le champ *size* indique le nombre d'éléments du tableau.

Le champ *adresse* est l'adresse de la zone mémoire réservée en C pour ce tableau. Ce sera, en fonction du type de l'objet :

INT_ARRAY

l'adresse d'une zone statique d'entiers (type int du C).

CHAR_ARRAY

l'adresse d'une zone de caractères[2] (1 octet par caractère).

STRING_ARRAY

l'adresse d'un tableau de pointeurs vers des chaînes[3] terminées par le caractère nul. Il peut être déclaré sous la forme : *char *tab[size];*

---

[2]Attention au mode de codage des caratères. Voir annexe E.

[3]Attention au mode de codage des chaînes. Voir annexe E.

***Attention**: il n'y a pas de test de débordement sur la taille des chaînes manipulées par l'interface. Ne pas oublier d'allouer effectivement l'espace pour les chaînes, et d'initialiser le tableau de pointeurs.*

SINGLE_FLOAT_ARRAY

l'adresse d'une zone de réels 32 bits IEEE.

DOUBLE_ARRAY

l'adresse d'une zone de réels 64 bits IEEE.

SYMBOL_ARRAY

l'adresse d'une sous-table d'associations (*chaîne d'un identificateur*, *représentation interne*). La partie chaîne est initialisée par le programmeur, la partie représentation interne est initialisée par Prolog.
Pour ce type de descripteur, le champ *name* définit le préfixe par défaut pour les symboles de la table (c'est-à-dire que si la chaîne *id_name* correspond à un nom abrégé, c'est ce préfixe par défaut qui est utilisé).
Ce type d'objet permet de construire automatiquement une table des valeurs internes des identificateurs Prolog que l'on désire manipuler. Tous les identificateurs de la tables sont permanents (cf. *set_permanent_symbol*) par défaut.

7.5.1. Exemple de zone commune de données

On déclare ici deux tableaux partagés par C et Prolog. Le premier est un tableau de 100 entiers, nommé *com* en C et *:com* en Prolog. Le deuxième est un tableau de 3 chaînes de 10 caractères au plus, nommé *m:str* en Prolog, et *str* en C.

```
#include  "proext.h"
int com[100];
char *str[3] =
   {
   "0123456789",
   "          ",
   "          "
   };

EXTERNAL_DESCRIPTOR descTable1 [] =
   {{":com",INT_ARRAY+OFFSET_ZERO_BASED, 100, (POINTER)com },
    {"m:str", STRING_ARRAY,   3, (POINTER) str },
    {     0,         0,       0,           0   }
   };
```

Supposons que ces déclarations soient contenues dans le module source C *table.c*, il peut être lié à Prolog après l'avoir compilé, en exécutant *prolink* pour l'édition de liens avec le module objet de *table* et le descripteur *descTable1*. (Voir le manuel d'utilisation §2.8.)

Au moment du chargement, Prolog créera automatiquement un lien avec le tableau C. Ce tableau pourra être modifié et consulté à partir des commandes standard Prolog *assign* et *val*. Il faut noter que pour les données partagées déclarées avec OFFSET_ZERO_BASED les indices Prolog commencent à 0, que pour les autres tableaux ils commencent à 1, et que les indices C commencent à 0. Les commandes suivantes C et Prolog ont strictement le même effet, et travaillent sur la même zone mémoire:

Prolog	C
`assign(:com[4], 100);`	`com[4] = 100;`
`val(:com[4],_x) outl(_x);`	`printf("%ld\n",com[4]);`
`val(m:str[1],_x) outm(_x);`	`printf("%s",str[0]);`
`assign(m:str[2],"abc");`	`strcpy(str[1],"abc");`

7.6. Ajout de fonctions externes

La définition d'une fonction externe se fait par la déclaration d'un descripteur. On distingue deux types de fonctions externes :

- les fonctions qui effectuent un traitement et se terminent; le champ *type* du descripteur vaudra alors C_FUNCTION ou DIRECT_C_FUNCTION. Les objets de type DIRECT_C_FUNCTION seront décrits au paragraphe 7.7.

- les fonctions non déterministes qui peuvent effectuer plusieurs traitements différents et donc retourner plusieurs résultats possibles; le champ *type* du descripteur vaudra alors C_FUNCTION_BACKTRACK.

On trouvera parmi les types d'objets possibles à déclarer par les descripteurs : C_FUNCTION_PROTECTED et C_FUNCTION_BACKTRACK_PROTECTED. Ces types identifient les mêmes objets que les types C_FUNCTION et C_FUNCTION_BACKTRACK, seulement leur "visibilité" change. En effet les objets de type PROTECTED seront cachés pour le mode de mise au point.

C_FUNCTION

La déclaration d'un objet de ce type, permet de créer automatiquement une règle relais Prolog, de nom *name* et d'arité *size*. L'exécution de cette règle Prolog se fera par l'appel de la fonction C dont l'adresse se trouve dans le champ *adresse*. Les arguments de la règle peuvent être accédés depuis la fonction C par les procédures standard de communication de données *get_...* et *put_...* . La fonction C pointée par *adresse* doit être déclarée de type *int* et est appelée par Prolog sans paramètres. Elle doit retourner la condition de terminaison (-1 pour ECHEC, 0 pour SUCCES, > 0 pour ERREUR).
Permet de construire des règles prédéfinies externes plus simplement que par l'utilisation directe des parasites (*/?n*) et de la procédure relais *user_rule* (cf. Annexe D).

C_FUNCTION_BACKTRACK

La déclaration d'un objet de ce type, permet de créer automatiquement une règle relais Prolog, de nom *name* et d'arité *size*. L'exécution de cette règle Prolog se fera par l'appel de la fonction C dont l'adresse se trouve dans le champ *adresse*. Les arguments de la règle peuvent être accédés depuis la fonction C par les procédures standard de communication de données *get_...* et *put_...* . La fonction doit être déclarée de type *int*. Elle est appelée par Prolog avec un seul paramètre de type *long* indiquant le numéro de l'appel courant. Elle doit retourner la condition de terminaison (ECHEC pour un échec, SUCCES pour un succès, > 0 pour une erreur, SUCCESS_END_OF_C_BACKTRACK pour indiquer le succès du dernier appel ou FAIL_END_OF_C_BACKTRACK pour indiquer l'échec du dernier appel). Lors d'une coupure, un appel supplémentaire à cette fonction sera fait avec en argument, la valeur conventionnelle CUT_OF_C_BACKTRACK. Le but de cet appel n'est pas de trouver une autre solution, mais de signifier qu'il n'y aura plus d'autres appels et ainsi permettre de terminer les traitements en cours (comme par exemple: libérer de la mémoire allouée, fermer des fichiers, …). Pendant un appel à cette fonction, il est possible de mémoriser une valeur entière dans l'espace de Prolog, à l'aide de la fonction *store_C_backtrack_data(long ptr)* ou de récupérer cette valeur par le retour de la fonction *long restore_C_backtrack_data(void)*. Permet de construire des règles prédéfinies en langage externe non déterministes.

7.6.1. Exemple de déclaration de règle prédéfinie

Le fichier suivant, lié à Prolog, produit automatiquement la nouvelle règle prédéfinie *sys:reverse(s1,s2)*, unifiant *s2* avec la chaîne *s1* inversée, ainsi que la règle *sys:enumerate(x,b,e)* qui unifie successivement *x* avec les entiers compris entre *b* et *e*. Si l'on veut utiliser le nom abrégé, il faut rajouter le nom dans le contexte fermé "sys" en effaçant la commande:

```
> add_implicit("sys","reverse")
add_implicit("sys","enumerate");


#include "proext.h"
#include <stdio.h>
#include <string.h>
#define MAX_STRING 512

int reverse();

EXTERNAL_DESCRIPTOR descTable2[] =
   {{"sys:reverse", C_FUNCTION, 2, (POINTER) reverse},
    {"sys:enumerate", C_FUNCTION_BACKTRACK, 3,
                                        (POINTER) enumerate},
    { 0, 0, 0, 0 }
   };

reverse()
```

```
   {
   int err, i, j, lg;
   char c1, c[MAX_STRING];
   if ( ! get_string(1,c,&err) )
      return err;
   lg = strlen(c);
   for (i=0; i<lg/2; i++)
      {
      j = lg -1 -i;
      c1 = c[i];
      c[i] = c[j];
      c[j] = c1;
      }
   if ( ! put_string(2,c,&err) )
      return err;
   return 0;
   }

typedef struct {
   int direction, current, start, end;
   } enumRecord, *EnumMemory;

int enumerate(call)
long call;
   {
   EnumMemory mem;
   int err;
   if (call==CUT_OF_C_BACKTRACK) /* cut */
      {
      mem = (EnumMemory) restore_C_backtrack_data();
      free(mem);
      return;
      }

   if (call==1) /*first call */
      {
      /* mem constituera la mémoire du prédicat d'un appel
      à l'autre */

      mem = (EnumMemory) malloc(sizeof(enumRecord));

      get_integer(2, &mem->start, &err);
      if (err) { free(mem); return 253;}
      get_integer(3, &mem->end, &err);
      if (err) { free(mem); return 253;}

      mem->direction = (mem->start > mem->end) ? -1 : 1;
      mem->current = mem->start;

      store_C_backtrack_data(mem);
   }

   /* on récupère l'adresse des données qui a été mémorisée
   */
   if (call != 1)
      mem = (EnumMemory) restore_C_backtrack_data();

   /* instantiation du premier argument du prédicat */
```

```
   put_integer(1,mem->current,&err);

   if (mem->current == mem->end)
      {
      /* c'est fini, il faut libérer */
      free(mem);
      return SUCCESS_END_OF_C_BACKTRACK;
      }
   else
      /* on incrémente le compteur */
      mem->current += mem->direction;

   return err;
}
```

Il faut maintenant compiler le fichier ci-dessus, faire l'édition de liens (voir le manuel d'utilisation §2.8.), puis activer le nouveau Prolog pour obtenir :

```
Prolog II+, ..
...
>sys:reverse("0123456",x);
{x="6543210"}
>sys:enumerate(x,-2,2);
{x=-2}
{x=-1}
{x=0}
{x=1}
{x=2}
>
```

7.7. Ajout de fonctions externes à appel direct

Les fonctions externes à appel direct, contrairement aux autres fonctions externes accessibles par la méthode des descripteurs, sont des fonctions qui n'ont pas d'interface C pour le passage des paramètres. Une telle fonction C peut être écrite avec ses paramètres, comme si elle devait être appelée par un programme C classique. En particulier cela peut être une fonction d'une librairie dont on n'a pas la maîtrise des sources. En effet, pour ce type de fonction, la transformation des données (Prolog - C et C - Prolog) se fera en Prolog, par l'intermédiaire de la règle prédéfinie *callC*.

Cette méthode permet donc de communiquer des types de données communs aux deux langages, à savoir : des entiers, des réels, des chaînes de caractères et des tableaux homogènes de ces types.

La déclaration d'une fonction externe à appel direct se fait par la déclaration d'un descripteur dont le champ *type* vaut DIRECT_C_FUNCTION.

DIRECT_C_FUNCTION

La déclaration d'un objet de ce type permet de créer automatiquement une règle relais Prolog de nom *name* et d'arité *size*, qui ne peut être effacée qu'à travers la primitive *callC* et dont l'exécution consiste à appeler la fonction C pointée par le champ *adresse* du descripteur.
Cette fonction C ne doit pas avoir plus de 20 arguments. Si le champ *size* du descripteur vaut -1, le type et le nombre d'arguments sont variables suivant l'appel. Il est possible d'appeler *sprintf* par exemple (voir § 5.2 de ce manuel).

7.7.1. Primitive CallC

La primitive prédéfinie *callC* détermine, au moment de l'appel, le mode de passage des arguments à la fonction C, leur type et le type de la valeur de retour, d'après les arguments effectifs du prédicat Prolog et conformément aux conventions adoptées que nous décrirons dans le prochain paragraphe. Ceci permet d'appeler des fonctions avec nombre et type d'argument variables.

callC(t1)
callC(t1,t2)

t1 est un terme représentant l'appel du prédicat Prolog, avec ses paramètres conformément aux conventions choisies. *t2* est un terme représentant le résultat. Si *t2* est *nil* le résultat de la fonction est ignoré. La forme *callC(t1)* est équivalente à *callC(t1,nil)*.
Les fonctions qui peuvent être appelées à travers ce prédicats, sont les fonctions décrites par l'utilisateur avec un descripteur dont le type est DIRECT_C_FUNCTION, ainsi que les fonctions *sprintf* et *sscanf* qui sont prédéclarées. Sous les systèmes permettant l'implantation de la primitive *lkload*, il est possible d'appeler sans déclaration des fonctions existant dans l'environnement Prolog. La primitive *callC* réalise alors dynamiquement le lien lors du premier appel. Il faut pour cela que le fichier exécutable *prolog* se trouve dans le répertoire courant.

Exemples:

```
> callC(sscanf("123","%lf",<"R",y>));
{y=1.230000000000000e+02}
> eq(x,"123") callC(sscanf(x,"%2f",<"R",y>));
{x="123", y=1.200000000000000e+01}
> eq(f,"%x %o") callC(sscanf("12 12",f,<"I",x>,<"I",y>));
{x=18, y=10}
> callC(sprintf(<"",x,80>,"valeur: %ld",200));
{x="valeur: 200"}
```

7.7.2. Conventions Prolog pour la communication des données

Il s'agit ici de pouvoir exprimer en Prolog toutes les informations nécessaires pour pouvoir faire l'appel d'une fonction C, concernant ses arguments et sa valeur de retour. On s'intéressera donc :

- au mode de passage des arguments,
- au type des arguments et de la valeur de retour de la fonction,
- à la valeur initiale des arguments,
- à la valeur résultat des arguments et de la fonction,
- à d'autres informations techniques liées au type de l'objet.

Plus qu'au mode de passage d'une donnée, on s'intéressera au rôle de la donnée, à savoir : donnée d'entrée et/ou de sortie.

En effet en Prolog, une donnée se voyant affecter une valeur, ne pourra plus en changer (sauf par backtracking, mais c'est assimilable en comparant avec C, à une autre exécution). Une procédure Prolog qui attendrait une valeur en entrée et un résultat en sortie, devrait utiliser deux données. C'est là la différence essentielle entre Prolog et C, il est donc important de connaître si une donnée (initialisée ou pas) doit changer de valeur au cours de l'exécution de la fonction C. Dans la suite de ce chapitre, on fera donc la distinction entre argument avec valeur de retour et argument sans valeur de retour. Le mode de passage par adresse ou par valeur en sera déduit automatiquement.

Il est possible de passer en paramètre:

1. un entier long avec ou sans valeur de retour,
2. un réel double précision avec ou sans valeur de retour,
3. une chaîne de caractères avec ou sans valeur de retour,
4. un tableau d'entiers, de chaînes de caractères ou de réels double précision, avec ou sans valeur de retour.

Il est possible d'attendre en résultat de la fonction:

1. un entier long,
2. un réel double précision.

Voyons à présent, quelles sont les conventions. Pour les illustrer sur des exemples, on supposera qu'un lien a été déclaré entre le prédicat `relaiProlog` et la fonction `fonctionC`.

7.7.2.1. Convention pour des paramètres sans valeur de retour

Dans ce cas, il n'est pas utile de spécifier le type de la donnée, Prolog le connaît puisque la donnée a déjà une valeur.

Les données de type entier, réel ou chaîne sont représentées par la donnée Prolog elle même.

Les données de type tableau homogène d'entiers, de réels ou de chaînes sont représentées par une liste Prolog (terminée éventuellement par *nil*), d'entiers de réels ou de chaînes.

Par exemple:

```
:relaiProlog(180, 3.14e0, "pi", 2.3.5.7.11.13.17.nil,
"b"."a"."ba".nil)
```

est équivalent aux instructions C suivantes:

```
{ long arg1 = 180L; double arg2 = 3.14;
char arg3[] = {'p','i','\0'};
long arg4[] = {2,3,5,7,11,11,13,17};
char *arg5[] = {"b","a","ba"};
fonctionC(arg1,arg2,arg3,arg4,arg5); }
```

Les chaînes et les tableaux sont toujours passés par adresse selon la convention habituelle en C. Prolog effectue de toute façon une copie des arguments dans une zone intermédiaire dont la taille peut être paramétrée sur la ligne de commande (voir le chapitre 2 du Manuel d'Utilisation).

7.7.2.2. Convention pour le retour de la fonction

Il est nécessaire de connaître le type du retour de la fonction, et de disposer d'un terme Prolog qui sera unifié avec sa valeur. La convention est la suivante:

le terme qui représente le résultat doit être un doublet :

< *type_résultat*, *variable_résultat* >

- dont le premier élément *type_résultat* est une chaîne qui indique le type: "I" pour un retour entier, "R" ou "X" pour un retour réel.
- dont le deuxième élément *variable_résultat* sera unifié avec la valeur retournée par la fonction (entier ou réel).

7.7.2.3. Convention pour des paramètres avec valeur de retour

Pour une donnée C avec valeur de retour attendue, il est nécessaire d'avoir deux données Prolog: une première qui doit être connue au moment de l'appel et qui sert à transmettre la valeur initiale à la fonction C, une deuxième qui sera unifiée avec la nouvelle valeur obtenue de la fonction C.

On distingue parmi les types de données : des types simples et des types qui nécessitent une allocation de zone mémoire.

Note : Certaines conventions avec valeur(s) de retour, spécifient des tailles de zones mémoire à allouer pour ranger le(s) résultat(s) des fonctions C appelées. L'utilisateur doit donc impérativement s'assurer que les zones sont suffisantes pour les résultats attendus. Il s'assurera également que les types des arguments sont cohérents avec les déclarations faites en C.

La convention pour les données de type simple, c'est à dire les **entiers** et les **réels**, est la suivante :

l'argument effectif du prédicat Prolog est un doublet :

< *valeur_initiale*, *variable_résultat* >

- dont le premier élément *valeur_initiale* est la valeur initiale de la zone dont l'adresse est passée en paramètre à la fonction C. Si la valeur est quelconque on peut mettre comme valeur initiale la marque "I" (pour indiquer un entier) ou "R" (pour un réel).
- dont le deuxième élément *variable_résultat* sera unifié avec la nouvelle valeur de l'argument rendue par la fonction (entier ou réel).

Par exemple:

```
:relaiProlog(<180,x>, <3.14e0,y>)
```

est équivalent aux instructions C suivantes:

```
{ long arg1 = 180L; double arg2 = 3.14;
fonctionC(&arg1,&arg2); }
```

suivies des affectations des variables x et y avec les nouvelles valeurs de arg1 et arg2.

La convention pour les données qui nécessitent une zone mémoire dépend du type de la donnée :

Pour une **chaîne de caractère**, il est nécessaire d'avoir une zone de caractères, dont on doit définir la taille, susceptible de contenir dans un premier temps la chaîne initiale puis la chaîne résultat:

l'argument effectif du prédicat Prolog est un triplet:

< *valeur_initiale*, *variable_résultat*, *taille_max_résultat* >

- dont l'élément *taille_max_résultat* est un entier spécifiant la taille de la zone à allouer pour l'opération, dont l'adresse est transmise à la fonction C.
- dont l'élément *valeur_initiale* est la valeur initiale copiée dans la zone allouée pour l'opération.
- dont l'élément *variable_résultat* sera unifié avec le résultat (chaîne Prolog).

Par exemple:

```
:relaiProlog(<"pi",x,100>)
```

est équivalent aux instructions C suivantes:

```
{ char arg1[100];
strcpy(arg1,"pi");
fonctionC(arg1); }
```

suivies de l'affectation de la variable x avec la nouvelle valeur de arg1.

Pour un **tableau d'entiers ou de réels**, il est nécessaire d'avoir une zone d'entiers ou de réels qui tient lieu de tableau, dont on définit le nombre d'éléments, et susceptible de contenir dans un premier temps les éléments du tableau en entrée puis les éléments du tableau en sortie:

l'argument effectif du prédicat Prolog est un quadruplet:

< *valeur_initiale*, *variable_résultat*, *taille_résultat*, *taille_tableau* >

- dont l'élément *taille_tableau* est un entier spécifiant le nombre d'éléments de la zone à allouer pour l'opération, dont l'adresse est transmise à la fonction C.
- dont l'élément *taille_résultat* indique le nombre d'éléments valides du tableau à prendre en compte pour le retour.
- dont l'élément *valeur_initiale* est la liste Prolog terminée par *nil*, des valeurs initiales des éléments du tableau. Le type de ces éléments détermine le type du paramètre.
- dont l'élément *variable_résultat* sera unifié avec le résultat (liste Prolog terminée par *nil*, d'entiers ou de réels).

Par exemple:

```
:relaiProlog(<2.4.6.8.12,x,1,5>, <0e0.nil,y,5,5>)
```

est équivalent aux instructions C suivantes:

```
{ long arg1[]={2,4,5,6,8,12}; double arg2[5];
arg2[0] = 0.0;
fonctionC(arg1,arg2); }
```

suivies des affectations des variables x et y avec les nouvelles valeurs de arg1 et arg2.

Pour un **tableau de chaînes de caractères**, il est nécessaire d'avoir une zone de pointeurs pour représenter le tableau, dont on doit définir le nombre d'éléments, et éventuellement une zone de caractères pour stocker tous les caractères de toutes les chaînes du tableau, dont on doit également définir la taille.

l'argument effectif du prédicat Prolog est un quadruplet:

< *valeur_initiale*, *variable_résultat*, *nbre_max_de_caractères, taille_tableau* >

- dont l'argument *taille_tableau* est un entier spécifiant le nombre d'éléments de la zone à allouer pour l'opération, dont l'adresse est transmise à la fonction C. La fin du tableau doit être indiquée par le pointeur NULL après le dernier élément.
- dont l'argument *valeur_initiale* est une liste Prolog terminée par nil de chaînes pointées par le tableau en entrée (*""*.*nil* au minimum).
- dont l'argument est un entier, il offre la possibilité de laisser à Prolog la gestion (allocation et libération) du buffer de caractères pour les chaînes en sortie. Si l'utilisateur gère le(s) espace(s) des chaînes en sortie, *nbre_max_de_caractères* doit valoir 0. Si l'utilisateur laisse à Prolog le soin d'allouer et par la suite libérer un buffer de caractères, *nbre_max_de_caractères* doit être non nul, il indique alors le nombre de caractères total maximal nécessaire pour stocker tous les caractères de toutes les chaînes du tableau. La valeur du premier élément du tableau (pointeur de chaîne de caractères) sera dans ce cas l'adresse de cet espace (où est copiée la première chaîne en entrée).
- dont l'élément *variable_résultat* sera unifié avec le résultat, une liste Prolog terminée par *nil*, de chaînes.

Par exemple:

```
:relaiProlog(<"il"."était"."une"."fois",x,100,5>)
```

est équivalent aux instructions C suivantes:

```
{ char * arg1[5]; char buffer[100];
strcpy(buffer,"il"); arg1[0] = buffer;
arg1[1] = "était"; arg1[2] = "une"; arg1[3] = "fois";
fonctionC(arg1); }
```

suivies de l'affectation de la variable x avec les nouvelles valeurs de arg1.

7.7.3. Exemple : méthode simple pour appeler une fonction C

Nous montrons ici des exemples avec divers types d'arguments, utilisant des fonctions déclarées par l'utilisateur, et une fonction système.

```
#include <stdio.h>
#include <string.h>
#include "proext.h"

#define REAL double
#define INTEGER long

static REAL complete_example(ii1,io1,ri1,ro1,si1,so1,tii1,
                             tri1,tio1,tro1,tsi1,tso1,reserved)
/*-------------------------------------------------------*/
INTEGER ii1,*io1;
REAL ri1,*ro1;
char *si1;
char *so1;
```

```
REAL *tri1;
REAL *tro1;
INTEGER *tio1,*tii1;
char **tsi1,**tso1;
int reserved;
{
int i;

printf("ii1= %d\n",ii1);      /*integer in input*/
printf("io1= %d\n",*io1);     /*integer in output*/
printf("ri1= %g\n",ri1);      /*real in input*/
printf("ro1= %g\n",*ro1);     /*real in output*/
printf("si1= %s\n",si1);      /*string in input*/
printf("so1= %s\n",so1);      /*string in output*/
for (i=0;i<2;i++)                          /*integer array*/
      printf("tii1[%d]= %d\n",i,tii1[i]); /*in input*/
for (i=0;i<2;i++)                          /*real array*/
      printf("tri1[%d]= %g\n",i,tri1[i]); /*in input*/
for (i=0;i<3;i++)                          /*integer array*/
      printf("tio1[%d]= %d\n",i,tio1[i]); /*in output*/
for (i=0;tsi1[i];i++)                      /*string array*/
      printf("tsi1[%d]= %s\n",i,tsi1[i]); /*in input*/
for (i=0;i<3;i++)                          /*real array*/
      printf("tro1[%d]= %g\n",i,tro1[i]); /*in output*/
*io1 = 3333;                        /*integer in output*/
*ro1 = - 8888.;                     /*real in output*/
strcpy(so1,"bonjour");              /*string in output*/
for (i=0;i<10;i++)                  /*integer array*/
      tio1[i] = i+200;              /*in output*/
for (i=0;i<10;i++)                  /*real array*/
      tro1[i] = (float) i +200.;    /*in output*/
for (i=0;tso1[i];i++)               /*string array*/
      printf("strin(%d)=%s\n",i,tso1[i]);/*in output*/
if(reserved)  /*space for strings reserved by prolog */
      {
      strcpy(tso1[0],"Zone1"); /*first pointer(tso1[0])*/
                                /*initialized by Prolog */
      tso1[1] = tso1[0]+6;    /*others pointers initialized*/
                              /*by the user */
      strcpy(tso1[1],"Zone2");
      tso1[2] = NULL;         /*end of array */
      }
else       /*space for strings reserved by the user */
      {
      tso1[0] = "Zone1";
      tso1[1] = (char *) malloc(10);
      strcpy(tso1[1],"Zone2");
      tso1[2] = NULL;
      }
return 1234.;           /*returned value */
}

static REAL average(tab,n)/*
----------------------------*/
INTEGER *tab,n;
{
INTEGER i,sum=0;
for (i=0;i<n;i++) sum += tab[i]; return (sum/n);
```

```
}

static mystrcmp(a1,a2)/*
------------------------*/
char **a1,**a2;
{
return strcmp(*a1,*a2);
}

static quicksort(tab_in,tab_out)/*
-------------------------------*/
char **tab_in, **tab_out;
{
int i,nbr;

for (nbr=0;tab_in[nbr];nbr++)
      tab_out[nbr] = tab_in[nbr];
tab_out[nbr] = NULL;
qsort(tab_out,nbr,sizeof(char *),mystrcmp);
}

static reverse(str)/*
------------------------*/
char *str;
{
char c;
INTEGER i,len;

len = strlen(str);
for (i=0;i<len/2;i++)
           {
           c = str[i];
           str[i] = str[len - (i+1)];
           str[len - (i+1) ] = c;
           }
}

EXTERNAL_DESCRIPTOR calld_desc[] =
{
{":complete_example", DIRECT_C_FUNCTION, 13, (POINTER)
complete_example},
{":average", DIRECT_C_FUNCTION, -1, (POINTER) average},
{":reverse", DIRECT_C_FUNCTION, 1, (POINTER) reverse},
{":qsort", DIRECT_C_FUNCTION, 2, (POINTER) quicksort},
{0, 0, 0, 0}
};
```

En supposant que le fichier ci-dessus s'appelle *callc.c*, il faut le compiler, faire l'édition de liens (voir le manuel d'utilisation §2.8.), puis activer le nouveau Prolog pour obtenir :

```
Prolog II+, ..
...
>  eq(r_eserved,30)
   callC(complete_example
      (
      11,          /*integer in input */
      <155,i>,     /* integer in output with initialization*/
```

```
      +13.0e0,     /* real in input*/
      <+222.0e0,r>,/* real in output with initialization*/
      "str_input", /* string in input*/
      <"ee",s,10>, /* string in output with initialization*/
      123.456.nil, /*array of integers in input */
      14.0e0.15.0e0.nil,   /*array of reals in input */
      <11.22.33,w,5,20>,   /*array of integers in output
                            /*with initialization*/
      <+44.0e0.+55.0e0.+66.0e0,t1,5,20>, /*array of reals in
                              output with initialization*/
      "list_str_in1"."list_str_in2".nil, /*array of STRINGS*/
                                         /*in input */
      <"list_str_out1"."list_str_out2"."list_str_out3".nil,
            T_S,r_eserved,5>,
      r_eserved    /*integer in input */
      ),
   <"R",m>)     /*result */
   ;

ii1= 11
io1= 155
ri1= 13
ro1= 222
si1= str_input
so1= ee
tii1[0]= 123
tii1[1]= 456
tri1[0]= 14
tri1[1]= 15
tio1[0]= 11
tio1[1]= 22
tio1[2]= 33
tsi1[0]= list_str_in1
tsi1[1]= list_str_in2
tro1[0]= 44
tro1[1]= 55
tro1[2]= 66
strin(0)=list_str_out1
strin(1)=list_str_out2
{r_eserved=30,i=3333,
r=-8.888000000000000e+03,
s="bonjour",
w=200.201.202.203.204.nil,
t1=2.000000000000e+02.2.010000000000e+02.2.020000000000e+02.
   2.030000000000e+02.2.040000000000e+02.nil,
T_S="Zone1" ."Zone2".nil,
m=1.234000000000e+03
}
> callC(average(1.3.5.nil,3),<"R",x>);
{x=3.000000000000e+00}
> callC(reverse(<"abcde",s,20>));
{s="edcba"}
> callC(reverse(<"abcdef",s,20>));
{s="fedcba"}
> callC(qsort("qqq"."zzz"."aaa"."iii".nil,<"".nil,s,0,50>));
/*0 because strings not created (already exist)*/
{s="aaa"."iii"."qqq"."zzz".nil}
```

8. Lancement d'un but Prolog par un programme C

8.1. Principes de base
8.2. Initialisation de Prolog
8.3. Empilement d'un but Prolog
8.4. Programmation
8.5. Méthode simple d'appel d'un but Prolog
8.6. Autres fonctions

Ce chapitre suppose une bonne connaissance du chapitre 7, les lecteurs non expérimentés peuvent sauter ce chapitre en première lecture.

La machine Prolog peut être utilisée comme un sous-programme d'un programme quelconque. Pour simplifier la compréhension de sa manipulation depuis un langage procédural, on peut présenter Prolog comme une machine à états (ou points d'arrêt): un jeu de procédures permet de faire transiter la machine Prolog de l'état courant vers un nouvel état.

Hormis aux points d'arrêt de la machine Prolog, un programme C ne peut être appelé que par une règle prédéfinie. Celle-ci peut empiler un nouveau but, ou provoquer une erreur.

A un point d'arrêt de la machine Prolog, un programme C peut: empiler un nouveau but, réactiver la machine jusqu'au prochain point d'arrêt pour avoir une autre solution, ou provoquer l'abandon du but en cours, avec retour à l'état précédent.

8.1. Principes de base

L' «état normal», pour une machine Prolog, consiste évidemment en l'exécution de la boucle d'effacement d'une suite de buts (cf. chapitre 2 de ce manuel); cette boucle est appelée parfois *horloge Prolog*. Lorsqu'elle se trouve dans cette situation, nous dirons que la machine Prolog est *active*.

Quand la machine Prolog s'active, elle possède déjà une suite initiale de buts à effacer; le processus d'effacement (choix d'une règle, unification de la tête, etc…) dure tant que la suite de buts courante n'est pas vide; quand cette suite est vide, la machine tombe dans un point d'arrêt.

Dans ce paragraphe on définit les divers états dans lesquels la machine Prolog peut se trouver lorsqu'elle n'est pas active, tels que les voit le programmeur C (ou Pascal, ou autre…); ces états seront appelés des *points d'arrêt*. On introduit aussi les procédures qui font passer la machine d'un point d'arrêt à un autre.

A un point d'arrêt, les états possibles de la machine sont :

0. Machine non initialisée. Ceci n'est pas vraiment un état de la machine, mais la situation dans laquelle on se trouve avant l'initialisation de celle-ci (allocation de l'espace, etc…). La procédure *ProStart(...)*, appelée une seule fois par session, fait passer dans l'état suivant :

1. Rien a exécuter (NO_GOAL). La machine est initialisée, mais *la suite de buts à effacer est vide.*
   La machine vient aussi dans cet état par son fonctionnement normal, lorsqu'il n'y a plus aucune manière possible d'effacer la suite de buts courante, c'est-à-dire lorsque toutes les solutions du problème courant ont été *précédemment* obtenues. Dans cet état, toute activation de la machine (par *next_solution()*) la ramène immédiatement dans le même état NO_GOAL.

2. But prêt à être exécuté. Ceci est un étape préalable dans l'activation de la machine : la procédure *new_goal()* vient d'installer la suite de buts *exec(_b, _x);* le programme appelant doit maintenant, à l'aide des routines standard (*put_integer*, *put_term*, etc…), unifier *_b* avec le but ou la liste de buts à effacer.

3. En exécution Prolog, dans une règle prédéfinie. Pendant l'effacement d'une règle externe, le programme C (ou Pascal, etc…) en question obtient le contrôle de l'exécution et la machine Prolog «paraît» à l'arrêt; cependant, cette situation diffère des états précédents dans le fait qu'il ne correspond pas à un point d'arrêt : la machine Prolog n'est pas *naturellement* arrêtée au début ou a la fin d'un effacement, mais au contraire elle se trouve en plein milieu d'un effacement, en attente de la terminaison d'une procédure externe.

4. Solution trouvée (SOLUTION_EXISTS). C'est le point d'arrêt le plus fréquent : la suite de buts courante vient d'être entièrement effacée. Le programme appelant peut récupérer les solutions à travers l'argument *_x* de l'appel *exec(_b,_x)*.

   La machine Prolog a mémorisé tous les points de choix de l'effacement en cours. A partir de cet état, on peut appeler la procédure *next_solution()* pour relancer la machine afin d'obtenir les autres solutions.

   La séquence d'états la plus fréquente sera donc:

   NO_GOAL → «But prêt» → SOLUTION_EXISTS → …
   … → SOLUTION_EXISTS → NO_GOAL

5. Erreur rencontrée (ERROR). C'est l'état dans lequel la machine est mise lorsqu'une erreur a fait avorter son fonctionnement normal. Une activation (par *next_solution()*) la fera passer alors dans l'état NO_GOAL.

Dans n'importe lequel des états 1 à 4 ci-dessus, la procédure *new_goal()* peut être appelée pour installer un nouveau problème par-dessus le problème courant, lequel est conservé exactement dans l'état où il se trouve afin de pouvoir ultérieurement continuer à obtenir ses solutions.

De la même manière, dans les états 2 à 5 on peut appeler la procédure *kill_goal()*, qui provoque l'abandon du problème en cours et le retour au problème précédent, qui se retrouve exactement dans l'état où il était quand on a appelé *new_goal()*.

Les fonctions permettant d'activer la machine Prolog (i.e. de la faire transiter vers un autre état) sont:

ProStart(..)

Fonction permettant d'initialiser la machine Prolog. Cette fonction doit être appelée une fois, avant tout autre appel concernant Prolog. Initialise la machine dans l'état 1.

ProFinal(..)

Fonction permettant de terminer et libérer la machine et l'environnement Prolog. Elle a un argument de type entier qui spécifie le status de terminaison de Prolog.

new_goal()

Mémorise l'état courant de la machine, et fait passer dans l'état 2 (but prêt à être exécuté). L'état 2 est initialisé avec l'appel *exec(_b,_s)* dont les variables doivent être instanciées par le programme appelant APRES l'appel de *new_goal*.
Cette procédure permet d'empiler des activations de buts *_b* jusqu'à une profondeur limitée seulement par la taille des piles de Prolog.

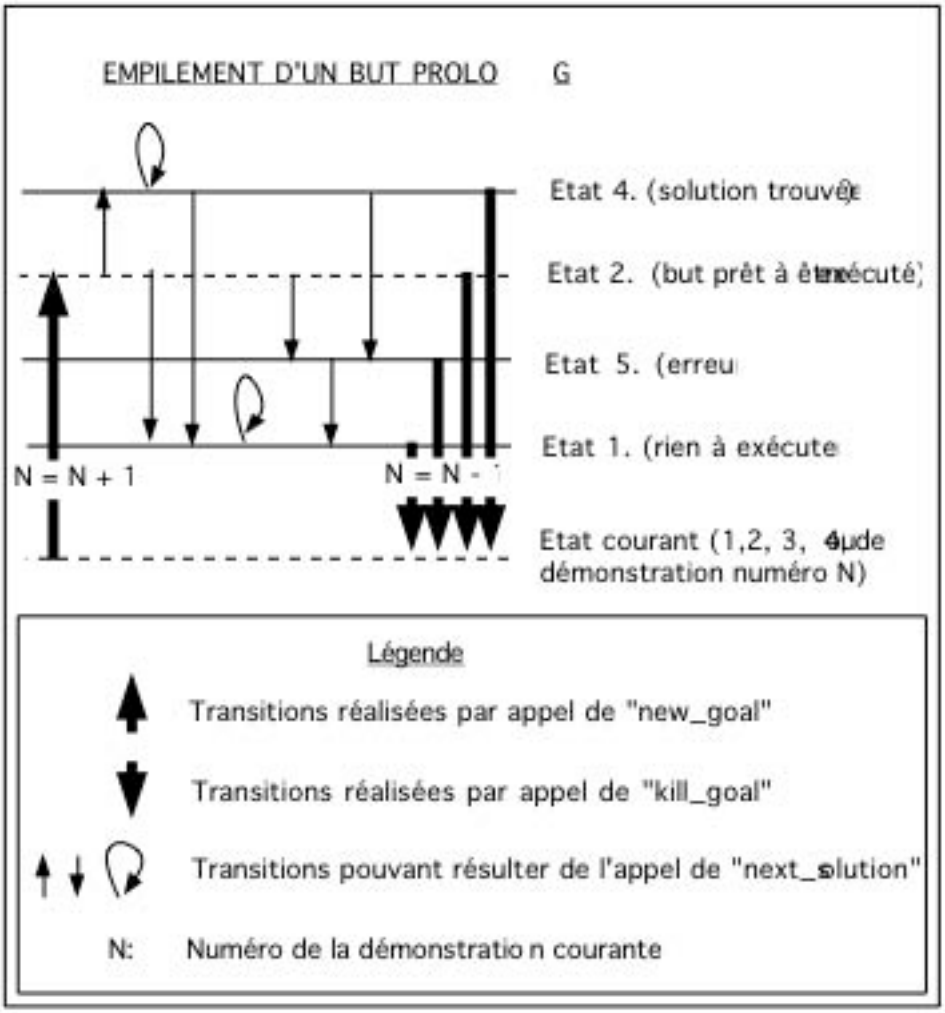
kill_goal()

Revient à l'état exact dans lequel se trouvait la machine avant le dernier appel de *new_goal*. *new_goal* et *kill_goal* fonctionnent comme des parenthèses encadrant l'effacement d'un nouveau but Prolog.

next_solution()

Fonction d'activation de la machine, la faisant transiter vers l'état d'arrêt suivant (1, 4, ou 5). La fonction a pour valeur le nouvel état de la machine Prolog: SOLUTION_EXISTS (0), NO_GOAL (-1), ERROR (>0).

Pour tout état Prolog différent de l'état non initialisé, le diagramme suivant définit les transitions possibles:



8.2. Initialisation et terminaison de Prolog

Au démarrage d'une application, Prolog se trouve dans un état non initialisé. La fonction *ProStart* permet d'initialiser Prolog avec une configuration définie dans la chaîne de caractères passée en paramètre, chaîne représentant les options de lancement de Prolog (taille des piles, fichier d'entrée,..). La fonction retourne 0 si tout c'est bien passé, un nombre non nul positif, correspondant au numéro de l'erreur, si une erreur s'est produite et l'initialisation ne s'est pas faite.

L'appel de toute autre procédure que *ProStart* dans l'état non initialisé est illégal.

Les types de la fonction et de son argument doivent être comme ci décrit:

```
int ProStart(params)
```

```
   char *params;
```

Lorsque l'utilisation de Prolog est terminée, la fonction *ProFinal* permet de libérer les structures et terminer Prolog. La fonction n'a pas de valeur de retour, elle a un argument qui spécifie le status de terminaison de Prolog, en fonction duquel *ProFinal* imprime un message. Ce message doit être défini dans le fichier d'erreurs de Prolog. Le status doit valoir 0 pour une terminaison normale, un nombre strictement positif pour un status d'erreur. Ce status d'erreur pourra être, par exemple, l'erreur retournée par une des fonctions suivantes *new_goal*, *next_solution* ou *kill_goal*.

Après l'exécution de *ProFinal*, Prolog se trouve à nouveau dans un état non initialisé.

Les types de la fonction et de son argument doivent être comme ci décrit:

```
ProFinal(status)
   int status;
```

Voici un exemple d'utilisation :

```
mon_initialisation_prolog()
   {
   int err;
   if ((err = ProStart("-c 400 -f cM prolog.po")) == 0)
            fprintf(stderr,"initialisation reussie");
   else     {fprintf(stderr,"initialisation echouee!");
            exit(err);}
   }


ma_terminaison_prolog(condition)
   int condition;
   {
   if (mon_erreur(condition))
      { mon_message_erreur(condition);
      ProFinal(0);
      }
   else
      ProFinal(condition);
   }
```

L'utilisation de la machine Prolog dans une application est délimitée par l'appel aux fonctions *ProStart* et *ProFinal*.

8.3. Empilement d'un but Prolog

L'empilement d'un but *_b* se fait à travers l'appel d'une règle relais *exec(_b,_s)*, qui garantit la conservation du but en cours et l'arrêt dans un état précis. Cette règle a deux arguments: le premier est un terme représentant le but *_b* à effacer, le deuxième est le terme qui sera obtenu en solution (c'est en général la liste des variables intéressantes du but *_b*).

Lorsqu'on a appelé *new_goal*, l'appel *exec(_b,_s)* est prêt à être exécuté. Il suffit donc d'utiliser les routines standard pour instancier les arguments (*put_term(1,..)* pour instancier *_b*, et *put_term(2,..)* pour définir les variables correspondant à la solution intéressante).

L'appel de la fonction *next_solution* provoque la transition de la machine Prolog vers un des états 1,4 ou 5. La valeur retournée par la fonction est l'état atteint: NO_GOAL (-1), ERROR (>0), ou SOLUTION_EXISTS (0).

La règle relais est l'équivalent de:

```
exec(_b,_s) ->
    block(_e,_b)
    condition(_e,_k)
    point_arret1(_k,_s)
    fail ;
exec(_b,_s) -> point_arret2(NO_GOAL,nil);

condition(_e,SOLUTION_EXISTS) -> free(_e) !;
condition(_e,_e) ->  integer(_e) !;
condition(_e,BLOCK_EXIT_WITHOUT_BLOCK) ->;

point_arret1(K,S) -> stop;

point_arret2(K,S)-> stop point_arret2(NO_GOAL,[]);
```

Conceptuellement, tout se passe comme si l'utilisateur se trouvait dans la règle *point_arret*: de par la manière dont est écrit *exec*, la valeur du 1er argument (obtenu par *get_integer(1,..)*) contient également l'état atteint. Si cet état est SOLUTION_EXISTS, le deuxième argument (obtenu avec *get_term(2,..)*) contient le terme définissant la solution intéressante.

8.4. Programmation

Le fichier *proext.h* contient les déclarations des procédures et des structures d'interface. Il faut mettre un *#include* de ce fichier en tête des modules utilisant ces procédures.

Voici un exemple de programme C imprimant toutes les solutions de *enum(i,-2,8)*, que l'on peut écrire tel quel dans le module C d'interface utilisateur : *prouser.c*, à la place de la déclaration EXTERNAL_DESCRIPTOR qui s'y trouve.

```
#define MAX_TAB 10
exemple()
   {
   int err;
   long i;
   P_SYMBOL pro_symbol();
   char tags[MAX_TAB];
   int  vals[MAX_TAB];
   int n = 1;
                              /* initialisation du but */
   new_goal();
   tags[0] = 'T'; vals[0] =  1;

   tags[1] = 'I'; vals[1] =  4;
   tags[2] = 'N'; vals[2] = pro_symbol("sys:enum");
   tags[3] = 'V'; vals[3] =  0;                /* i */
   tags[4] = 'I'; vals[4] = -2;
   tags[5] = 'I'; vals[5] =  8;
   put_term(1, MAX_TAB,0,0, tags,vals, 0,0, 5, &err);
   if (err)  goto error;
                              /* creation solution
interessante */
   tags[0] = 'V'; vals[0] =  0;
   put_term(2, MAX_TAB,0,0, tags,vals, 0,0, 0, &err);
   if (err)  goto error;
                         /* impression des solutions */
   while ((err = next_solution()) == SOLUTION_EXISTS)
      {
      get_integer(2, &i, &err);  /*valeur de i*/
      if (err)  goto error;
      printf("solution %d : %ld\n", n++,i);
      }
   if (err > 0)
      goto error;
   kill_goal();
   return 0;
error:
   printf("erreur %d a la %d ieme etape\n",err,n);
   kill_goal();
   return err;
   }

EXTERNAL_DESCRIPTOR prouser_desc[] =
   {{":exemple", C_FUNCTION, 0, (POINTER)exemple},
    {0,0,0,0}};
```

Vous pouvez trouver un exemple d'utilisation de Prolog comme un sous programme, dans le fichier *princip.c* qui est le module principal de Prolog.

8.5. Méthode simple d'appel d'un but Prolog

Une autre méthode, plus simple à utiliser, permet également depuis un langage externe, d'installer un but Prolog et d'en récupérer les solutions.

Elle est plus simple pour installer un but, grâce à la fonction *new_pattern*, équivalente à l'ensemble *new_goal*, *put_term* du but et *put_term* du résultat. Elle est moins efficace sur ce point là car elle utilise l'analyseur Prolog.

Elle est plus simple pour extraire des valeurs de la solution, grâce à la fonction *get_formats*, équivalente à l'ensemble *get_term* et traitement des différents tableaux.

Le principe de cette méthode est de spécifier au moment de l'installation du but, la structure des arguments et les valeurs qui y sont intéressantes, de façon à simplement les nommer à l'obtention d'une solution, plutôt que faire une recherche algorithmique des sous_termes voulus de la solution.

Les arguments du but appelé peuvent être un terme quelconque (sauf infini) et les valeurs à extraire sont des termes simples tels que entier, réel, double, identificateur ou chaîne.

8.5.1. Description

*int new_pattern(char * but)*

est une fonction avec en argument unique: une chaîne de caractères représentant un terme Prolog qui est le but à effacer, et où les résultats intéressants sont représentés par des formats. La fonction retourne 0 si tout c'est bien passé, un numéro d'erreur si une erreur s'est produite.
Les différents formats acceptés sont:

%d	pour les entiers
%s	pour les chaînes de caractères
%f	pour les réels simples
%F	pour les réels doubles
%i	pour les identificateurs sous forme d'entier
%I	pour les identificateurs sous forme de chaîne de caractères

Voici un exemple d'appels de *new_pattern*:

```
erreur = new_pattern("sys:enum(%d,-2,10)");
erreur = new_pattern(":essai(<%i,x,y>,%d,0)");
```

*int get_formats(int no, char *p1, char *p2, ...)*

est une fonction à nombre variable d'arguments où:
no, le 1er argument, est un entier qui indique le numéro d'ordre de la valeur à récupérer (numéro d'ordre du format concerné, dans la chaîne transmise à *new_pattern*), ou s'il vaut 0 indique que toutes les valeurs sont à récupérer.
Le ou les arguments suivants sont les adresses des variables C dans lesquelles seront mémorisées la ou les valeurs voulues, si ce ne sont pas des chaînes. Dans le cas où l'on attend une chaîne de caractères (formats %s ou %I), deux arguments sont nécessaires. Le premier des deux doit être un pointeur sur une zone mémoire dans laquelle Prolog pourra ranger la chaîne, le second doit être un entier qui indique la taille maximum de la chaîne. La fonction retourne 0 si tout c'est bien passé, un numéro d'erreur si une erreur s'est produite.
Voici pour chaque format le type de(s) variable(s) C associée(s):

format	type C	mode de passage

%d	*long*	*adresse*
%s	*char *, int*	*valeur*
%f	*float*	*adresse*
%F	*double*	*adresse*
%i	*long*	*adresse*
%I	*char *, int*	*valeur*

Voici un exemple d'appel de *get_formats*:

```
long i;
erreur = get_formats(0, &i);
```

La fonction *get_formats* va se charger de garnir l'entier *i* avec la valeur associée au premier format (l'entier *-2* dans notre exemple). A noter que cet appel est strictement équivalent à:

```
erreur = get_formats(1, &i);
```

8.5.2. Exemple

Voici un exemple d'utilisation des fonctions *new_pattern* et *get_formats*:

```
regles_test(recuperation, "recuperation", 1.2e0)->;
regles_test(des, "de chaînes", 4444.2e0)->;
regles_test(idents, "de", 0.2e0)->;
regles_test(Prolog, "caractères", 1.2e0)->;


int exemple2()
    {
    int erreur;
    long i;
    char s[100];
    float f;

    erreur = new_pattern("regles_test(%i,%s,%f)");
    if (erreur) return error;
    while (next_solution() == SOLUTION_EXISTS)
        {
        erreur = get_formats(0, &i, s, 100, &f);
        if (erreur) goto error;
        printf("%d %s %f\n",i,s,f);
        }

    kill_goal();
    return 0;
error:
    printf("erreur\n");
    kill_goal();
    return erreur;
   }
```

8.6. Autres fonctions

*ConnectDescriptors(EXTERNAL_DESCRIPTOR * paD[])*

Cette routine permet de déclarer dans le code de l'application le tableau de descripteurs *paD*, écrit comme on l'aurait fait pour une extension C directe. Ce tableau de descripteurs doit être persistant durant toute la session Prolog qui l'utilise (par exemple, qualifié de *static* s'il est en position de variable locale), et la déclaration doit être faite **avant** le début de la session (avant l'appel de *ProStart()*).
Si l'argument est NULL, les descripteurs seront supprimés dans la prochaine session. Si la routine est invoquée plusieurs fois, c'est le dernier tableau qui est pris en compte. La routine retourne 0 en cas de succès, -1 en cas d'échec (session déjà lancée).

*ConnectInString(InStringFunction * pfIS)*

Cette routine permet de redéfinir la fonction de lecture dans la console, par la déclaration d'une fonction d'entrée de texte *pfIS()* à laquelle Prolog soumettra toutes les entrées (prédicats). Le remplacement et la suppression de cette fonction fonctionnent comme la précédente. Cette routine retourne la foncion qui était installée avant son appel.
Le format de la fonction de lecture est imposé: son premier argument est l'adresse d'un buffer prêt à recevoir le texte (donc alloué), son second argument est la capacité maximale de ce buffer. Le code de retour de la fonction est ignoré: en cas d'erreur, elle doit rendre une chaîne vide dans le buffer.
Intérieurement, la fonction peut effectuer toute opération même bloquante (en traitant les événements) nécessaire pour obtenir le texte à retourner.

*ConnectOutString(OutStringFunction * pfOS)*

Cette routine permet de redéfinir la fonction d'écriture dans la console, par la déclaration d'une fonction de sortie de texte *pfOS()* à laquelle Prolog soumettra toutes les sorties (prédicats, messages). Le remplacement et la suppression de cette fonction fonctionnent comme la précédente. Cette routine retourne la foncion qui était installée avant son appel.
Le format de la fonction d'écriture est imposé: son unique argument est l'adresse d'un buffer contenant le texte à imprimer. Le code de retour de la fonction est ignoré: aucune erreur n'est attendue.
Intérieurement, la fonction peut effectuer toute opération même bloquante (en traitant les événements) nécessaire pour afficher le texte. Toutefois, on remarquera que Prolog peut émettre des lignes vides, et donc un filtrage peut être nécessaire si par exemple des boîtes de messages sont utilisées.

*get_error_complement(int lgmax, char *str, int in_external_code)*

Cette routine permet d'obtenir sous forme de chaîne de caractères le terme correspondant à l'éventuel complément d'erreur remonté par l'exécution d'un but. Elle doit être appelée avant la fonction *kill_goal()*. La variable *str* (chaîne de caractères) doit avoir été allouée auparavant et doit être d'une longueur supérieure ou égale à l'entier *lgmax* passé en premier paramètre. Si le booléen *in_external_code* est non nul, cela signifie que l'on désire obtenir la chaîne en code caractère externe.

9. Interruptions

9.1. Concepts
9.2. Description des interfaces
9.3. Exemple complet

On décrit dans ce chapitre comment lier des programmes Prolog à la survenue d'événements asynchrones tels que des interruptions matérielles. Lorsqu'une interruption survient, le programme Prolog courant est automatiquement suspendu, et la fonction gérant l'interruption est activée dès que la machine Prolog se trouve dans un état où l'on peut réaliser des appels croisés (c'est à dire à la prochaine inférence). Lorsque la fonction se termine, la démonstration suspendue reprend son cours normalement. Il est possible de communiquer entre les deux environnements par l'intermédiaire des données statiques (base de faits, tableaux, assignations).

Ce mécanisme permet donc de suspendre l'exécution en cours et d'empiler l'activation d'une nouvelle machine Prolog pour traiter l'interruption.

9.1. Concepts

La machine Prolog possède un vecteur de bits d'événements externes. Les huit premiers bits (bits 0 à 7) sont réservés pour des événements définis par l'utilisateur. A chaque bit doit être associée une fonction C pour la construction des arguments et l'appel d'un but Prolog. Cette fonction C est installée par l'utilisateur en appelant la procédure *pro_signal*.

L'interruption d'un programme Prolog ne peut être servie qu'en certains états précis tels que la fin d'une unification, ou à l'intérieur d'un prédicat évaluable. Cette interruption doit donc être mémorisée jusqu'à ce que la machine Prolog soit en état de la traiter. Ceci est réalisé par la procédure *send_prolog_interrupt* qui doit être appelée par le gestionnaire d'interruption pour signaler à Prolog cet événement.

Pour lier un programme Prolog à un événement asynchrone donné, il faut donc réaliser les opérations suivantes :

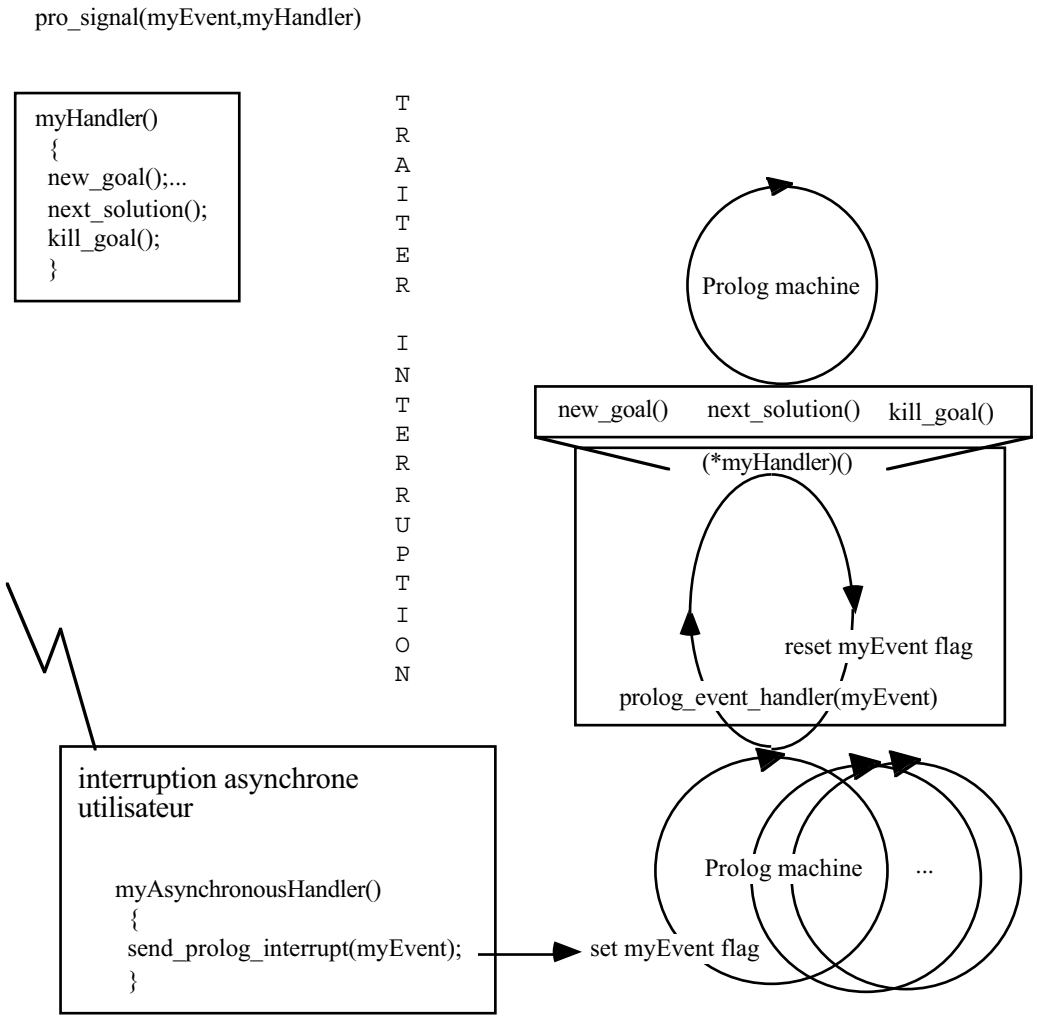
1. Choisir un bit (0 à 7) et le représenter par un masque (par exemple *myEvent = (1 << numéro du bit)*).

2. Ecrire une fonction (par exemple *myHandler*) qui servira à activer le programme Prolog gérant l'interruption.

3. Associer, dans l'environnement Prolog, l'événement *myEvent* avec la fonction le traitant en appelant la fonction :
   pro_signal(myEvent,myHandler).

4. Dans la routine appelée lors de l'interruption, transmettre l'événement à Prolog en appelant la fonction :
   send_prolog_interrupt(myEvent).

Un exemple de séquence d'événements et de traitements peut être représenté par le schéma suivant :



9.2. Description des interfaces

Les fonctions définies ici permettent de lier Prolog II+ à des procédures d'interruption. Les prototypes de ces fonctions sont donnés à la fin de ce chapitre.

pro_signal(event_mask,event_handler)

Permet d'associer à l'événement *event_mask* la fonction d'activation *event_handler*. Cette fonction renvoie -1L en cas d'erreur (notamment s'il existe déjà une fonction associée à *event_mask*), NULL sinon. Si le deuxième argument est NULL, la fonction d'activation associée au premier argument est supprimée et sa valeur est renvoyée comme résultat de la fonction. L'argument *event_mask* doit être une des valeurs parmi 1, 2, 4, 8, 16, 32, 64, 128.

Lorsque la fonction *event_handler* est appelée, si son résultat *n* est supérieur à 0, l'erreur Prolog correspondante est déclenchée.

send_prolog_interrupt(event_mask)

Déclenche le traitement associé à l'interruption *event_mask* dès que la machine Prolog le permet.

Conventions pour l'appel de la fonction d'activation *event_handler*.

Cette fonction doit retourner un entier qui est interprété comme un code erreur Prolog si cet entier est positif.

Lorsque plusieurs interruptions sont en attente, elles sont traitées dans l'ordre défini par les numéros des bits (bit 0 d'abord).

Il est déconseillé d'utiliser les fonctions *get_strterm* et *put_strterm* si le temps est critique, ces fonctions étant relativement longues d'exécution.

Prototypes des fonctions d'interface

```
long  pro_signal(event_mask,event_handler)
    unsigned short event_mask;
    int (*event_handler)();

void send_prolog_interrupt(event_mask)
   unsigned short event_mask;
```

9.3. Exemple complet

Nous décrirons dans cette section, à titre d'exemple, un cas précis construit pour le système UNIX et utilisant la primitive système *kill* pour envoyer des signaux à la tâche Prolog à partir du *Shell*.

Pour mettre en oeuvre cet exemple, suivre les étapes suivantes :

1. Ecrire l'extension Prolog pour réagir au signal SIGUSR1[1] : 30 et y associer la règle *message* dans le fichier *test.c*.

```
#include <signal.h>
#include "proext.h"
#define MYEVENT 8

extern long pro_signal();
extern void send_prolog_interrupt();
extern P_SYMBOL pro_symbol();
```

[1]C'est un signal asynchrone que l'on peut générer hors Prolog, depuis un autre processus UNIX. UNIX est un système multitâches, tous les programmes qui s'exécutent sont des processus.

```
static int my_P_Handler()
   {
   int err=0;
   new_goal();
   put_strterm(1,":message",&err);
   if (!err)
      err = next_solution();
   kill_goal();
   return err>0 ? err : 0 ;
   }

static void my_C_signal_handler()
   {
   send_prolog_interrupt(MYEVENT);
   }
      /* Cet appel pourrait être mis dans le programme
      principal pour faire l'initialisation automatiquement.
      Le tableau de descripteurs n'est alors plus nécessaire
      */
static long install_handler()
   {
   pro_signal(MYEVENT,my_P_Handler);
   signal(SIGUSR1,my_C_signal_handler);
   return 0;
   }

EXTERNAL_DESCRIPTOR testTable[] =
   {{":install_handler", C_FUNCTION, 0,
                        (POINTER) install_handler},
    { 0, 0, 0, 0 }};
```

2. Compiler *test.c* et faire l'édition de lien avec Prolog[1]. Ajouter dans Prolog le programme pour traiter les interruptions.

```
$ cc -c test.c[2]
$ prolink test.o testTable
$ prolog
...
> insert;
message -> val(counter,n) outl(n);
count(N) ->
      val(mod(N+1,1000000),N1)
      assign(counter,N1)
      count(N1);
;
> install_handler count(0);
```

3. Se connecter sur une autre console, déterminer le numéro de processus (pid) de Prolog avec la commande *ps*, et envoyer le signal 30 à Prolog. A chaque envoi du signal, la valeur courante du compteur s'imprime sur la console Prolog.

---

[1]Voir le manuel d'utilisation § 2.8 pour trouver les instructions de compilation et d'édition de lien avec Prolog, adaptées à votre système.

[2]Commande pour créer le module objet, compilé de *test.c*.

```
shell$ ps -ax

....2525...Prolog
shell$ kill -30 2525[1]
shell$ kill -30 2525
```

```
> install_handler
  count(0);

64790
301267
```

[1]Cette commande envoie le signal SIGUSR1 (auquel doit réagir Prolog) au processus Prolog.

10. Extensions Edinburgh

10.1. Syntaxe
10.2. Le contrôle
10.3. Manipulation des règles
10.4. Opérations prédéfinies sur les données
10.5. Les entrées / sorties
10.6. L'environnement
10.7. Traduction de DCG

Ce chapitre décrit les possibilités supplémentaires présentes lorsque le module de compatibilité Edinburgh est chargé.

Le plan de ce document suit le plan du manuel de référence en décrivant pour chaque rubrique les spécificités du mode Edinburgh.

Un certain nombre de règles prédéfinies dupliquent sous un autre nom les règles prédéfinies Prolog II+.

Toutes les fonctionnalités Prolog II+ sont également disponibles en syntaxe Edinburgh.

IMPORTANT: Dans ce chapitre, tous les atomes donnés en position d'argument ont le préfixe vide (""). Dans l'écriture des exemples, il est supposé que l'option permettant que "tous les atomes en position d'argument non préfixés explicitement ont le préfixe vide" est activée. Cette option est décrite en annexe A. Si cette option n'est pas activée, il faut avoir à l'esprit que les atomes dont la représentation abrégée est déjà connue du système doivent être explicitement préfixés avec le préfixe vide, sinon c'est le préfixe *sys* qui sera pris par défaut (du moins dans le contexte *user*). C'est notamment le cas des atomes *debug*, *string, fail* (utilisés dans le prédicat *set_prolog_flag/2*), *true*, *false* (utilisés par exemple dans le prédicat *write_term/2*) et *input* , *output*, *read*, *write* (pour *stream_property/2* ou *open/4*).

10.1. Syntaxe

10.1.1. Généralités

La syntaxe Edinburgh est acceptée en entrée dans l'un des cas suivants:

1) à l'activation de Prolog II +
   a) si l'on utilise l'option *-E* (équivalent de *-m edinburg.mo*)
   b) si l'on utilise un état binaire sauvé alors que la syntaxe Edinburgh était active,

2) sur la ligne de commande Prolog II+

a) si le fichier *edinburg.mo* est chargé (par le prédicat *load* ou *reload*).
b) si le but *edinburgh* est tapé,

La syntaxe exacte est décrite au chapitre 1 en utilisant la variante E des règles syntaxiques. Les principales différences avec la syntaxe Edinburgh définie dans la norme ISO/IEC sont décrites en annexe A.

Sur la ligne de commande ou dans une directive, lorsque le premier terme rencontré est une liste, celle-ci est interprétée comme une liste de fichiers à consulter. *user* est équivalent à *"console"*.

10.1.2. Les opérateurs

En syntaxe Edinburgh , les opérateurs suivants sont prédéfinis:

opérateur	précédence	type	terme construit
:-	1200	xfx	sys:':-'(T1,T2)
:-	1200	fx	sys:':-'(T1)
?-	1200	fx	sys:'?-'(T1)
-->	1200	xfx	sys:'-->'(T1,T2)
;	1100	xfy	sys:';'(T1,T2)
->	1050	xfy	sys:'->'(T1,T2)
,	1001	xfy	sys:','(T1,T2)
\+	900	fy	sys:not(T1)
is	700	xfx	sys:is(T1,T2)
=	700	xfx	sys:'='(T1,T2)
<	700	xfx	sys:'<'(T1,T2)
>	700	xfx	sys:'>'(T1,T2)
>=	700	xfx	sys:'>='(T1,T2)
=<	700	xfx	sys:'=<'(T1,T2)
@<	700	xfx	sys:'@<'(T1,T2)
@>	700	xfx	sys:'@>'(T1,T2)
@>=	700	xfx	sys:'@>='(T1,T2)
@=<	700	xfx	sys:'@=<'(T1,T2)
\=	700	xfx	sys:'\='(T1,T2)
==	700	xfx	sys:'=='(T1,T2)
=\=	700	xfx	sys:'=\='(T1,T2)
\==	700	xfx	sys:'\=='(T1,T2)
=..	700	xfx	sys:'=..'(T1,T2)
=:=	700	xfx	sys:'=:='(t1,t2)
+	500	yfx	sys:'+'(t1,t2)
-	500	yfx	sys:'-'(t1,t2)
/\	500	yfx	sys:'/\'(T1,T2)
\/	500	yfx	sys:'\/'(T1,T2)
mod	400	yfx	sys:mod(t1,t2)
rem	400	yfx	sys:rem(t1,t2)

opérateur	précédence	type	terme construit
*	400	yfx	sys:'*'(t1,t2)
/	400	yfx	sys:'/'(t1,t2)
<<	400	yfx	sys:'<<'(T1,T2)
>>	400	yfx	sys:'>>'(T1,T2)
//	400	yfx	sys:'//'(T1,T2)
\	200	fy	sys:'\'(t1,t2)
**	200	xfx	sys:'**'(t1,t2)
+	200	fy	sys:'+'(t1)
-	200	fy	sys:'-'(t1)
^	200	xfy	sys:'^'(T1,T2)

current_op(P,M,O)

Unifie respectivement les arguments *P*, *M* et *O* avec la précédence, le type de parenthèsage et l'opérateur. Exemple:

```
?- current_op(1200,M,O).
{M=fx,O=?-}
{M=fx,O=:-}
{M=xfx,O=:-}
{M=xfx,O=-->}
```

10.2. Le contrôle

\+ *X* ou *\+(X)*

Equivalent à *not(X)*.

X =Y ou *=(X,Y)*

Succès si *X* est unifiable avec *Y*, sinon échec. Est défini par la règle:

```
X = X.
```

X \= Y ou *\=(X,Y)*

Succès si *X* n'est pas unifiable avec *Y*, sinon échec. Exemples:

```
?- 1 \= 1.
?- X \= 1.
?- X \= Y.
?- 1 \= 1.0.
{}
```

X -> Y ou *->(X,Y)*

Si-alors. Est défini par la règle (les parenthèses ne servent qu'à améliorer la lisibilité):

```
(X -> Y) :- X,!,Y.
```

X , Y ou *,(X,Y)*

"Et" propositionnel. *X* est exécuté puis *Y*.

X ; Y ou *;(X,Y)*

"Ou" propositionnel. Est défini par les règles suivantes (Note: en Prolog II la coupure est transparente):

```
X;Y :- X.
X;Y :- Y.
```

call(X)

Est défini par la règle:

```
call(X) :- X.
```

catch(G,C,R)

Lance l'exécution du but *G*. Fait un succès dans deux cas:
- l'exécution de *G* réussit sans interruption de type "*throw*"
- une interruption de type "*throw*" dont l'argument s'unifie avec l'argument *C* se produit et l'exécution du but *R* réussit. Si l'unification échoue, l'interruption est propagée selon le mécanisme de *block*/*block_exit*. Exemples:

```
?- reconsult(user).
foo(X) :- Y is X*2, throw(test(Y)).
bar(X) :- X=Y, throw(Y).
coo(X) :- throw(X).
car(X) :- X=1, throw(X).
g :-    catch(p, B, write(h2)), coo(c).
p.
p :- throw(b). .
{}
?- catch(foo(5),test(Y),true).
{Y=10}
?- catch(bar(3),Z,true).
{Z=3}
?- catch(true,C,write(foo)), throw(bla).
-> bla([])   'block_exit' SANS 'block' CORRESPONDANT

?- catch(coo(X),Y,true).
{}
?- catch(car(X),Y,true).
{X=1,Y=1}
?- catch(g, C, write(h1)).
h1{C=c}
```

fail_if(X)

Est équivalent à l'appel de *not(call(X))*.

```
?- fail_if(true).
?- fail_if(4=5).
{}
```

once(G)

Efface le but *G* de la première manière possible. Est défini par la règle:

```
once(G) :- G, ! .
```

throw(X)

Est équivalent à un appel de *block_exit(X1)* , *X1* étant une copie du terme *X* (renommage des variables).

true

S'efface toujours avec succès.

unify_with_occurs_check(X, Y)

Tente l'unification des termes X et Y avec une vérification de non production d'arbres infinis. Echoue si l'unification échoue ou si elle génère un arbre infini. Exemples:

```
?- unify_with_occurs_check(1,1).
{}
?- unify_with_occurs_check(1,2).
?- unify_with_occurs_check(X,1).
{X=1}
?- unify_with_occurs_check(X,a(X)).
?- unify_with_occurs_check(X,[1|X]).
?- unify_with_occurs_check(X,[X|1]).
?-
```

10.3. Manipulation des règles

abolish(P)

Se comporte comme *suppress(P)* où *P* est un identificateur ou un terme de la forme identificateur/arité.

assert(X)
assertz(X)

Ajoute en queue de paquet la règle ou le fait *X*. Voir *assert* décrit pour Prolog II. Attention, en syntaxe Edinburgh, la virgule est à la fois un séparateur et un opérateur, il faut donc parenthéser l'argument dans le cas des règles:

```
?- assert(a).
{}
?- assert((a :- b,c)).
{}
```

asserta(X)

Ajoute en tête de paquet la règle ou le fait *X*. Voir *asserta* décrit pour Prolog II. Même remarque sur la virgule que ci dessus.

clause(T,Q)

T ne doit pas être une variable libre. Unifie *T* et *Q* respectivement avec la tête et la queue de toutes les clauses dont l'accès est défini par *T*. Se comporte comme *rule* avec la différence qu'une queue vide est représentée par *true*, et qu'une queue d'au moins deux littéraux est représentée par une structure avec le noeud ','.

consult(F)
reconsult(F)

Est équivalent respectivement à *insert* et *reinsert*. Si *F* est un identificateur, c'est la chaîne correspondant à son abréviation qui est prise en compte comme nom de fichier (c.à.d. le préfixe est ignoré). Si *F* vaut *user* ou *"user"* lit sur l'unité courante de lecture.

listing
listing(X)

Est équivalent respectivement à *list* et *list(X)*.

retract(X)

Même comportement que *retract* à deux arguments. *X* doit être un fait ou une règle dont le prédicat de tête est instancié.
retract((T:- true)) est équivalent à *retract(T)*,
retract(T) est équivalent à *retract(T,[])*, si *T* ne s'unifie pas avec *:- (T1,T2)*,
retract((T :- Q)) est équivalent à *retract(T,Q)*.

retractall(X)

Toutes les règles dont la tête s'unifie avec *X* sont supprimées (sans unification de *X* ni backtracking). *X* doit avoir la forme d'une tête de règle, sinon, le prédicat n'aura aucun effet. Fait toujours un succès.

10.4. Opérations prédéfinies sur les données

10.4.1. Les tests de type

atom(X)

Succès si *X* est un identificateur, sinon échec. Equivalent à *ident(X)*.

atomic(X)

Succès si *X* est une constante, sinon échec.

compound(X)

Succès si *X* est une liste ou un n-uplet, sinon échec.

float(X)

Est équivalent à *real(X)*.

nonvar(X)

Est équivalent à *bound(X)*.

number(X)

Succès si *X* est un entier ou un réel, sinon échec.

var(X)

Succès si *X* est une variable libre, sinon échec. Est équivalent à *free(X)*.

10.4.2. Les opérations arithmétiques

L'évaluation d'une expression est faite au moyen de la règle prédéfinie *val*. Certaines expressions peuvent être utilisées en syntaxe Edinburgh directement comme des termes à effacer.

Les termes suivants peuvent être effacés, sans l'intermédiaire du prédicat *val*.

X =:= Y ou *=:=(X,Y)*

Est équivalent à *val(eql(X,Y),1)*.

X =\= Y ou *=\= (X,Y)*

Est équivalent à *val('=\='(X,Y),1)*.

X < Y ou *<(X,Y)*
X =< Y ou *=<(X,Y)*
X > Y ou *>(X,Y)*
X >= Y ou *>=(X,Y)*

Sont respectivement équivalents à:

val('<'(X,Y),1)
val('=<'(X,Y),1)
val('>'(X,Y),1)
val('>='(X,Y),1)

X is Y ou *is(X,Y)*

Est équivalent à *val(Y,X)*.

Les fonctions suivantes sont spécifiques au mode Edinburgh. Elles sont évaluables par les règles prédéfinies *val*, *tval* ou *is*. Elles doivent être appliquées à des arguments de type entier ou réel.

\X ou *\(X)*

valeur (*\X*) = valeur(*'~'(X)*) = complément bit à bit de *X*.
La valeur de *X* doit être de type entier. Le résultat est de type entier.

X // Y ou *//(X,Y)*

valeur(*//*(*X*, *Y*)) = division entière de valeur(*X*) par valeur(*Y*).
Le résultat est de type entier.

log(t)

valeur(*log*(*t*)) = valeur(*ln*(*t*)) = *logarithme népérien*(valeur(*t*)).
Le résultat est de type réel.

truncate(t)

valeur(*truncate*(*t*)) = valeur(*trunc*(*t*)) = conversion en entier de la valeur de *t*.
Le résultat est de type entier.

10.4.3. Composition et décomposition d'objets

X =.. *Y* ou =..*(X,Y)*

Si *X* est instancié, *Y* est unifié avec une liste dont le premier élément est le foncteur de *X*, et les éléments suivants les éventuels arguments dans l'ordre. Si *X* est libre, *Y* doit être instancié avec une liste; *X* est alors unifié avec le n-uplet construit avec le premier élément de la liste comme foncteur, et les autres éléments comme arguments. A un élément atomique correspond une liste constituée de ce seul élément. Exemples:

```
?- '=..'(foo(a,b),[foo,a,b]).
{}
?- '=..'(X,[foo,a,b]).
{X=foo(a,b)}
?- '=..'(foo(a,b),L).
{L=[foo,a,b]}
?- '=..'(foo(X,b),[foo,a,Y]).
{X=a,Y=b}
?- '=..'(1,[1]).
{}
?- '=..'(foo(a,b),[foo,b,a]).
?- '=..'(f(X),[f,u(X)]).
{X=_714, _714=u(_714)}
```

arg(N,T,X)

S'efface si *X* est le *N*ième élément du terme *T*. *N* entier et *T* tuple ou paire pointée, doivent être connus au moment de l'appel.

Si *T* est une paire pointée alors si

N=1, *X* est unifié avec le premier élément de la paire.

N=2, *X* est unifié avec le deuxième élément de la paire.

Si *N* est un entier et *T* est un n-uplet de taille M>1, alors si

0<*N*<M, *X* est unifié avec l' argument *N*+1 du n-uplet *T*.

(c.à.d. avec l'argument *N* de $T=f(a_1,...,a_n,...,a_{M-1})$).

Sinon échec.

Exemple :

```
?- arg(1,eq(john,fred), X).
{X=john}
?- arg(0,eq(john,fred), Y).
?-
```

atom_chars(A,L)

Associe à un identificateur *A* la liste *L* des caractères qui le constituent et vice-versa. Le préfixe de l'identificateur *A* est ignoré (en entrée) ou égal à "" (en sortie). Exemples:

```
?- atom_chars('',L).
{L=[]}
?- atom_chars('''', L).
{L=['''']}
?- atom_chars('ant',L).
{L=[a,n,t]}
?- atom_chars(Str, ['s', 'o', 'p']).
{Str=sop}
```

atom_codes(A,L)

Associe à un identificateur *A* la liste *L* des codes internes (codes ISO ou codes de la machine hôte) qui le constituent et vice-versa. Le préfixe de l'identificateur *A* est ignoré (en entrée) ou égal à "" (en sortie).

atom_concat(A1,A2,A3)

Fonctionne sur les identificateurs de la même manière que *conc_string* sur les chaînes de caractères. Le préfixe des identificateurs *A1, A2, A3* sont ignorés (en entrée) ou égaux à "" (en sortie). Exemples:

```
?- atom_concat('hello',' world',S3).
{S3='hello world'}
?- atom_concat(T,' world','small world').
{T=small}
?- atom_concat(T1,T2,'hello').
{T1='',T2=hello}
{T1=h,T2=ello}
{T1=he,T2=llo}
{T1=hel,T2=lo}
{T1=hell,T2=o}
{T1=hello,T2=''}
```

atom_length(A,N)

Unifie *N* avec la longueur de l'identificateur *A* . Le préfixe de l'identificateur *A* est ignoré. Exemples:

```
?- atom_length('enchanted evening',N).
{N=17}
?- atom_length('',N).
{N=0}
```

functor(T,F,N)

Associe à un arbre *T* son foncteur *F* et son arité *N*, et vice-versa.

```
?- functor(foo(a,b,c),X,Y).
{X=foo,Y=3}
?- functor(X,foo,3).
{X=foo(_515,_516,_517)}
?- functor(X,foo,0).
{X=foo}
?- functor(foo(a),foo,2).
?- functor(foo(a),fo,1).
?- functor(1,X,Y).
{X=1,Y=0}
?- functor(X,1.1,0).
{X=1.1}
?- functor([_|_],'.',2).
{}
?- functor([],[],0).
{}
```

name(X,L)

Si *X* est un identificateur ou un nombre, *L* est unifié avec la liste des codes internes (codes ISO ou codes de la machine hôte) des caractères constituant la représentation de l'identificateur *X*.

Si *L* est une liste de codes internes des lettres d'un identificateur, *X* est instancié avec l'identificateur déterminé par les conventions courantes de préfixage.

```
?- name(abc:def,L).
```

```
{L=[97,98,99,58,100,101,102]}
?- name(123,L).
{L=[49,50,51]}
?- name(X,[65,66|4]).
?- name("asd",X).
?- name(Y,X).
?- name(ab:'cd',L).
{L=[97,98,58,99,100]}
?- name(ab:' %',L).
{L=[97,98,58,39,32,37,39]}
?- name(I,[97,98,58,99,100]).
{I=ab:cd}
?- name(M,[97,98,58,39,32,37,39]).
{M=ab:' %'}
?-
```

number_chars(N,L)

Associe à un nombre *N* la liste *L* des caractères qui le constituent. Réciproquement, considère la liste *L* des caractères comme une entrée et y associe le nombre *N* (imprimé sous sa forme décimale). Exemples:

```
?- number_chars(33, L).
{L=['3','3']}
?- number_chars(33.0, L).
{L=['3','3','.','0']}
?- number_chars(X, ['3', '.', '3', 'E', '+', '0']).
{X=3.3}
?- number_chars(3.3, ['3', '.', '3', 'E', '+', '0']).
{}
?- number_chars(A, ['-', '2', '5']).
{A=-25}
?- number_chars(A, [' ', '3']).
{A=3}
?- number_chars(A, ['0', 'x', 'f']).
{A=15}
?- number_chars(A, ['0', '''', 'a']).
{A=97}
?- number_chars(A, ['4', '.', '2']).
{A=4.2}
?- number_chars(A,  ['4', '2', '.', '0', 'e', '-', '1']).
{A=4.2}
```

number_codes(N,L)

Même description que *number_chars/2* mais *L* est une liste de codes internes (codes ISO ou codes de la machine hôte).

phrase(X,Y)

phrase(X,Y,Z)

Essaie de décomposer la liste Y en une phrase de la grammaire et un reste. X doit être une tête de règle de grammaire, Y et Z peuvent être des variables libres ou des listes. Enumère les solutions possibles en unifiant le reste avec Z. La forme à deux arguments *phrase(X,Y)* équivaut à *phrase(X,Y,[])*.

Note: Si Y ou Z sont des chaînes de caractères, celles-ci sont d'abord transformées en listes de caractères avant de réaliser l'appel : *phrase(somme(X),"1+2+3")* est transformé automatiquement dans l'appel *phrase(somme(X), ["1","+","2","+","3"])*.

sub_atom(A1,N1,N2,N3,A2)

Fait un succès si l'identificateur *A1*, qui doit être connu lors de l'appel, peut être vu comme la concaténation de trois parties telles que *N1* soit la longueur de la première partie, *N2* la longueur de la seconde partie qui n'est rien d'autre que l'identificateur *A2*, et *N3* la longueur de la troisième partie. Les préfixes des identificateurs *A1* et *A2* sont ignorés (en entrée) ou égaux à "" (en sortie). Exemples:

```
?- sub_atom(abracadabra, 0, 5, A, S2).
{A=6,S2=abrac}
?- sub_atom(abracadabra, _, 5, 0, S2).
{S2=dabra}
?- sub_atom(abracadabra, 3, L, 3, S2).
{L=5,S2=acada}
?- sub_atom(abracadabra, B, 2, A, ab).
{B=0,A=9}
{B=7,A=2}
?- sub_atom('Banana', 3, 2, A, S2).
{A=1,S2=an}
?- sub_atom('charity', B, 3, A, S2).
{B=0,A=4,S2=cha}
{B=1,A=3,S2=har}
{B=2,A=2,S2=ari}
{B=3,A=1,S2=rit}
{B=4,A=0,S2=ity}
?- sub_atom('ab', Start, Length, A, Sub_atom).
{Start=0,Length=0,A=2,Sub_atom=''}
{Start=0,Length=1,A=1,Sub_atom=a}
{Start=0,Length=2,A=0,Sub_atom=ab}
{Start=1,Length=0,A=1,Sub_atom=''}
{Start=1,Length=1,A=0,Sub_atom=b}
{Start=2,Length=0,A=0,Sub_atom=''}
```

10.4.3. Comparaison de termes quelconques

X == *Y* ou ==*(X,Y)*

Succès si *X* est formellement égal à *Y*, sinon échec. Exemples:

```
?- 1 == 1.
{}
?- X == X.
{}
?- 1 == 2.
?- X == 1.
?- X == Y.
?- _ == 1.
?- _ == _.
?-
```

X \== *Y* ou \== *(X,Y)*

Succès si *X* n'est pas formellement égal à *Y*, sinon échec. Exemples:

```
?- 1 \== 1.
?- 1 \== 2.
{}
?- X \== 1.
{}
?- _ \== _.
{}
?-
```

X @< *Y*

Compare les termes *X* et *Y*, fait un succès si *X* précède *Y*, échoue sinon.

X @> *Y*

Compare les termes *X* et *Y*, fait un succès si *Y* précède *X*, échoue sinon.

X @>= *Y*

Compare les termes *X* et *Y*, fait un succès si *Y* précède ou est formellement égal à *X*, échoue sinon.

X @=< *Y*

Compare les termes *X* et *Y*, fait un succès si *X* précède ou est formellement égal à *Y*, échoue sinon. Quelques exemples:

```
?- 1.0 @< 1.
{}
?- @<(aardvark,zebra).
{}
?- @<(short,short).
?- short @< shorter.
{}
?- @<(foo(a),foo(b)).
{}
?- @<(foo(a,b),north(a)).
?- @<(X,X).
?- Y @< X.
{}
?- @<(_,_).
{}
?- @<(foo(X,a),foo(Y,b)).
{}
?- "foo" @> foo.
{}
?- [1,2,3] @> [1,1,3,4].
{}
?- [1,2] @> <>(1,2).
?- [1,2] @> <>(X).
{}
```

10.5. Les entrées / sorties

10.5.1. Généralités

A l'ouverture d'une unité d'entrée/sortie (prédicats *open/3* et *open/4)*, un argument de sortie, qui doit obligatoirement être une variable libre lors de l'appel, est unifié avec une constante (en l'occurence un entier négatif). Cette constante permettra dans les prédicats de désigner de manière unique le canal d'accès à l'unité. Une option d'ouverture permet d'associer à l'unité un identificateur, appelé "alias", qui sera dans les sources de programmes un moyen plus clair et plus global de désigner ce numéro de canal.

close(C,O), close(C)

Ferme l'unité associée au canal *C*. L'argument *O*, s'il existe, doit être une liste d'options prises parmi:

force(true): une erreur sur la fermeture de l'unité sera remontée.
force(false): une erreur sur la fermeture de l'unité sera ignorée.

open(S,M,C,O), *open(S,M,C)*

Ouvre selon le mode *M*, l'unité ayant pour nom l'atome *S* et renvoie le canal associé *C*. L'argument *O*, s'il existe, doit être une liste d'options prises parmi:

- *alias(A)* où *A* doit être un identificateur et indiquera un alias par lequel l'unité pourra être nommée dans les prédicats.
- *eof_action(A)* qui indique l'action à accomplir sur une fin de fichier suivant la valeur de *A*:
  - *error*: une erreur est générée (c'est le défaut).
  - *eof_code*: un code spécial, dépendant du prédicat de lecture, est rendu à chaque tentative d'accès au fichier (Voir les prédicats de lecture).
  - *reset*: le même code que dans le cas de *eof_code* est rendu et la lecture de la fin de fichier est annulée. La valeur de la propriété *end_of_stream* reste *at*. Ceci est utile pour des terminaux.
- *lg_buffer(L)* où *L* doit être un entier et indiquera la taille du buffer associé à l'unité.
- *reposition(B)* où *B* peut prendre la valeur *:true* ou *:false* et indique si l'index de lecture (ou d'écriture) peut être repositionné (appel à la primitive *set_stream_position/2*). Défaut: *true*.
- *type(T)* où *T* désignera le type de l'unité.

Le canal *C* associé à l'unité doit être une variable libre lors de l'appel et est unifié avec une valeur entière négative (transparent pour l'utilisateur). Exemple:

```
?- open(bfile,write,X,[alias(bfile),type(binary)]).
{X=-7}
```

set_stream_position(C,N)

Positionne au *Nième octet* le pointeur de lecture ou d'écriture de l'unité associée au canal *C*. *N* doit être un entier et l'unité doit être un fichier disque.

stream_property(C,P)

Réussit si l'unité associée au canal *C* a la propriété *P*. *C* peut être une variable libre, auquel cas tous les canaux ayant la propriété *P* seront unifiés successivement avec *C*. *P* peut soit être libre, soit indiquer une propriété prise parmi:

- *alias(A)* où *A* sera unifié avec l'alias de l'unité.
- *file_name(S)* où *S* sera unifié avec le nom de l'unité.
- *input* qui sera vrai si l'unité est une unité d'entrée.
- *mode(M)* où *M* sera unifié avec le mode d'ouverture de l'unité.
- *output* qui sera vrai si l'unité est une unité de sortie.
- *position(P)* où *P sera* unifié avec la position du pointeur de l'unité.
- *type(T)* où *T* sera unifié avec le type de l'unité.

- *reposition(B)* où *B* sera unifié avec la valeur *true* ou *false* suivant l'option choisie à l'ouverture de l'unité.
- *eof_action(A)* où *A* sera unifié avec l'action à accomplir sur une fin de fichier.
- *end_of_stream(E)* où *E* sera unifié avec:
    - *not* si l'on n'est pas sur une fin de fichier.
    - *at* si l'on se trouve sur une fin de fichier.
    - *past* si l'on a dépassé la fin de fichier (une lecture a encore été tentée après une fin de fichier signalée).

C ne peut pas être un alias car il peut aussi être utilisé en sortie.

Exemples

```
?- stream_property(S,P).
{S=-6,P=file_name(foo)}
{S=-7,P=file_name(bfoo)}
{S=-6,P=mode(append)}
{S=-7,P=mode(read)}
{S=-6,P=alias(momo)}
{S=-7,P=alias(bfile)}
{S=-6,P=type(text)}
{S=-7,P=type(binary)}
{S=-7,P=input}
{S=-6,P=output}
{S=-6,P=position(69)}
{S=-7,P=position(4)}
{S=-6,P=reposition(:true)}
{S=-7,P=reposition(false)}
{S=-6,P=eof_action(error)}
{S=-7,P=eof_action(eof_code)}
{S=-6,P=end_of_stream(:not)}
{S=-7,P=end_of_stream(:not)}
?- stream_property(S,type(text)).
{S=-6}
?- stream_property(-7,type(B)).
{B=binary}
```

10.5.2. Entrées

10.5.2.1. Interprétation des chaînes de caractères

Un texte en position d'argument noté entre doubles quotes peut être interprété par l'analyseur **syntaxique** de différentes manières suivant l'option choisie (sur la ligne de commande ou par exécution du prédicat *set_prolog_flag/2*):

- Soit comme une vraie chaîne de caractères (type à part entière).
- Soit comme un atome: il y aura alors une équivalence avec l'écriture entre simples quotes.
- Soit comme une liste composée des caractères (atomes préfixés par le préfixe vide) constituant la chaîne.
- Soit comme une liste composée des codes des caractères constituant la chaîne.

Exemples:

```
?- set_prolog_flag(double_quotes,string), read(R).
"hello world".
{R="hello world"}
?- set_prolog_flag(double_quotes,chars), read(R).
```

```
"hello world".
{R=[h,e,l,l,o,' ',w,o,r,l,d]}
?- set_prolog_flag(double_quotes,codes), read(R).
"hello world".
{R=[104,101,108,108,111,32,119,111,114,108,100]}
?- set_prolog_flag(double_quotes,atom), read(R).
"hello world".
{R='hello world'}
?-
```

10.5.2.2. Prédicats

Si l'argument désignant l'unité d'entrée est un argument en entrée, on pourra utiliser indifféremment le n° de canal ou bien l'alias associé à cette l'unité. Dans le cas contraire (cas du prédicat *current_input/1)*, c'est le n° de canal qui sera rendu.

La plupart des prédicats d'entrée peuvent s'utiliser soit sur l'unité spécifiée en argument (n° de canal, alias), soit sur l'unité courante d'entrée, auquel cas l'argument indiquant l'unité est absent.

Le comportement des prédicats de lecture, quand une fin de fichier est rencontrée, est conforme à l'option *eof_action* choisie à l'ouverture de l'unité sur laquelle se fait la lecture.

at_end_of_stream(C), *at_end_of_stream*

Réussit si une lecture sur le canal *C* a déjà signalé une fin de fichier, échoue sinon. Contrairement au prédicat *eof*, ce prédicat n'essaie pas d'effectuer une nouvelle lecture sur le canal C mais se contente d'examiner le status de la lecture précédente.

current_input(C)

C est unifié avec le canal associé à l'unité d'entrée courante. Attention: *C* ne sera pas un alias.

get(C,X), get(X)

X est unifié avec l'entier égal au code interne (code ISO ou code de la machine hôte) du premier caractère non blanc lu sur l'unité d'entrée associée au canal *C*. Si une fin de fichier est rencontrée et que l'unité a été ouverte avec l'option *eof_action(eof_code)*, *X* est unifié avec la valeur -1.

get_byte(C,X), *get_byte(B)*

X est unifié avec le premier octet (entier >= 0) lu sur l'unité d'entrée associée au canal *C*. Cette unité doit être de type *:binary*. Si une fin de fichier est rencontrée et que l'unité a été ouverte avec l'option *eof_action(eof_code)*, *X* est unifié avec la valeur -1.

get_char(C,X), get_char(X)

X est unifié avec le premier caractère lu sur l'unité d'entrée associée au canal *C*. Si une fin de fichier est rencontrée et que l'unité a été ouverte avec l'option *eof_action(eof_code)*, *X* est unifié avec la valeur *end_of_file*.

get_code(C,X), get_code(X)

X est unifié avec l'entier égal au code interne (code ISO ou code de la machine hôte) du premier caractère lu sur l'unité d'entrée associée au canal *C*. Si une fin de fichier est rencontrée et que l'unité a été ouverte avec l'option *eof_action(eof_code)*, *X* est unifié avec la valeur -1.

get0(X)

Identique à *get_code/1*.

peek_byte(C,X), peek_byte(X)

Essaie d'unifier *X* avec le premier octet (entier >= 0) en entrée sur l'unité associée au canal *C*. Cette unité doit être de type *:binary*. L'octet n'est pas lu. Ce prédicat ne modifie donc pas les propriétés *position* et *end_of_stream* de l'unité. Si une fin de fichier est rencontrée et que l'unité a été ouverte avec l'option *eof_action(eof_code)*, *X* est unifié avec la valeur -1.

peek_char(C,X), peek_char(X)

Essaie d'unifier *X* avec le premier caractère en entrée sur l'unité associée au canal *C*. Le caractère n'est pas lu. Ce prédicat ne modifie donc pas les propriétés *position* et *end_of_stream* de l'unité. Si une fin de fichier est rencontrée et que l'unité a été ouverte avec l'option *eof_action(eof_code)*, *X* est unifié avec la valeur *end_of_file.*

peek_code(C,X), peek_code(X)

Essaie d'unifier *X* avec l'entier égal au code interne (code ISO ou code de la machine hôte) du premier caractère en entrée sur l'unité associée au canal *C*. Le caractère n'est pas lu. Ce prédicat ne modifie donc pas les propriétés *position* et *end_of_stream* de l'unité. Si une fin de fichier est rencontrée et que l'unité a été ouverte avec l'option *eof_action(eof_code)*, *X* est unifié avec la valeur -1.

read(C,X), read(X)

Lit le prochain terme se trouvant sur l'unité associée au canal *C*, plus le premier caractère non blanc qui le suit. Equivalent à *in(X,_)* sur l'unité *C*. Si une fin de fichier est rencontrée et que l'unité a été ouverte avec l'option *eof_action(eof_code)*, *X* est unifié avec la valeur *end_of_file.*
Exemple:

```
?- read(X).
[1,2].
{X=[1,2]}
```

read_line(C,S,N), read_line(S,N), read_line(S)

Equivalent au prédicat *inl(S,N)* sur l'unité associée au canal *C*.

read_term(C,X,O), read_term(X,O)

Même chose que le prédicat *read*, mais l'on peut en plus obtenir une liste d'options prises parmi:

- *variables(L_V)* où *L_V* sera la liste des variables du terme *X*.

- *variable_names(L_VN)* où *L_VN* sera le dictionnaire des variables du terme *X*. Ce dictionnaire sera composé d'éléments de la forme: "nom de la variable" = variable.
- *singletons(L_S)* où *L_S* sera le dictionnaire des variables qui n'apparaissent qu'une seule fois dans le terme *X*. Ce dictionnaire sera composé d'éléments de la forme: "nom de la variable" = variable.

Exemple:

```
?- read_term(T,[variables(L_V),variable_names(L_VN),
   singletons(S)]).
aa(X,[Y,X,Y,Z],<>(Z,U)).
{T=aa(_940,[_969,_940,_969,_1042],_1042(_1208)),
   L_V=[_940,_969,_1042,_1208],
   L_VN=["X" = _940,"Y" = _969,"Z" = _1042,
         "U" = _1208],
   S=["U" = _1208]}
```

see(F)

Est équivalent à *input(F)*. Si F est un identificateur, c'est la chaîne correspondant à son abréviation qui est prise en compte comme nom d'unité (c.à.d. le préfixe est ignoré).

seeing(F)

Est équivalent à *input_is(F)*. Si F est un identificateur, c'est la chaîne correspondant à son abréviation qui est prise en compte comme nom d'unité (c.à.d. le préfixe est ignoré).

seen

Est équivalent à *close_input.*

set_input(C)

Redirige l'entrée courante vers l'unité associée au canal *C.*

10.5.3. Sorties

Si l'argument désignant l'unité de sortie est un argument en entrée, on pourra utiliser indifféremment le n° de canal ou bien l'alias associé à cette l'unité. Dans le cas contraire (cas du prédicat *current_output/1)*, c'est le n° de canal qui sera rendu.

La plupart des prédicats de sortie peuvent s'utiliser soit sur l'unité spécifiée en argument (n° de canal, alias), soit sur l'unité courante de sortie, auquel cas l'argument indiquant l'unité est absent.

current_output(C)

C est unifié avec le canal associé à l'unité de sortie courante. Attention: *C* ne sera pas un alias.

flush_output(C), flush_output

Equivalent au prédicat *flush/0* pour l'unité de sortie associée au canal *C*.

nl(C), nl

Est équivalent à *line/0* pour l'unité de sortie associée au canal *C*.

put(X)

Equivalent à *put_code/1*.

put_byte(C,X), put_byte(X)

L'octet *X* (entier >= 0) est envoyé sur l'unité de sortie associée au canal *C*. Cette unité doit être de type *:binary*.

put_char(C,X), put_char(X)

Le caractère *X* est envoyé sur l'unité de sortie associée au canal *C*.

put_code(C,X), put_code(X)

Le code interne de caractère *X* est envoyé sur l'unité de sortie associée au canal *C*.

set_output(C)

Redirige la sortie courante vers l'unité associée au canal *C*.

tab(N)

Envoie *N* blancs sur l'unité courante de sortie.

tell(F)

Est équivalent à *output(F)*. Si F est un identificateur, c'est la chaîne correspondant à son abréviation qui est prise en compte comme nom d'unité (c.à.d. le préfixe est ignoré).

telling(F)

Est équivalent à *output_is(F)*. Si F est un identificateur, c'est la chaîne correspondant à son abréviation qui est prise en compte comme nom d'unité (c.à.d. le préfixe est ignoré).

told

Est équivalent à *close_output*.

write(C,X), write(X)

Est équivalent à *write_term* avec les options *[quoted(false), numbervars(true), ignore_ops(false)]*.

writeq(C,X), writeq(X)

Est équivalent à *write_term* avec les options *[quoted(true), numbervars(true), ignore_ops(false)]*.

write_canonical(C,X), write_canonical(X)

Est équivalent à *write_term* avec les options *[quoted(true), numbervars(false), ignore_ops(true)]*.

write_term(C,X,O), write_term(X,O)

Ecrit le terme *X* sur l'unité de sortie associée au canal *C*, en tenant compte de la liste d'options O, options prises parmi:

- *quoted(true)* ou *quoted(false)*: indique si les constantes chaînes et les identificateurs contenus dans *X* doivent être écrits sous forme quotée.
- *ignore_ops(true)* ou *ignore_ops(false)*: indique si les déclarations courantes d'opérateurs doivent être ignorées.
- *numbervars(true)* ou *numbervars(false)*: indique si les termes de la forme *'$VAR'(N)*, *N* étant un entier positif doivent être écrits comme une variable constituée d'une lettre majuscule suivie d'un entier. La lettre est la (*i*+1)ième lettre de l'alphabet et l'entier est *j*, tels que:

  $i = N$ modulo 26

  $j = N / 26$ (division entière).

Si une option n'est pas spécifiée, elle a pour valeur par défaut *:false*.

Exemples:

```
?- write_term(['quoted atom', "string", 1+2,
   '$VAR'(35)],[]).
[quoted atom,string,1 + 2,'$VAR'(35)]{}
?- write_term(['quoted atom', "string", 1+2,
   '$VAR'(35)],[quoted(:true),numbervars(true),
   ignore_ops(true)]).
['quoted atom',"string",+(1,2),J1]{}
?-
```

10.6. L'environnement

Un certain nombre de paramètres globaux appelés "variables d'état" peuvent être modifiés ou consultés. Ce sont:

- l'état du débogueur:
  nom de la variable d'état: *debug*
  valeurs possibles: *on* (actif) ou *off* (inactif).
- le comportement de Prolog sur une règle appelée et non définie:
  nom de la variable d'état: *unknown*
  valeurs possibles: *error* (génération d'une erreur), *warning* (génération d'un avertissement) ou *fail* (échec).
- l'interprétation des doubles quotes en position d'argument:
  nom de la variable d'état: *double_quotes*
  valeurs possibles: *chars* (liste de caractères), *codes* (liste des codes des caractères), *atom* (atome) ou *string* (vraies chaînes).
- la possibilité de convertir des caractères en d'autres:
  nom de la variable d'état: *char_conversion*
  Seule valeur possible actuellement: *off* (inactif).
- l'affirmation que l'arithmétique entière n'est pas limitée:
  nom de la variable d'état: *bounded*
  valeur non modifiable: *false*.
- l'arité maximale (nombre d'arguments) d'une règle ou d'une structure:
  nom de la variable d'état: *max_arity*
  valeur non modifiable: *8388607*.
- la définition de la division entière et du reste de celle-ci:

nom de la variable d'état: *integer_rounding_function*
valeur non modifiable: *toward_zero*.

current_prolog_flag(X,Y)

Renseigne sur l'état courant des variables d'état de l'environnement. Les variables d'état: *max_integer* et *min_integer* provoquent toujours un échec car l'arithmétique entière n'est pas bornée. Exemple: On veut connaître la valeur de toutes les variables d'état:

```
?- current_prolog_flag(F, V).
{F=bounded,V=false}
{F=integer_rounding_function,V=toward_zero}
{F=char_conversion,V=off}
{F=max_arity,V=8388607}
{F=:debug,V=off}
{F=unknown,V=warning}
{F=double_quotes,V=:string}
```

halt(X), *halt*

Sont respectivement équivalents à *quit(X)* et *quit*.

set_prolog_flag(X,Y)

Positionne la variable d'état *X* de l'environnement à la valeur *Y*. Ce prédicat est aussi une directive de compilation, c'est à dire que les variables d'environnement peuvent être modifiées au milieu d'un source que l'on compile. Exemples:

```
?- consult.
set_prolog_flag(double_quotes, codes).
foo :- myrule("abc").
.
{}
?- set_prolog_flag(debug, on).
{}
```

10.7. Traduction de DCG[1]

En mode insertion, Prolog transforme en règles Prolog II+ les règles de grammaire de la forme

A --> B.

où A est un terme et B est une suite de termes. Pour chaque terme de la suite de termes B :

- Si c'est une liste, il est interprété comme une suite de symboles terminaux.

[1]Definite Clause Grammar, extension des grammaires hors-contexte et sous-ensemble des grammaires de métamorphose.

- Si c'est une chaîne de caractères, celle-ci est transformée en une liste de caractères qui est donc ensuite considérée également comme une liste de symboles terminaux.

- Si c'est un terme entre accolades, il est interprété comme un appel à une règle à effectuer tel quel.

- Sinon le terme est considéré comme un non-terminal représentant une règle de réécriture.

La règle prédéfinie *phrase/2* ou *phrase/3* permet d'analyser ou de synthétiser les chaînes grammaticales définies par cette grammaire (voir § 10.2).

Pour traduire la règle de grammaire en règle Prolog, Prolog appellera le prédicat *:term_expansion/2* s'il a été défini, sinon il utilisera son propre programme de transformation. Il est ainsi possible de réaliser automatiquement des transformations sur les termes de la forme (A --> B). Si l'on désire définir un programme de transformation, il doit définir la règle suivante :

:term_expansion(X,Y)

Lors de l'appel, X est le terme que Prolog vient de lire et Y est une variable libre. X est de la forme (A --> B).
Après appel, Y doit représenter la règle Prolog qui doit être associée à la règle de réécriture X (c'est à dire la règle Prolog qui sera assertée).

Voici deux exemples de compilation et d'interrogation d'une grammaire DCG :

```
?- [user].
somme(Z) --> chiffre(Z).
somme(X+Y) --> chiffre(X), "+", somme(Y).
chiffre(1) --> "1".
...
chiffre(9) --> "9".
.
{}
?- list.
somme(_53,_54,_55) :-
  chiffre(_53,_54,_55).
somme(_57 + _58,_54,_55) :-
  chiffre(_57,_54,["+" | _59]),
  somme(_58,_59,_55).

chiffre(1,["1"|_54],_54).
...
chiffre(9,["9"|_54],_54).
{}
?- phrase(somme(Z),"1+2+3").
{Z=1 + (2 + 3)}
?- phrase(somme(1+(2+9)),L).
{L=["1","+","2","+","9"]}
```

Exemple 2 :

```
?- [user].
somme(Z) --> chiffre(Z).
somme(Z) --> chiffre(X), "+", somme(Y),{Z is X+Y}.
chiffre(N) --> [C],{string_integer(C,N)}.

liste3 -->  elt, elt, elt.
elt --> [a].
elt --> [b].
elt --> [c]. .
{}
?- phrase(somme(Z),"1+2+3").
{Z=6}
?- phrase(liste3,L).
{L=[a,a,a]}
{L=[a,a,b]}
{L=[a,a,c]}
...
{L=[c,c,c]
```

Annexe A
Différences Prolog II+ / Norme

A.1 Niveau général
A.2 Fonctionalités non implantées
A.3 Fonctionalités restreintes

A.1 Niveau général

Portée de la coupure dans un méta-appel

En Prolog II+, la coupure dans un méta-appel a une portée supérieure à celle définie dans la normalisation. Ce qui signifie que l'on a par exemple les différences de fonctionnement suivantes par rapport à la norme:

```
?- insert.
a(1).
a(2).
.
?- call( (Z=!, a(X), Z) ).
{Z=!,X=1}   /* 1 seule solution au lieu de 2 */
```

Les chaînes de caractères

En Prolog II+, une option supplémentaire par rapport à celles définies par la norme permet de classer un texte "entre doubles quotes" dans un type à part entière qui est le type "chaînes de caractères". Cette option est celle en vigueur par défaut au lancement de Prolog.

Préfixage des identificateurs

En Prolog II+, les identificateurs lus se voient toujours attribuer un préfixe en fonction du contexte courant, qu'ils se trouvent en position d'argument (foncteurs) ou bien en position de but (tête ou queue de règle). Néanmoins, nous fournissons la possibilité d'attribuer automatiquement le préfixe vide ("") aux identificateurs non préfixés, en position d'argument. Il suffit pour cela de démarrer Prolog avec l'option "-f a0".

Par exemple, dans ce mode là, la lecture de *call(nl)* fera appel à une règle sans doute non définie (*:nl*), tandis que *call(sys:nl)* imprimera bien un saut de ligne.

Pour annuler cette possibilité, exécuter le prédicat:

```
?- set_options("a1").
```

Vue logique et vue immédiate

Dans le mécanisme de la modification dynamique d'un paquet de règles (*assert*, *retract*, *suppress*), la norme définit la vue dite "logique", c'est à dire que tout se passe comme si l'exécution d'un paquet de règles se faisait sur une copie de celui-ci, donc indépendamment des modifications éventuelles subites lors de cette exécution. Une autre vue possible est celle dite "immédiate", qui consiste à prendre immédiatement en considération toute modification d'un paquet.

La vue de Prolog II+ est la vue **immédiate**.

Les seuls cas où le comportement peut être différent d'une prise en compte immédiate sont ceux de la suppression d'une règle dans un paquet où il reste encore des alternatives à exécuter. L'énumération de tous les cas de figures est alors assez complexe: cela dépend du fait qu'il y ait ou pas une indexation sur le paquet de règles en question, de la position de la règle modifiée par rapport à la règle du même paquet en cours d'exécution, du type du premier argument du but (variable, constante, ...).

Les n-uplets

Le type n-uplet n'est pas défini dans la norme Prolog. En termes d'équivalence, nous pouvons dire que le terme *f(a1, a2, ..., an)* **où *f* est un atome**, est équivalent au n-uplet<>*(f, a1, a2, ..., an)*, **sauf** dans le cas où n=0 (*f* n'est pas équivalent à <>*(f)*).

A.2 Fonctionalités non implantées

Les directives *char_conversion/2* et *current_char_conversion/2* ne sont pas implantées.

A.3 Fonctionalités restreintes

La directive *multifile/1* est équivalente à la directive *discontiguous/1*, la notion de fichier n'étant pas prise en compte par la compilation.

Seule la valeur *:off* est prise en compte pour le second paramètre de la directive de compilation et primitive *set_prolog_flag/2* dans le cas où le premier vaut *:char_conversion.*

Annexe B
Liste des directives et prédicats prédéfinis

a : constante (identificateur, entier, réel, ou chaîne)
b : 0 ou 1
c : chaîne d'un caractère
d : réel double précision
f : identificateur ou chaîne
h : canal ou alias (entier opaque ou identificateur)
i : identificateur
l : liste
m : entier ou réel
n : entier
r : réel quelconque
s : chaîne
t : terme
u : unité: "console" ou nom de fichier ou nom de fenêtre (chaîne ou atome)
v : variable

\+ : l'argument doit être instancié (même partiellement)
- : l'argument doit être une variable (instancié si succès)
? : l'argument peut être quelconque, il sera instancié si succès
@ : l'argument peut être quelconque, il restera inchangé après exécution

1 Directives de compilation

discontiguous(+i / +n)	`discontiguous(foo/2)`
dynamic(+i / +n)	`dynamic(aa/2)`
ensure_loaded(+u)	`ensure_loaded("myfile")`
include(+u)	`include("myfile")`
initialization(+t)	`initialization(write(end))`
module(+s)	`module("Interface")`
module(+s, +l)	`module("room",["",["table","glass"]])`
module(+s, +l1, +l2)	`module("lex",["",["uncle","son"]], ["1"])`
module(+t, +l1, +l2, +s)	`module(lexicon,[],["sys"],"lex")`
multifile(+i / +n)	`multifile(foo/2)`
omodule(+s)	`omodule("Interface")`
op(+n, +i1, +i2)	`op(900,fy,not)`
set_alias(+i, +a)	`set_alias(foo,122)`

set_prolog_flag(+i1, +i2)	`set_prolog_flag(unknown,warning)`

2 Prédicats

A

*abolish(**@**t)*	`abolish(myrule/2)`
abs(+m)	`val(abs(-1.5e0),1.5e)`
add(+m1, +m2)	`val(add(2.3E0,3.1E1),3.33E1)`
add_implicit(+s1, +s2)	`add_implicit("sys","outml")`
add_sentence_terminator(+c)	`add_sentence_terminator(";")`
*add_tlv(-v, **?**t, +i)*	`add_tlv(V, 33, foo)`
alloc	
*alloc(**?**n1, **?**n2, **?**n3, **?**n4, **?**n5, **?**n6, **?**n7, **?**n8, **?**n9, **?**n10, **?**n11, **?**n12)*	
*arg(+n, +t1, **?**t2)*	`arg(0,<>(aa,bb),aa)`
*arg2(+n, +t1, **?**t2)*	`arg2(2,<>(aa,bb),bb)`
assert(+t)	`assert(brother(jean,paul))`
assert(+t1, +t2)	`assert(pp(X),[qq(X),rr(X)])`
assert''(+t1, +t2)	`assert''(pp(X),[qq(X),rr(X)])`
asserta(+t)	`asserta(brother(jean,paul))`
asserta(+t1, +t2)	`asserta(pp(X),[qq(X),rr(X)])`
assertn(+t1, +t2,+n)	`assertn(pp(x),[qq(X),rr(X)],2)`
assertz(+t1, +t2)	`assertz(pp(x),[qq(X),rr(X)])`
assertz(+t)	`assertz(brother(X,Y):-son(X,Z), son(Y,Z))`
*assign(+i, **@**t)*	`assign(fo,tt(3,[1,2,x],tab[5],"str"))`
atan(+m)	`val(atan(1),X)`
*atom(**@**t)*	`atom(foo)`
*atomic(**@**t)*	`atomic(1)`
*atom_chars(+i, **?**l)*	`atom_chars(foo,X)`
atom_chars(-i, +l)	`atom_chars(X,["a","b"])`
*atom_codes(+i, **?**l)*	`atom_codes(foo,X)`
atom_codes(-i, +l)	`atom_codes(X,[65,66])`
*atom_concat(**?**i1, **?**i2, +i)*	`atom_concat(X,Y,foo)`
atom_concat(+i1, +i2, -l)	`atom_concat(fo1,fo2,X)`
*atom_length(+i, **?**n)*	`atom_length(foo,X)`
at_end_of_stream	
at_end_of_stream(+h)	`at_end_of_stream(alias1)`

B

*bagof(**?**v, +t, **?**l)*	`bagof(X,frere(X,Y),L)`
beep	
*block(**?**t1, +t2)*	`block(edit,call_editor)`
*block(**?**t1, **?**t2, +t3)*	`block(edit,_info,call_editor)`
*block_exit(**@**t)*	`block_exit(edit)`
*block_exit(**@**t1, **@**t2)*	`block_exit(edit, "this error")`
*bound(**@**t)*	`bound(X)`

C

call(+t)	`call(outml("Hello"))`
callC(+t1)	`callC(sscanf("12.","%f",<>("R",X)))`
callC(+t1, ?t2)	`callC(open("myfile"),<>("I",X))`
catch(+t1, ?t2, +t3)	`catch(foo(5),test(Y),outml("Hello"))`
char_code(+c, ?n)	`char_code("A",65)`
char_code(-c, +n)	`char_code(X,65)`
chrono(?v)	
clause(+t1, ?t2)	`clause(brother(X,Y),Q)`
clear_input	
close(+h)	`close(alias1)`
close(+h, +l)	`close(alias1, force(true))`
close_context_dictionary(+s)	`close_context_dictionary("data")`
close_input	
close_input(+u)	`close_input("myfile")`
close_output	
close_output(+u)	`close_output("myfile")`
compound(@t)	`compound([1,2])`
conc_list_string(+l, ?s)	`conc_list_string(["ab","cd"],"abcd")`
conc_string(?s1, ?s2, +s3),	`conc_string("ab",X,"abcd")`
conc_string(+s1, +s2, -s3)	`conc_string("ab","cd",X)`
consult(+f)	`consult("prog1.p2E")`
copy_term(@t1, ?t2)	`copy_term([1,2,<>("zz",X),Z], K)`
copy_term_with_constraints(@t1, ?t2)	`dif(X,3) copy_term_with_constraints( [1,2,<>(X),Z], K)`
cos(+m)	`val(cos(+3.14e0),X)`
cpu_time(?v)	
current_context(?t)	`current_context(X)`
current_context(?t, ?l1, ?l2, ?s)	`current_context(X,Y,Z,D)`
current_file(?u)	`current_file(X)`
current_file(?u , ?i1, ?i2)	`current_file(X,T,M)`
current_input(?n)	`current_input(X)`
current_op(?n, ?i, ?f)	`current_op(200,xfx,Y)`
current_output(?n)	`current_output(X)`
current_predicate(?i / ?n)	`current_predicate(:to_begin/0)`
current_prolog_flag(?i1, ?i2)	`current_prolog_flag(unknown,X)`

D

date(?v1, ?v2, ?v3, ?v4)	`date(J,M,A,S)`
date_string(?v)	
date_stringF(?v)	
debug	
debug(+u)	`debug("echodebug.dat")`
debug(+n)	`debug(3)`
debug(+n, +u)	`debug(3, "echodebug.dat")`
def_array(+i, +n)	`def_array(stack,100)`
default(+t1, +t2)	`default(man(X),eq(x,[]))`

delay(+n)	`delay(5000)`
dictionary	
dictionary(?l)	
dictionary(+s, ?l)	`dictionary("sys",X)`
dif(?t1, ?t2)	`dif(X,toto)`
div(+m1, +m2)	`val(div(8.0e,3),X)`
dot(@t)	`dot([1,X])`
double(+m)	`val(double(1),1e0)`
draw_equ(@t)	`draw_equ(aa(bb(cc),bb(cc)))`
draw_mode(?s)	
draw_tree(@t)	`draw_tree(aa(bb(cc),bb(cc,dd)))`

E

echo	
edinburgh	
edit(+i / +n)	`edit(hors_d_oeuvre/1)`
edit(+l)	`edit([meat/1,fish/1])`
edit(+s)	`edit("file.dat")`
editm(+s)	`editm("my_module")`
enum(?v, +n)	`enum(X,10)`
enum(?v, +n1, +n2)	`enum(X,10,20)`
eof	
eol	
eq(?t1, ?t2)	`eq(X,father(john,mary))`
eql(+a1, +a2)	`val(eql(3,3),1)`
equations(+t1, ?t2, ?l)	`equations(X,T,1)`
exit	
exit(+s)	`exit("myfile.psv")`
exp(+m)	`val(exp(+1.0e0),X)`

FGH

fail	
fail_if(+t)	`fail_if(fail)`
fasserta(+t)	`fasserta(town("Marseille",13))`
fassertz(+t)	`fassertz(town("Marignane",13))`
file_dictionary(+s, ?l)	`file_dictionary("foo.mo",X)`
find_pattern(+s1, +s2, ?n)	`find_pattern("abcdef","de",4)`
findall(?V, +T, ?L)	`findall(X,brother(X,Y),L)`
float(+m)	`val(float(5),+5.0e0)`
flush	
flush_output	
flush_output(+h)	`flush_output(alias1)`
free(@t)	`free(X)`
freeze(-v, +t)	`freeze(X,list(X))`
freplace(+t, +n, +t')	`freplace(old(X,Y),1,"010180")`
freplace(+i / +a, +n1, +n2, +t)	`freplace(old/2,10,2,"011293")`
fretract(+t)	`fretract(town(X,83))`
fretractall(@t)	`fretractall(town(X,83))`

functor(-t, +i, +n)	`functor(X,aa,2)`
functor(+t, ?i, ?n)	`functor(aa(bb(X),bb(cc)),aa,X)`
gc(+i)	`gc(:dictionary)`
gensymbol(-v)	`gensymbol(X)`
get(?n)	`char_code(";",C), get(C)`
get(+h, ?n)	`get(alias1,C)`
get_byte(?n)	
get_byte(+h, ?n)	`get_byte(alias1, C)`
get_char(?c)	
get_char(+h, ?c)	`get_char(alias1, C)`
get_code(?n)	`char_code(";",C), get_code(C)`
get_code(+h, ?n)	`get_code(alias1, C)`
get_option(+c, ?c)	`get_option("r", C)`
get_tlv(-v, ?l)	`get_tlv(V, L)`
get0(?n)	`get0(X)`
getenv(+s1, ?s2)	`getenv("PrologEdit",X)`
halt	
halt(+n)	`halt(1)`
hidden(+i / +n)	`hidden(my_rule/0)`
hidden(+s)	`hidden("private")`
hidden_debug(+i / +n)	`hidden_debug(my_rule/0)`
hidden_debug(+s)	`hidden_debug("private")`
hidden_rule(+i / +n)	`hidden_rule(my_rule/0)`
hidden_rule(+s)	`hidden_rule("private")`

IK

ident(@t)	`ident(toto)`
if(+b, +a1, +a2)	`val(if(inf(1,5),add(1,1),3),2)`
import_dir(?s)	
in(?t , ?c)	`in(X,C)`
in(?t , ?l, ?c)	`in(X,D,C)`
in_char(?c)	`in_char(X)`
in_char'(?c)	`in_char'(X)`
in_double(?d)	`in_double(X)`
in_ident(?i)	`in_ident(X)`
in_integer(?n)	`in_integer(X)`
in_real(?r)	`in_real(X)`
in_sentence(?t1, ?t2)	`in_sentence(X,Y)`
in_string(?s)	`in_string(X)`
in_word(?s, ?a)	`in_word(X,Y)`
include(+s)	`include("myfile")`
index(+i / +n)	`index(database/1)`
inf(+a1, +a2)	`val(inf("to","zou"),1)`
infe(+a1, +a2)	`val(infe("to","zou"),1)`
infinite	
infinite_flag	`val(infinite_flag,X)`
init_fassert(+i / +a, +l)	`init_fassert(town/2,[[1,2],1,2])`
inl(?v)	`inl(X)`

inl(?v1, ?v2)	`inl(X,Y)`
input(+u)	`input("console")`
input(+u, +n)	`input("myfile", 1000)`
input_is(?u)	`input_is("console")`
insert	
insert(+s)	`insert("mutant.p2")`
insertz	
insertz(+s)	`insert("mutant.p2")`
integer(@t)	`integer(3)`
is(?t1, @t2)	`X is 1+2`
is_array(+ i, ?v)	`is_array(tab,X)`
keysort(+l, ?l)	`keysort([43,31,52,12], L)`
keysort(+l, ?l, +i)	`keysort([43,31,52,12], L, mycompar)`
kill_array(+i)	`kill_array(tab1)`
kill_module(+s)	`kill_module("lex")`

L

line	
line_width(?n)	`line_width(X)`
list	
list(+i / +n)	`list(foo/2)`
list(+i / +n, n)	`list(foo/2,4)`
list(+s)	`list("mymodule")`
list_of(?v, +l1, +t, ?l2)	`list_of(X,[],data(X),L)`
list_string(+l, ?s)	`list_string(["a","b"],X)`
list_tuple(+l, ?t)	`list_tuple([aa,bb,cc],x)`
listing	
listing(+s)	`listing("mymodule")`
lkload(+s1, +s2)	`lkload("my_file","my_table")`
ln(+m)	`val(ln(1.0e0),0.0e0)`
log(+m)	`val(log(1.0e0),0.0e0)`
load(+s)	`load("myfile")`
load(+s, +l)	`load("myfile", [<>("aa","bb")])`

MN

member(?t, ?l)	`member(1,[1,2,3,4])`
memory_file(+s)	`memory_file("non-terminal")`
memory_file(+s, +n)	`memory_file("lexical",32000)`
mod(+n1, +n2)	`val(mod(7,3),1)`
month(?i, ?s)	`month(5,"may")`
ms_err(+i, ?s)	`ms_err(104,X)`
mul(+m1, +m2)	`val(mul(add(3.1e0,0.077d0),5),X)`
name(+f, ?l)	`name(toto,X)`
name(-f, +l)	`name(X,[116,111,116,111])`
new_tlv(-v, ?t, +i)	`new_tlv(V, 33, foo)`
next_char(?c)	`next_char(",")`
next_char'(?c)	`next_char'(X)`
nil	

nl	
nl(+h)	`nl(alias1)`
no_debug	
no_echo	
no_index(+i / +a)	`no_index(foo/2)`
no_infinite	
no_paper	
no_spy(+i / +n)	`no_spy(foo/2)`
no_trace	
nonvar(@t)	`nonvar([1|X])`
not(@t)	`not(true)`
not_defined(?l)	
number(@t)	`number(10)`
number_chars(+n, ?l)	`number_chars(123,X)`
number_chars(-n, +l)	`number_chars(X,["1","2","3"])`
number_codes(+n, ?l)	`number_codes(123,X)`
number_codes(-n, +l)	`number_codes(X,[49,50,51])`

OPQ

once(+t)	`once(out(foo))`
op(?n, ?i1, ?i2)	`op(900,fy,not)`
op(+n, +i1, +i2, +i3)	`op(700,xfx,"=",eq)`
open(+s, +i, -v)	`open("file",:read,X)`
open(+s, +i, -v, +l)	`open("file",:read,X,[alias(alias1))`
or(+t1, +t2)	`or(eq(X,3),outl(X).fail)`
out(@t)	`out([a,b])`
out_equ(@t)	`out_equ(aa(bb))`
outl(@t)	`outl([a,b])`
outm(+s)	`outm("bonjour")`
outm(+s, +n)	`outm("_",30)`
outml(+s)	`outml("The cat and the dog")`
output(+s)	`output("myfile.pro")`
output(+u , +n)	`output("myfile", 1000)`
output_is(?s)	`output_is("myfile.pro")`
page	
paper	
peek_byte(?n)	
peek_byte(+h, ?n)	`peek_byte(alias1,X)`
peek_char(?c)	
peek_char(+h, ?c)	`peek_char(alias1,X)`
peek_code(?n)	
peek_code(+h, ?n)	`peek_code(alias1,X)`
phrase(?t1, ?t2)	`phrase(somme(X),"1+2+3")`
phrase(?t1, ?t2, ?t3)	`phrase(somme(X),"1+2+3",Y)`
predefined(+t)	`predefined(outm("Ok"))`
prefix_limit(?c)	`prefix_limit(X)`
prologII	
prologIIE	

put(+n)	`char_code("!",C), put(C)`
put_byte(+n)	`put_byte(12)`
put_byte(+h, +n)	`put_byte(alias1,12)`
put_char(+c)	`put_char("e")`
put_char(+h, +c)	`put_char(alias1,"e")`
put_code(+n)	`put_code(12)`
put_code(+h, +n)	`put_code(alias1,12)`
quit	
quit(+n)	`quit(1)`

R

rad(+m)	`val(rad(+90.0e0),X)`
read(?t)	`read(X)`
read(+h, ?t)	`read(alias1,X)`
read_rule(?v1, ?v2)	`read_rule(T,Q)`
read_term(?t, +l)	`read_term(X,[variables(L)])`
read_term(+h, ?t, +l)	`read_term(alias1,X,[variables(L)])`
read_unit(?v1, ?v2)	`read_unit(_type,_valeur)`
real(@t)	`real(-5.9e0)`
realloc(+i, +n)	`realloc(:code,50)`
reconsult(+f)	`reconsult("prog1.p2E")`
recv_double(+n1, ?v2)	`recv_double(5,D)`
recv_integer(+n1, ?v2)	`recv_integer(5,N)`
recv_real(+n1, ?v2)	`recv_real(5,R)`
recv_string(+n1, ?v2)	`recv_string(5,S)`
redef_array(+i, +n)	`redef_array(foo,1000)`
reinsert	
reinsert(+u)	`reinsert("prog.p2")`
reload(+s)	`reload("myfile")`
reload(+s, +l)	`reload("myfile", [<>("aa","bb")])`
remove_implicit(+s1, +s2)	`remove_implicit("sys","outml")`
remove_sentence_terminator(+c)	`remove_sentence_terminator(";")`
repeat	
reset_chrono	
reset_cpu_time	
retract(+t1, ?t2)	`retract(father(john,X),Y)`
retract(+t)	`retract(brother(X,Y):- true)`
retractall(@T)	`retractall(father(john,X))`
rule(+t1, ?t2)	`rule(father(john,x),Y)`
rule(+n, +t1, ?t2)	`rule(r,father(john,X),Y)`
rule(+n, +i, +t1, ?t2)	`rule(5,father,T,Q)`
rule_nb(+i / +n, ?v)	`rule_nb(foo/2,X)`

S

save(+l, +s)	`save(["aa","bb"], "myfile")`
save_state(+s)	`save_state("myfile")`
see(+f)	`see(filei)`
seeing(+f)	`seeing(filei)`

seen	
send_double(+n, +d)	`send_double(5,12d0)`
send_integer(+n1, +n2)	`send_integer(5,1000)`
send_real(+n, +r)	`send_real(5,3.14e0)`
send_string(+n, +s)	`send_string(5,"error")`
set_context(+s)	`set_context("user")`
set_context(+s1, +l1, +l2, +s2)	`set_context("user",[],["sys"],"")`
set_cursor(+n1, +n2)	`set_cursor(1,24)`
set_draw_mode(+s)	`set_draw_mode("TTY")`
set_import_dir(+s)	`set_import_dir("/home/user/")`
set_input(+h)	`set_input(alias1)`
set_line_cursor(+n)	`set_line_cursor(10)`
set_line_width(+n)	`set_line_width(132)`
set_options(+s)	`set_options("vEw2")`
set_output(+h)	`set_output(alias1)`
set_prefix_limit(+c)	`set_prefix_limit("$")`
set_prolog_flag(+i1, +i2)	`set_prolog_flag(unknown,warning)`
set_stream_position(+h, +n)	`set_stream_position(alias1,200)`
setarg(+n, +t1, @t2)	`setarg(3,[1,2,3,4,5],[1,2,3])`
setof(?v, +t, ?l)	`setof(X,frere(X,Y),L)`
show_spy(?v)	
sin(+m)	`val(sin(+3.14e0),X)`
sort(+l, ?l)	`sort([43,31,52,12], L)`
sort(+l, ?l, +i)	`sort([43,31,52,12], L, mycompar)`
split(@t, ?l)	`split(father(john),[father,john])`
split(+s, ?l)	`split("abc",["a","b","c"])`
spy(+i / +n)	`spy(foo/2)`
sqrt(+m)	`val(sqrt(+9.0e0),+3.0e0)`
state	
statistics	
statistics(?n1, ?n2, ?n3, ?n4, ?n5, ?n6, ?n7, ?n8, ?n9, ?n10, ?n11, ?n12)	
stream_property(+h, ?l)	`stream_property(alias1,alias(X))`
string(@t)	`string("abc")`
string_double(+s, ?d)	`string_double(1d0,X)`
string_double(-s, +d)	`string_double(X,1.2345678d15)`
string_ident(?s1, ?s2, +i)	`string_ident(X,"john",john)`
string_ident(+s1, +s2, -i)	`string_ident("user","john",X)`
string_ident(+s, ?i)	`string_ident("john",X)`
string_ident(-s, +i)	`string_ident(X,john)`
string_integer(+s, ?n)	`string_integer("123",X)`
string_integer(-s, +n)	`string_integer(X,123)`
string_real(+s, ?r)	`string_real("+12.E15",X)`
string_real(-s, +r)	`string_real(X,+1.2E16)`
string_term(+s, ?t)	`string_term("err(500,X)",X)`
string_term(-s, +t)	`string_term(X,err(500,V60))`
string_term(+s, ?t1, ?t2),	`string_term("err(500,X) ss",X,Y)`
string_term(-s, +t1, ?t2)	`string_term(X,err(1),Y)`
string_term(+s, ?t1, ?t2, +n),	`string_term("err(500,X) ss",X,Y,1000)`

string_term(-s, +t1, ?t2, +n)	`string_term(X,err(1),Y,1000)`
sub(+m1, +m2)	`val(sub(2d1,1e0),X)`
sub_atom(+i1, ?n1, ?n2, ?n3, ?i2)	`sub_atom(abracadabra,0,5,X,Y)`
substring(+s1, +n1, +n2, ?s2)	`substring("abcdef",4,2,X)`
sup(+a1, +a2)	`val(sup("to","zou"),0)`
supe(+a1, +a2)	`val(sup("zou","zou"),1)`
suppress(+i / +n)	`suppress(foo/2)`
suppress(+i / +n, +n)	`suppress(foo/2,5)`
sys_command(+s)	`sys_command("showusers")`

TUVW

tab(+n)	`tab(10)`
tan(+m)	`val(tan(-3.14e0),X)`
tassign(+i, @t)	`tassign(aa,tt(3,[1,X],tab[5],"str"))`
tassign(+i [+n], @t)	`tassign(tab[1],123)`
tell(+f)	`tell(file)`
telling(+f)	`telling(file)`
term_cmp(@t1, @t2, ?n)	`term_cmp(1,"foo",X)`
term_cmpv(@t1, @t2, ?n)	`term_cmpv(1,"foo",X)`
term_vars(@t1, ?l)	`term_vars(one(2.X,Y,Y),L)`
throw(@t)	`throw(12)`
time(?n)	
time(?n1, ?n2, ?n3)	
told	
trace	
trace(+s)	`trace("debug.dat")`
true	
trunc(+m)	`val(trunc(+3.8e0),3)`
truncate(+m)	`val(truncate(+3.8e0),3)`
tuple(@t)	`tuple(ff(a))`
tval(@t1, ?t2)	`tval(1+2*3+pile[1],X)`
unify_tlv(-v, ?t2)	`unify_tlv(V, 22)`
unify_with_occurs_check(?t1, ?t2)	`unify_with_occurs_check(X,1)`
val(@t1, ?t 2)	`val(3,3)`
var(@t)	`var(X)`
var_time(-v, ?n)	`var_time(X,Y)`
version(?n)	`version(X)`
week(?n, ?s)	`week(0,"sunday")`
write(@t)	`write("Hello world!")`
write(+h,@t)	`write(alias1,"Hello world!")`
write_canonical(@t)	`write_canonical("Hello world!")`
write_canonical(+h, @t)	`write_canonical(alias1, "Hello world!")`
write_term(@t ,+l)	`write_term("foo",[quoted(:true)])`
write_term(+h, @t , +l)	`write_term(alias1,"f", [quoted(:false)])`
writeq(@t)	`writeq(foo(1,2,3)`
writeq(+h, @t)	`writeq(alias1,foo(1,2,3)`

Autres

\(+n)	`val(\(4),X)`
~(+n)	`val(~(4),X)`
*+m1 * +m2*	`val(-3.14e0 * 0.5e,X)`
+m1 + +m2	`val(1+2,X)`
+t1 , +t2	`write("ca va"),nl`
+t1 ; +t2	`integer(X);real(X)`
- +m	`-2`
+m1 - +m2	`val(1-2,X)`
+t1 --> +t2	`phrase --> gn, gv.`
+t1 -> +t2	`string(X) -> see(X)`
+m1 / +m2	`val(3/4,X)`
+m1 // +m2	`Y is 3 // 4`
+n1 /\ +n2	`val(4 '/\' 8,X)`
+a1 < +a2	`val(1 '<' 2,X)`
+n1 << +n2	`val(1 '<<'4,X)`
***?**t1 = **?**t2*	`1 = X`
*+t1 =.. **?**t2*	`toto(1,2) =.. X`
***-**t1 =.. +t2*	`Y =.. [toto,X]`
+a1 =< +a2	`val(1 '=<' 1,X)`
***@**t1 == **@**t2*	`func(X) == func(Y)`
+t1 =:= +t2	`tab[I] =:= I`
+t1 =\= +t2	`(1+2) =\= (2*3)`
***@**t1 \= **@**t2*	`1 \= 2`
+a1 > +a2	`val(2 '>' 1,X)`
+a1 >= +a2	`val(2 '>=' 1,X)`
n1 >> +n2	`val(1024 '>>' 2,X)`
+n1 \/ +n2	`val(14 '\/' 687,X)`
***@**t1 \== **@**t2*	`func(1,X) \== func(1,2)`
+a1 @=< +a2	
+a1 @> +a2	
+a1 @>= +a2	
+a1 @< +a2	
*\+(**@**T)*	`\+(fail)`

Annexe C
Quelques exemples de programmes Prolog II+

C.1. Les grammaires
C.2. Dérivation formelle
C.3. Les mutants
C.4. Interrogation par évaluation d'une formule logique
C.5. Un casse-tête
C.6. Construction d'un chemin
C.7. Les arbres infinis

C.1. Les grammaires

Description du problème

La grammaire d'un langage est un ensemble de règles qui permet de déterminer si oui ou non une suite de mots construits sur un certain alphabet appartient au langage. Si oui, on peut alors mettre en évidence la structure sous-jacente de la phrase analysée.

Les langage *hors-contexte* forment une classe importante : voici par exemple la grammaire définissant l'ensemble de toutes les expressions arithmétiques que l'on peut construire sur les nombres entiers, les opérateurs +, -, * avec leurs priorités habituelles :

(0) <expression> ::= <somme>

(1) <somme> ::= <produit> <reste de somme>

(2) <produit> ::= <primaire> <reste de produit>

<primaire>
(3) ::= <nombre>
(4) ::= (<expression>)

<reste de somme>
(5) ::= <op add> <produit> <reste de somme>
(6) ::= <vide>

<reste de produit>

(7) ::= <op mul> <primaire> <reste de produit>
(8) ::= <vide>

(9) <op mul> ::= *

<op add>
(10) ::= +
(11) ::= -

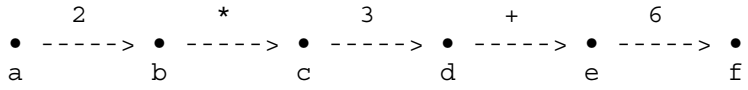
<nombre>
(12) ::= 0
(13) ::= 1
...
(21) ::= 9

(22) <vide> ::=

Par exemple, la décomposition de l'expression : 2 * 3 + 6 est donnée dans la figure C.1 (les noms des symboles non-terminaux ont été abrégés).

C.1.1. Représentation de la grammaire en Prolog

Examinons comment écrire cet analyseur en Prolog. Pour cela, considérons la chaîne d'entrée comme un graphe : le premier sommet du graphe est placé au début de la phrase, et on ajoute un sommet après chaque mot de la phrase. Chacun des mots va alors étiqueter l'arc qui joint les deux sommets qui l'encadrent. Avec notre exemple nous obtenons :



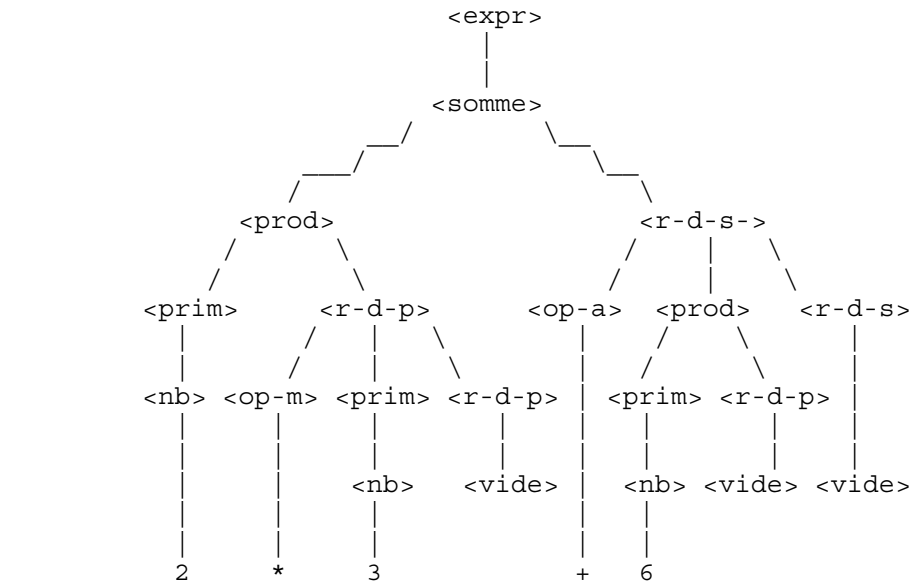


Figure C.1

Les règles de la grammaire peuvent alors être vues comme des instructions permettant de compléter ce graphe. C'est ainsi qu'une règle comme (21) sera interprétée par : s'il y a un arc étiqueté par "9" entre le noeud *x* et le noeud *y* du graphe, rajouter entre *x* et *y* un arc étiqueté "nombre".

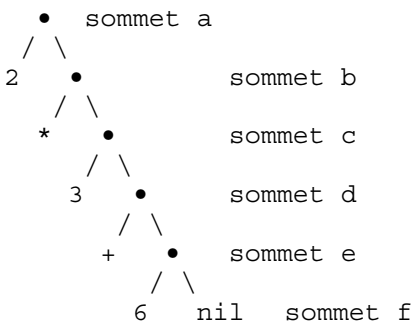
Une règle comme (1) donnera : s'il y a un arc étiqueté *produit* entre les noeuds *x* et *y* et un arc étiqueté *reste de somme* entre les noeuds *y* et *z*, rajouter un arc étiqueté *expression* entre *x* et *z*. Avec ces conventions, analyser une phrase revient à chercher un arc étiqueté *expression* entre le premier et le dernier sommet du graphe associé. Si on y parvient, la phrase sera acceptée, sinon elle sera rejetée.

Ecrivons maintenant le programme en Prolog : à chaque symbole non -terminal *N* de la grammaire, nous associerons un prédicat du même nom *N*(<*x, y*>) qui sera interprété par : il existe un arc étiqueté *N* entre les noeuds *x* et *y* du graphe.

Pour la règle (1) ceci donne la clause :

```
somme(<x, y>) -> produit(<x, z>) reste_de_somme(<z, y>);
```

Revenons un peu sur le graphe, celui-ci peut être simplement représenté par la liste des mots constituant la phrase d'entrée :



Chacun des sommets du graphe correspond à un sous-arbre de l'arbre précédent. Avec cette façon de faire, dans *N*(<*x*, *y*>), *x* représente la chaîne d'entrée avant d'effacer *N* et *y* représente la chaîne restant à analyser après avoir effacé *N*. De plus, cette représentation permet de traduire les règles terminales comme (12), (13), etc… par :

```
nombre(<x, y>) -> mot(n, <x, y>) integer(n) ;
```

qui utilise la règle standard :

```
mot(a, <a.x, x>) -> ;
```

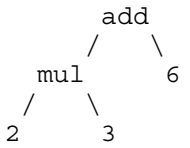
et la règle prédéfinie *integer*. Enfin, reconnaître si une phrase appartient au langage défini par notre grammaire revient à effacer :

```
expression(<p, nil>)
```

où *p* représente la phrase d'entrée sous forme de liste.

C.1.2. Mise en évidence de la structure profonde

Nous sommes donc rendus au point où nous savons traduire une grammaire en un ensemble de clauses qui diront si oui ou non une phrase appartient au langage. Nous pouvons faire mieux en complétant nos relations pour faire ressortir l'arbre construit par l'analyse de la phrase : pour l'exemple précédent.



Pour cela, nous n'aurons besoin que d'ajouter un argument aux prédicats associés aux non-terminaux de la grammaire. Celui-ci ne fera qu'exprimer comment une phrase est construite à partir des sous-phrases qui la composent. Nous obtenons donc :

```
nombre(n, <x, y>) -> mot(n, <x, y>) ;
primaire(n, <x, y>) -> nombre (n, <x, y>) ;
```

Finalement, notre règle de départ deviendra :

```
somme(e, <x, y>) ->
   produit(p, <x, z>) reste_de_somme(p, e, <z, y>);
```

où *e* représentera l'arbre associé à l'expression analysée. Voici le programme définitif :

```
"grammaire des expressions"

expression(e,<x,y>) -> somme(e,<x,y>);

somme(e,<x,y>) ->
   produit(p,<x,z>)
   reste_de_somme(p,e,<z,y>);

produit(p,<x,y>) ->
   primaire(f,<x,z>)
   reste_de_produit(f,p,<z,y>);

primaire(n,<x,y>) -> mot(n,<x,y>) integer(n);
primaire(e,<x,y>) ->
   mot("(",<x,z>)
   expression(e,<z,t>)
   mot(")",<t,y>);

reste_de_somme(p,e,<x,y>) ->
   mot(o,<x,z>)
   op_add(o,o_a)
   produit(p',<z,t>)
   reste_de_somme(<o_a,p,p'>,e,<t,y>);
reste_de_somme(e,e,<x,x>) ->;

reste_de_produit(f,p,<x,y>) ->
   mot(o,<x,z>)
   op_mul(o,o_m)
   primaire(f',<z,t>)
   reste_de_produit(<o_m,f,f'>,p,<t,y>);
```

```
reste_de_produit(f,f,<x,x>) ->;

mot(a,<a.x,x>) ->;

op_add("+",add) ->;
op_add("-",sub) ->;

op_mul("*",mul) ->;

"lecture"

read(l) -> in_sentence(s,l);

"lancement"

run ->
   outm("l'expression ")
   lire(p)
   analyse(p,e)
   val(e,f)
   outm("vaut")
   outl(f);

analyse(p,e) -> expression(e,<p,".".nil>) !;
analyse(p,e) -> outm("... est incorrecte") fail;
```

On utilise le prédicat *read* qui lit une phrase terminée par "." et la transforme en une liste de mots. Par exemple :

```
> read(p);
12+(23-4)*5+210-34-43.
{p=12."+"."(".23."-".4.")"."*".5."+".210."-".34."-
".43.".".nil}
```

Le prédicat *expression* construit l'arbre syntaxique associé à la phrase lue (si elle appartient au langage). On peut ensuite communiquer cet arbre au prédicat évaluable *val* qui calcule et imprime sa valeur. Tout ceci est fait par l'appel de la règle *run*.

C.2. Dérivation formelle

En analyse mathématique, la dérivation est une opération qui, à une expression algébrique associe une autre expression algébrique appelée dérivée. Les expressions sont construites sur les nombres, les opérateurs +, -, *, des symboles fonctionnels comme *sin*, *cos*… et un certain nombre d'inconnues. La dérivation se fait par rapport à l'une de ces inconnues. Dans notre exemple, les nombres seront toujours entiers. Nous n'aurons qu'une inconnue : *x*, et les seuls symboles fonctionnels unaires seront *sin* et *cos*. La première partie du programme est la grammaire d'analyse de ces expressions : c'est celle de l'exemple précédent qui a été complétée. La voici :

```
"grammaire des expressions"

expression(e,<x,y>) -> somme(e,<x,y>);

somme(e,<x,y>) ->
```

```
   produit(p,<x,z>)
   reste_de_somme(p,e,<z,y>);

produit(p,<x,y>) ->
   facteur(f,<x,z>)
   reste_de_produit(f,p,<z,y>);

facteur(f,<x,y>) ->
   primaire(p,<x,z>)
   reste_de_facteur(p,f,<z,y>);

primaire(n,<x,y>) -> mot(n,<x,y>) integer(n);
primaire("x",<x,y>) -> mot("x",<x,y>);
primaire(<o_u,p>,<x,y>) ->
   mot(o,<x,z>)
   op_un(o,o_u)
   primaire(p,<z,y>);
primaire(e,<x,y>) ->
   mot("(",<x,z>)
   expression(e,<z,t>)
   mot(")",<t,y>);

reste_de_somme(p,e,<x,y>) ->
   mot(o,<x,z>)
   op_add(o,o_a)
   produit(p',<z,t>)
   reste_de_somme(<o_a,p,p'>,e,<t,y>);
reste_de_somme(e,e,<x,x>) ->;

reste_de_produit(f,p,<x,y>) ->
   mot(o,<x,z>)
   op_mul(o,o_m)
   facteur(f',<z,t>)
   reste_de_produit(<o_m,f,f'>,p,<t,y>);
reste_de_produit(f,f,<x,x>) ->;

reste_de_facteur(p,f,<x,y>) ->
   mot(o,<x,z>)
   op_exp(o,o_e)
   primaire(p',<z,t>)
   reste_de_facteur(<o_e,p,p'>,f,<t,y>);
reste_de_facteur(f,f,<x,x>) ->;

mot(a,<a.x,x>) ->;

op_add("+",plus) ->;
op_add("-",minus) ->;

op_mul("*",mult) ->;

op_exp("^",exp) ->;

op_un("-",minus) ->;
op_un(sin,sin) ->;
op_un(cos,cos) ->;

"règles de dérivation"

derivee(x,x,1) ->;
derivee(n,x,0) -> integer(n);
derivee(plus(u,v),x,plus(u',v')) ->
```

```
   derivee(u,x,u')
   derivee(v,x,v');
derivee(minus(u,v),x,minus(u',v')) ->
   derivee(u,x,u')
   derivee(v,x,v');
derivee(mult(u,v),x,plus(mult(u,v'),mult(v,u'))) ->
   derivee(u,x,u')
   derivee(v,x,v');
derivee(exp(u,n),x,mult(n,mult(u',exp(u,minus(n,1))))) ->
   derivee(u,x,u');
derivee(minus(u),x,minus(u')) -> derivee(u,x,u');
derivee(sin(u),x,mult(u',cos(u))) -> derivee(u,x,u');
derivee(cos(u),x,minus(mult(u',sin(u)))) -> derivee(u,x,u');

"règles de simplification"

simplifier(<o_b,x,y>,u) ->
   simplifier(x,x')
   simplifier(y,y')
   simp(o_b,x',y',u);
simplifier(<o_u,x>,u) -> simplifier(x,x') simp(o_u,x',u);
simplifier(x,x) ->;

simp(plus,0,x,x) ->;
simp(plus,x,0,x) ->;
simp(minus,x,0,x) ->;
simp(minus,0,x,y) -> simp(minus,x,y);
simp(mult,0,x,0) ->;
simp(mult,x,0,0) ->;
simp(mult,1,x,x) ->;
simp(mult,x,1,x) ->;
simp(exp,x,0,1) ->;
simp(mult,minus(x),y,u) ->
   simp(mult,x,y,v)
   simp(minus,v,u);
simp(mult,x,minus(y),u) ->
   simp(mult,x,y,v)
   simp(minus,v,u);
simp(exp,x,1,x) ->;
simp(exp,0,x,0) ->;
simp(exp,1,x,1) ->;
simp(o_b,x,y,u) ->
   dif(o_b,exp)
   integer(x)
   integer(y)
   evalCst(<o_b,x,y>,u);
simp(o_b,x,<o_b,u,v>,t) -> simp(o_b,x,u,z) simp(o_b,z,v,t);
simp(o_b,x,y,<o_b,x,y>) ->;

simp(minus,0,0) ->;
simp(minus,minus(x),x) ->;
simp(sin,0,0) ->;
simp(cos,0,1) ->;
simp(o_u,x,<o_u,x>) ->;

evalCst(plus(x,y),u) -> val(x+y,u);
evalCst(minus(x,y),u) -> val(x-y,u);
evalCst(mult(x,y),u) -> val(x*y,u);

"lecture"
```

```
read(nil) -> next_char'(".") ! in_char'(".");
read(a.b) -> in_ident(a) ! read(b);
read("+".b) -> next_char'("+") ! in_char'("+") read(b);
read("-".b) -> next_char'("-") ! in_char'("-") read(b);
read(a.b) -> in_integer(a) ! read(b);
read(a.b) -> in_char'(a) read(b);

"écriture"

ecrire(<o_b,x,y>) ->
   op_bin(o,o_b)
   outm("(")
   ecrire(x)
   outm(o)
   ecrire(y)
   outm(")");
ecrire(minus(x)) -> ! outm("-") ecrire(x);
ecrire(<o_u,x>) -> ! op_un(o,o_u) out(o) ecrire(x);
ecrire(x) -> string(x) ! outm(x);
ecrire(x) -> out(x);

op_bin(o,o_b) -> op_add(o,o_b);
op_bin(o,o_b) -> op_mul(o,o_b);
op_bin(o,o_b) -> op_exp(o,o_b);

"lancement"

run ->
   outm("l'expression ")
   read(p)
   analyse(p,e)
   derivee(e,"x",e')
   simplifier(e',f)
   outm("a pour derivee")
   ecrire(f)
   line
   !;

analyse(p,e) -> expression(e,<p,nil>) !;
analyse(p,e) -> outm("... est incorrecte") fail;
```

Voici par exemple le résultat de l'analyse d'une expression:

```
$ prolog

> insert("deriv.p2");
{}
> read(p) expression(e, <p, nil>);
3*x^2+6*x+5.
{p=3."*"."x"."^".2."+".6."*"."x"."+".5.".".nil,
e=plus(plus(mult(3,exp("x",2)),mult(6,"x")),5)}
```

Nous avons ensuite défini la relation de dérivation : *derivee(f, x, f')* signifie : *f'* est la dérivée de *f* par rapport à *x*. Cette relation comporte une règle par opérateur ou symbole fonctionnel et s'exprime de manière très naturelle. Par exemple, la dérivée de l'expression précédente est donnée par :

```
> read(p) expression(e, <p, nil>) derivee(e, "x", e');
3*x2^+6*x+5.
```

```
...,
e'=plus(plus(plus(mult(3,mult(2,mult(1,exp("x",minus(2,1))))))
,
mult(exp("x",2),0)),plus(mult(6,1),mult("x",0))),0)}
```

Comme on le voit, le résultat est loin d'être simplifié ! Nous avons donc adjoint un programme de simplification (très incomplet) qui permet d'obtenir une écriture plus condensée. Tout cela est résumé par le prédicat *run* dont voici un exemple d'utilisation :

```
> run;
l'expression 3*x^2+6*x+5.
a pour derivee ((6*x)+6)
> run;
l'expression cos(-3*x^2+2).
a pour derivee ((6*x)*sin(-(3*(x^2))+2))
```

C.3. Les mutants

Il s'agit de produire des mutants issus d'animaux différents. Pour cela les animaux sont connus par leur nom sous forme de chaîne de caractères. Deux animaux donnent naissance à un mutant si la fin du nom du premier animal est identique au début du nom du second. L'aspect intéressant de ce programme est qu'on utilise une même relation, conc, de deux façons différentes : d'une part pour réunir deux listes, d'autre part pour décomposer une liste en deux sous-listes.

Voici les résultats produits en partant de l'ensemble d'animaux : alligator, tortue, caribou, ours, cheval, vache, lapin, pintade, hibou, bouquetin et chèvre.

```
> joli_mutant;
alligatortue
caribours
caribouquetin
chevalligator
chevalapin
vacheval
vachevre
lapintade
hibours
hibouquetin
>
```

et voici le programme (avec l'option -f vU):

```
"MUTANTS"

mutant(_z) ->
   animal(_x)
   animal(_y)
   conc(_a,_b,_x)
   dif(_b,nil)
   conc(_b,_c,_y)
   dif(_c,nil)
   conc(_x,_c,_z);

conc(nil,_y,_y) ->;
```

```
conc(_e._x,_y,_e._z) -> conc(_x,_y,_z);

joli_mutant -> mutant(_z) expl(_z) line fail;

expl(nil) ->;
expl(_a._l) -> out(_a) expl(_l);

animal(a.l.l.i.g.a.t.o.r.nil) ->;
animal(t.o.r.t.u.e.nil) ->;
animal(c.a.r.i.b.o.u.nil) ->;
animal(o.u.r.s.nil) ->;
animal(c.h.e.v.a.l.nil) ->;
animal(v.a.c.h.e.nil) ->;
animal(l.a.p.i.n.nil) ->;
animal(p.i.n.t.a.d.e.nil) ->;
animal(h.i.b.o.u.nil) ->;
animal(b.o.u.q.u.e.t.i.n.nil) ->;
animal(c.h.e.v.r.e.nil) ->;
```

C.4. Interrogation par évaluation d'une formule logique

Dans cet exemple on a constitué une banque de données portant sur des individus dont on connaît le nom, l'âge, la ville d'origine et le fait qu'ils portent ou non des lunettes. Tous ces renseignements sont résumés par une assertion telle que :

```
individu(candide, 20, constantinople, non) -> ;
```

qui indique que l'individu nommé *candide* est âgé de 20 ans, qu'il est né à *constantinople* et ne porte pas de lunettes. On dispose également de relations élémentaires (les formules atomiques) portant sur ces données.

Le programme consiste à évaluer une formule logique construite à partir des formule atomiques, des connecteurs "et" et "ou" et des quantificateurs existentiel et universel portant sur des variables typées, c'est à dire dont le domaine des valeurs est précisé. Ainsi, la question type que l'on peut poser est du genre :

> *« quels sont les valeurs de x appartenant au domaine D pour lesquelles la propriété P est vraie ? »*

ce qui est traduit par la formule :

```
element(x, ens(x, D, P)).
```

Par exemple, la question : (1) *dans quelle ville habite mimosa ?* se traduit par la formule.

```
element(x, ens(x, ville, habite_a(mimosa, x)))
```

de même : (2) *olive porte-t-elle des lunettes ?* se traduit par :

```
element(x, ens(x, booleen, lunette(olive, x)))
```

et enfin : (3) *quelles sont les villes ayant au moins un habitant age de moins de 20 ans et portant des lunettes ?* correspondant à :

```
element(x, ens(x, ville, existe(y, nom, et(habite_a(y, x),
   et(est_age_de(y, a),
   et(inferieur(a, 20),
   lunette(y, oui)))))))
```

Ces trois questions ont été pré-enregistrées et sont activées par *reponse-a-tout* qui écrit la question en clair suivie des réponses. Voici ce que cela donne (les réponses de la machine sont précédées de "-->") :

```
> reponse_a_tout;

dans quelle ville habite mimosa ?
--> aspres_sur_buech

olive porte-t-elle des lunettes ?
--> non

quelles sont les villes ayant au moins un habitant
age de moins de 20 ans et portant des lunettes ?
--> severac_le_chateau
--> aspres_sur_buech
```

Voici le programme complet :

```
"(1) banque de donnees"

individu(candide,20,constantinople,non) ->;
individu(cunegonde,20,constantinople,oui) ->;
individu(gontran,94,aspres_sur_buech,non) ->;
individu(casimir,2,severac_le_chateau,oui) ->;
individu(clementine,1,cucugnan,non) ->;
individu(popeye,99,aspres_sur_buech,oui) ->;
individu(olive,99,aspres_sur_buech,non) ->;
individu(mimosa,1,aspres_sur_buech,oui) ->;
individu(bip,15,pampelune,non) ->;
individu(ignace,114,loyola,oui) ->;
individu(balthazar,87,jerusalem,non) ->;
individu(gaspard,96,smyrne,oui) ->;
individu(melchior,34,kartoum,non) ->;

"(2) definition des types"

type(x,nom) -> nom(x);
type(x,age) -> age(x);
type(x,ville) -> ville(x);
type(x,booleen) -> booleen(x);

nom(candide) ->;
nom(cunegonde) ->;
nom(gontran) ->;
nom(casimir) ->;
nom(clementine) ->;
nom(popeye) ->;
nom(olive) ->;
nom(mimosa) ->;
nom(bip) ->;
nom(ignace) ->;
nom(balthazar) ->;
nom(gaspard) ->;
```

```
nom(melchior) ->;

age(20) ->;
age(94) ->;
age(2) ->;
age(1) ->;
age(99) ->;
age(15) ->;
age(114) ->;
age(87) ->;
age(96) ->;
age(34) ->;

ville(constantinople) ->;
ville(aspres_sur_buech) ->;
ville(severac_le_chateau) ->;
ville(cucugnan) ->;
ville(pampelune) ->;
ville(loyola) ->;
ville(jerusalem) ->;
ville(smyrne) ->;
ville(kartoum) ->;

booleen(oui) ->;
booleen(non) ->;

"(3) listes des formules atomiques"

atomique(habite_a(x,y)) ->;
atomique(est_age_de(x,y)) ->;
atomique(lunette(x,y)) ->;
atomique(plus_age(x,y)) ->;
atomique(inferieur(x,y)) ->;
atomique(different(x,y)) ->;

"(4) evaluation des formules atomiques"

habite_a(x,y) -> individu(x,a,y,b);

est_age_de(x,y) -> individu(x,y,v,b);

plus_age(x,y) ->
   individu(x,a,v,b)
   individu(y,a',v',b')
   val(inf(a',a),1);

lunette(x,y) -> individu(x,a,v,y);

inferieur(x,y) -> val(inf(x,y),1);

different(x,y) -> dif(x,y);

"(5) evaluation des formules"

vrai(p) -> atomique(p) p;
vrai(non(p)) -> non(vrai(p));
vrai(et(p,q)) -> vrai(p) vrai(q);
vrai(ou(p,q)) -> vrai(p);
vrai(ou(p,q)) -> vrai(q);
vrai(existe(x,t,p)) -> type(x,t) vrai(p);
vrai(tout(x,t,p)) -> non(vrai(existe(x,t,non(p))));
```

```
"(6) definitions utiles"

non(p) -> p ! fail;
non(p) ->;

"(7) calcul des reponses"

reponse_a_tout ->
   question(i,q)
   element(y,q)
   line
   outm("-->")
   out(y)
   line;

element(x,ens(x,t,p)) ->
   type(x,t)
   vrai(p);

"(8) listes des questions"

question(1,ens(x,ville,habite_a(mimosa,x))) ->
   outm("(1) dans quelle ville habite mimosa ?");
question(2,ens(x,booleen,lunette(olive,x))) ->
   outm("(2) olive porte-t-elle des lunettes ?");
question(3,ens(x,ville,existe(y,nom,et(habite_a(y,x),
  et(est_age_de(y,a),et(inferieur(a,20),lunette(y,oui)))))))
->
   outm("(3) quelles sont les villes ayant au moins un
habitant")
   outm("age de moins de 20 ans et portant des lunettes ?");
```

C.5. Un casse-tête

Dans cet exemple, il s'agit de résoudre le célèbre problème de crypto-arithmétique dans lequel en remplaçant chaque occurrence des lettres s, e, n, d, m, o, r, y par un même chiffre, on ait :

```
          SEND
      +MORE
      -----
      MONEY
```

Une programmation conventionnelle oblige à prendre en compte deux problèmes simultanément : celui de l'addition proprement dite et le fait que deux lettres différentes sont remplacées par deux chiffres différents. Au contraire, avec le *dif* retardé, ces deux problèmes sont bien séparés : le prédicat *differents* met en place toutes les inéquations à l'avance. On n'a plus ensuite qu'à exprimer l'addition et les *dif* se débloquent progressivement au fur et à mesure que l'on avance dans la résolution du problème. Le programme est rendu plus clair, mais aussi plus efficace. Voici la solution :

```
> jolie_solution;
  9567
 +1085
 -----
```

```
 10652
{}
```

Voici le programme :

```
"resolution de SEND+MORE=MONEY"

solution(s.e.n.d.m.o.r.y) ->
   different(s.e.n.d.m.o.r.y.nil)
   somme(r1,0,0,m,0)
   somme(r2,s,m,o,r1)
   somme(r3,e,o,n,r2)
   somme(r4,n,r,e,r3)
   somme(0,d,e,y,r4);

somme(r,0,0,r,0) -> ! retenue(r);
somme(r,x,y,z,r') ->
   retenue(r)
   chiffre(x)
   chiffre(y)
   val(add(r,add(x,y)),t)
   val(div(t,10),r')
   val(mod(t,10),z);

chiffre(0) ->;
chiffre(1) ->;
chiffre(2) ->;
chiffre(3) ->;
chiffre(4) ->;
chiffre(5) ->;
chiffre(6) ->;
chiffre(7) ->;
chiffre(8) ->;
chiffre(9) ->;

retenue(1) ->;
retenue(0) ->;

different(nil) ->;
different(x.l) -> hors_de(x,l) different(l);

hors_de(x,nil) ->;
hors_de(x,a.l) -> dif(x,a) hors_de(x,l);

jolie_solution -> solution(s) jolie_sortie(s);

jolie_sortie(s.e.n.d.m.o.r.y) ->
   outm(" ") out(s) out(e) out(n) out(d) line
   outm("+") out(m) out(o) out(r) out(e) line
   outm("----") line
   out(m) out(o) out(n) out(e) out(y) line;
```

C.6. Construction d'un chemin

Ce programme énumère des chemins sans boucle par utilisation de la règle *freeze*. Rappelons que *freeze* retarde l'effacement du littéral sur lequel il porte tant qu'une certaine variable n'est pas affectée. Il est utilisé ici pour construire un *bon chemin* qui est un chemin qui ne passe pas deux fois par la même étape.

Pour cela, on calcule un chemin possible par la règle *chemin* que l'on valide au fur et à mesure par la règle *bonne-liste*, ce qui permet de rejeter automatiquement le chemin en cours de construction dès qu'on tente de lui adjoindre une étape qui y figure déjà. Voici la liste des chemins sans boucle passant par Marseille, Londres, et Los Angeles suivie du programme :

```
> bon_chemin(l);
{l=Marseille.nil}
{l=London.nil}
{l=LosAngeles.nil}
{l=Marseille.London.nil}
{l=Marseille.London.LosAngeles.nil}
}l=Marseille.LosAngeles.nil}
{l=Marseille.LosAngeles.London.nil}
{l=London.Marseille.nil}
{l=London.Marseille.LosAngeles.nil}
{l=London.LosAngeles.nil}
{l=London.LosAngeles.Marseille.nil}
{l=LosAngeles.Marseille.nil}
{l=LosAngeles.Marseille.London.nil}
{l=LosAngeles.London.nil}
{l=LosAngeles.London.Marseille.nil}
>
```

Et voici le programme :

```
"chemins sans boucles"

bon_chemin(l) -> bonne_liste(l) chemin(l);

chemin(x.nil) -> etape(x);
chemin(x.x'.l) -> route(x,x') chemin(x'.l);

route(x,x') -> etape(x) etape(x');

etape(Marseille) ->;
etape(London) ->;
etape(LosAngeles) ->;
```

```
"liste sans repetition"

bonne_liste(l) -> freeze(l,bonne_liste'(l));

bonne_liste'(nil) ->;
bonne_liste'(x.l) -> hors_de(x,l) bonne_liste(l);

hors_de(x,l) -> freeze(l,hors_de'(x,l));

hors_de'(x,nil) ->;
hors_de'(x,x'.l) -> dif(x,x') hors_de(x,l);
```

C.7. Les arbres infinis

Les automates d'états finis sont un bon exemple d'objets qui peuvent être représentés par des arbres infinis (rationnels). Le programme présenté ici calcule un automate minimal qui accepte certaines chaînes et en refuse d'autres. Toutes ces chaînes sont faites à partir de *a* et *b*. La règle prédéfinie *draw-equ(x)* est utilisée à la fois pour dessiner l'automate et pour minimiser sa taille, c'est à dire le nombre de ses états. Son utilisation suppose que le module de dessin soit chargé.

```
> infinite;
> accepte(s, "a"."b".nil)     accepte(s, "b"."a".nil)
  refuse(s, "a".nil)          refuse(s, "b".nil)
  refuse(s, "a"."a".nil)      refuse(s, "b"."b".nil)
  refuse(s, "a"."a"."b".nil) refuse(s, "b"."b"."a".nil)
  entier_de_Peano(n)
  automate_de_taille(s, n)
  !
  draw_equ(s)
  fail;
x
x = final
      / \
    aa   bb
     |   |
     y   z
y = non_final
      / \
     aa   bb
      |   |
      z   x
z = non_final
      / \
     aa   bb
      |   |
      x   y
```

Le terme *fail* force le backtracking pour éviter l'impression finale. Voici le programme complet permettant de produire tout cela. Il faut noter que les buts *accepte(s,x)* et *refuse(s,x)* sont effacés de façon complétement déterministe. Pour cette raison, afin d'obtenir l'automate minimal, il n'est pas nécessaire de placer *entier_de_Peano(n)* comme premier but de la suite des buts précédents (à effacer).

```
"Automate d'etat fini"

accepte(s,nil) -> etat_final(s);
accepte(s,e.x) -> fleche(s,e,s') accepte(s',x);

refuse(s,nil) -> non_etat_final(s);
refuse(s,e.x) -> fleche(s,e,s') refuse(s',x);

etat_final(<final,aa(s1),bb(s2)>) ->;

non_etat_final(<non_final,aa(s1),bb(s2)>) ->;

fleche(<f,aa(s1),bb(s2)>,"a",s1) ->;
fleche(<f,aa(s1),bb(s2)>,"b",s2) ->;

"Calcul d'un automate de taille donnee"

automate_de_taille(s,n) -> automate_de_taille(s.nil,n,nil);
automate_de_taille(nil,0,l) ->;
automate_de_taille(s.l,n,l') ->
   element_de(s,l')
   automate_de_taille(l,n,l');
automate_de_taille(s.l,suc(n),l') ->
   non_element_de(s,l')
   fleche(s,"a",s1)
   fleche(s,"b",s2)
   automate_de_taille(s1.s2.l,n,s.l');

element_de(s,s.l) ->;
element_de(s,s'.l) -> dif(s,s') element_de(s,l);

non_element_de(s,nil) ->;
non_element_de(s,s'.l) -> dif(s,s') non_element_de(s,l);

"Enumeration des entiers de Peano"

entier_de_Peano(0) ->;
entier_de_Peano(suc(n)) -> entier_de_Peano(n);
```

Annexe D
ajout de règles externes (méthode des parasites)

D.1. Description générale
D.2. Règle Prolog réalisant l'interface
D.3. Appel du programme externe
D.4. Comment provoquer un *backtracking* ou renvoyer une erreur

Ce chapitre montre comment ajouter de nouvelles règles faisant référence à des fonctions externes écrites en C ou tout autre langage compatible avec C, et décrit les procédures d'interface. La méthode décrite ici est la seule qui fonctionne en mode 16 bits sur PC. Sinon, une méthode plus simple utilisant des descripteurs peut être utilisée avec le compilateur (voir le chapitre 7. du Manuel de Référence).

La fonction externe peut provoquer un *backtracking*, renvoyer une erreur qui sera traitée par le mécanisme *block*, ou afficher un message défini par l'utilisateur.

Le module utilisateur *prouser* sert de relais entre Prolog et les routines utilisateur écrites en C, de même le module utilisateur *fprouser* sert de relais entre Prolog et les routines utilisateur écrites en Fortran. De cette manière il est possible d'avoir simultanément des règles prédéfinies dans les deux langages. Les fichiers sources sont inclus dans le volume de distribution.

D.1. Description générale

Nous rappelons que l'appel depuis Prolog d'une fonction externe définie en C peut se faire de deux manières:

1. Par la méthode des descripteurs (voir § R 7.6). C'est une méthode d'appel direct avec génération automatique de la règle d'appel Prolog.

2. En utilisant un parasite via une procédure relais définie dans un module utilisateur fourni dans le kit. C'est la méthode décrite dans ce chapitre.

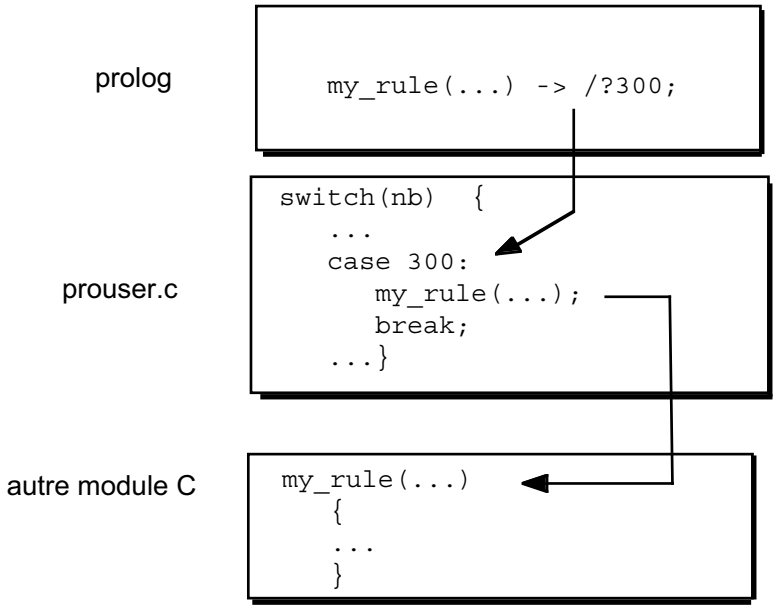
Quand Prolog est lancé sans spécifier d'état initial, l'état initial standard (fichier *initial.po*) est chargé. Ajouter de nouvelles règles écrites dans un autre langage, par la méthode des parasites décrite ici, consiste à:

1. Définir en Prolog l'appel à ces règles par Prolog, et l'inclure dans un état initial (cette étape se fait automatiquement dans la méthode des descripteurs).

2. Définir l'implantation externe en C (ou Fortran ou tout autre langage externe compatible), et l'inclure dans la Machine *prolog*.

Cette méthode nécessite deux relais d'appels :

- l'un entre le programme Prolog et le module utilisateur *prouser*
- l'autre entre ce module et la fonction externe.



Prenons un exemple. Pour ajouter une nouvelle règle, il faut d'abord choisir le module dans lequel on désire la définir. On peut utiliser un module réservé à cet usage, par exemple "Interface". Nous nous placerons dans ce cas pour l'exemple ci-dessous.

Les deux étapes de la méthode sont :

1. Définir la règle d'appel dans le module "Interface".

```
module("Interface");
my_rule(x,y,z) -> /?300;
end_module("Interface");
```

Ceci peut se faire dans un module source séparé, ou directement sur la ligne de commande de Prolog. Vous pouvez sauver ce module par la commande :

```
save( ["Interface"], "Interface.mo");
```

Il peut alors être rechargé dans n'importe quel état initial avec la commande *load*. Vous pouvez aussi créer directement un nouvel état initial par la commande:

```
exit("nouvel_etat.po");.
```

Nous nous placerons dans ce dernier cas pour la suite : *nouvel_etat.po* est alors un fichier contenant l'état initial augmenté de la règle d'appel compilée par Prolog.

2. Créer une nouvelle Machine *prolog* contenant le programme externe :
   - Ecrire le programme externe en utilisant les fonctions de communication décrites au § R 7.1 et le compiler avec le compilateur approprié.
   - Mettre à jour le module utilisateur *prouser* correspondant (voir § D.3.) et le compiler.
   - Refaire l'édition de liens entre les nouveaux modules et ceux qui constituent Prolog (se référer au § U 2.7 pour les commandes exactes pour votre machine)

Il suffit ensuite d'utiliser la nouvelle Machine Prolog avec l'état initial augmenté.

D.2. Règle Prolog réalisant l'interface

L'appel sous Prolog d'une fonction externe se fait au moyen d'une règle satisfaisant les conditions suivantes:

- la tête de la règle est un identificateur suivi par des arguments qui sont des variables libres.

- la queue de la règle contient un *parasite*, représenté par un nombre précédé de /?.

Exemple: la règle relais pour *find_pattern* est:

```
sys:find_pattern(s1,s2,n) -> /?201;
```

Le nombre sélectionne la fonction externe adéquate (voir la fonction *user_rule* dans le module *prouser*). Pour chaque nombre, il y a un appel de fonction externe correspondant dans la fonction C *user_rule* ou la fonction Fortran *fuserrule*.

Note inportante:
Les nombres inférieurs à 250 sont réservés au système (mais aucune vérification n'est faite). Les nombres supérieurs à 250 sont disponibles pour l'utilisateur. Il est conseillé d'utiliser les nombres compris entre 250 et 9999 pour les routines C dont l'appel se trouve dans le module *prouser*, les nombres compris entre 10000 et 20000 pour les routines Fortran (ou d'autres langages de type Fortran) dont l'appel se trouve dans le module *fprouser*, et les nombres supérieurs à 20000 pour les routines du mode 16 bits sous DOS/WINDOWS3 dont l'appel se trouve dans le module *prowuser*.

D.3. Appel du programme externe

Un programme externe ne peut être appelé depuis Prolog par la méthode des parasites, que si le nombre associé et l'instruction d'appel ont été ajoutés dans la fonction *user_rule* du module C *prouser* (respectivement *fuserrule* du module Fortarn *fprouser*).

Voici un extrait du texte de la fonction C *user_rule*:

```
user_rule(nb, err, err_nb)
   int nb, *err, *err_nb;
   {
   *err_nb = 0;

   if (num >10000)
      fuserrule(&nb, err, err_nb);
   else
   if (num >20000)
      real_mode_user_rule(nb, err, err_nb);
   else
      switch (nb)
      {
      case 201:
            find_pattern(err, err_nb);
            break;
      ...
                  /* add procedure calls here */
      default:
            *err_nb = 500;
      }
   }
```

La terminaison normale du programme externe (qui se traduit par un succès en Prolog) intervient quand le paramètre *err_nb* de la fonction C *user_rule* vaut zéro.

Selon la nature de votre programme externe, vous pouvez choisir de l'insérer directement dans le module *prouser* ou bien de l'écrire dans un fichier séparé.

Si vous avez écrit vos règles externes dans un ou plusieurs modules différents des modules *prouser* et *fprouser*, alors vous devez compiler ce(s) module(s) puis donner le(s) nom(s) au programme *prolink* pour l'édition de liens.

Tel qu'il est livré, le module *prouser* établit les relais d'appel entre le compilateur Prolog et la fonction externe *find_pattern*, qui constitue pour vous un exemple d'écriture de règles externes.

> ***Attention**: Le programme externe* find_pattern *donné en exemple met en œuvre la règle prédéfinie* find-pattern(s1,s2,n). *Sa modification inconsidérée peut introduire un mauvais fonctionnement de cette règle.*

D.4. Comment provoquer un *backtracking* ou renvoyer une erreur

Pour renvoyer une erreur depuis une fonction externe, celle-ci doit positionner le paramètre *err_nb* de la fonction *user_rule.*

Note importante :
A parir de la version 4.1, le paramètre *err* n'est plus significatif, il est ignoré par Prolog. La compatibilité ascendante des versions est toujours assurée.

Si *err_nb* est positionné à une valeur strictement positive, alors une commande équivalente à la commande Prolog *block_exit(err_nb)* est exécutée. Pour y associer un message, ajoutez la ligne correspondante dans le fichier *err.txt*, sous la forme:

```
Numéro  Texte_décrivant_l'erreur
```

La règle prédéfinie *ms_err* permet d'accéder au message associé à un numéro donné. Les numéros d'erreurs inférieurs à 1000 sont réservés pour Prolog, les nombres supérieurs à 1000 sont disponibles pour l'utilisateur. Quelques définitions actuelles de messages d'erreur concernant les modules externes, sont:

```
500  ERREUR DANS UNE REGLE PREDEFINIE
501  ARGUMENT ERRONE DANS LE MODULE UTILISATEUR
502  VALEUR HORS LIMITES DANS LE MODULE UTILISATEUR
503  CHAINE TROP LONGUE DANS LE MODULE UTILISATEUR
520  get_term: TABLEAU TROP PETIT POUR CODER LE TERME
521  get_term: TABLEAU DES CHAINES TROP PETIT
522  get_term: TABLEAU DES REELS TROP PETIT
523  put_term: INDEX ERRONE DANS LE TABLEAU DES TERMES
```

⇒ indique qu'à un tag de type 'T' ou 'D', correspond une valeur en dehors des index possibles pour le tableau *val_tab*.

```
524  put_term: INDEX ERRONE DANS LE TABLEAU DES CHAINES
```

⇒ indique qu'à un tag de type 'S' correspond une valeur en dehors des index possibles pour le tableau des chaînes.

```
525  put_term: INDEX ERRONE DANS LE TABLEAU DES REELS
```

⇒ indique qu'à un tag de type 'R', correspond une valeur en dehors des index possibles pour le tableau des réels.

```
526  put_term: CODAGE DE TUPLE ERRONE
```

⇒ un index de tuple référence une suite dont le premier élément n'est pas de type entier.

```
527  put_term: TAG NON VALIDE
528  put_term: L'ARGUMENT PROLOG N'EST PAS UNE VARIABLE LIBRE
```

Un *backtracking* est provoqué quand on retourne à Prolog avec *err_nb* égal à un entier strictement négatif.

Annexe E
Prolog II+ et les caractères.

E.1. Présentation
E.2. Jeu ISO : avantages et inconvénients
E.3. Jeu hôte : avantages et inconvénients
E.4 Remarques
E.5 Conventions

E.1. Présentation

Il existe deux jeux de caractères utilisables en Prolog II+ :

- un jeu défini par le code ISO 8859-1.
- le jeu de la machine hôte.

La détermination du jeu choisi se fait au lancement d'une session Prolog et est valable jusqu'à la fin de la session (voir le manuel d'utilisation §2.3).

Avoir deux jeux de caractères implique avoir deux modes de codage interne de ces caractères :

- Un mode dans lequel quelque soit le système hôte, le codage interne est invariant, c'est celui du code ISO 8859-1.
- Un second mode dans lequel le codage interne utilisé est celui du système hôte.

Tant que l'on manipule des caractères, il n'y a pas de problème. Cependant si l'on manipule les codes des caractères, seul le jeu ISO assure d'avoir le même code pour toutes les machines. Par exemple en jeu ISO char_code("é",x) a la même valeur sur Macintosh et sur PC, ce n'est pas le cas lorsque l'on choisit le jeu hôte.

Avoir deux modes de codage, c'est aussi avoir deux types de fichiers binaires:

- Des fichiers contenant des chaînes codées d'après le code ISO.
- Des fichiers contenant des chaînes codées avec les caractères du système hôte.

Lors de la sauvegarde, le codage choisi pour la session est mémorisé dans le fichier que l'on crée; un message d'avertissement s'affichera au moment du chargement si le codage mémorisé dans le fichier binaire n'est pas le même que celui choisi pour la session en cours.

D'autre part, en mode ISO, puisque Prolog manipule des données (caractères) qui peuvent ne pas exister sur la machine hôte, il faut leur associer une représentation[1] externe (suite de caractères) pour les visualiser. Il existe deux modes de représentation des caractères accentués n'appartenant pas au jeu hôte:

- Un mode dans lequel ils sont représentés par une séquence accent_escape (cf R1.31) qui est très lisible.
- Un mode dans lequel ils sont représentés par une séquence octale.

Le choix de ce mode se fait au lancement de Prolog et est valable pour toute la session (voir le manuel d'utilisation §2.3).

Dans ce qui suit, nous ferons la distinction entre une représentation interne (binary string), qui est une valeur en mémoire, et la visualisation de cette donnée (print string), c'est à dire sa représentation sous forme de caractères sur un canal d'entrée ou de sortie.

E.2. Jeu ISO : avantages et inconvénients

Le choix du jeu ISO a l'avantage d'assurer la portabilité des modules (sources ou binaires) inter-machines. Il permet également de manipuler des caractères qui n'existent pas forcément sur la machine hôte (par exemple les machines UNIX qui utilisent le jeu ASCII 7 bits), notamment les caractères accentués.

L'inconvénient de ce choix, pour Prolog qui est un langage très ouvert et qui réalise beaucoup d'échanges ou de partage de données, est la différence de codage interne de certaines données (chaînes) entre Prolog et l'extérieur. Ce qui nécessiterait de faire une transformation à chaque communication avec l'extérieur. En fait, l'utilisateur n'en a pas forcément toujours besoin, cela dépend de l'opération qu'il veut réaliser sur ces données. Il peut les vouloir:

- en code ISO.
- en code hôte en tant que donnée (binary string).
- en code hôte en tant que représentation (print string).

En effet, des procédures qui manipulent les chaînes sans en connaître le sens, par exemple `find_pattern` qui compare ou `substring` qui décompose, sont indépendantes d'un code particulier et n'ont donc pas besoin de transformations. Par contre des procédures qui font appel au système hôte en auront besoin.

---

1 La visualisation est une représentation externe à Prolog et donc nécessairement codée par le système hôte; Prolog lui ne manipule que des données, donc des codes ISO. Ce sont les entrées sorties qui font la transformation entre la représentation graphique et le codage interne.

En définitive, il suffira de disposer de procédures de transformation des chaînes entre le codage ISO et le codage hôte, et de savoir quel type de codage est supposé dans les procédures de communication de Prolog.

E.3. Jeu hôte : avantages et inconvénients

Le choix du jeu de la machine hôte a l'avantage de simplifier le passage de la représentation des chaînes dans le langage hôte au codage des chaînes en Prolog II+: il n'est pas nécessaire de transformer les données entre Prolog et l'extérieur.

L'inconvénient de ce choix tient à la pauvreté du jeu ASCII et à la non standardisation des caractères étendus sur l'ensemble des machines, ce qui rend les programmes non portables.

E.4. Remarques

•En utilisant le même codage que celui de la machine hôte, la donnée et sa représentation sont confondues, tandis qu'en utilisant le codage ISO il arrive que la donnée et sa représention soient différentes et que la représentation externe occupe plus d'octets en mémoire que la donnée.

•La première moitié de la table du code ISO est identique à la table du code ASCII US. Si un programme Prolog ne contient pas de caractères étendus, il est inutile de faire des transformations entre ISO et hôte.

•Le fait de vouloir utiliser des caractères qui n'existent pas sur la machine hôte, impose d'une part à choisir le mode ISO, d'autre part, pour les communications, soit à les transmettre en ISO, soit à les transmettre sous forme imprimable (print string).

•Le mode ISO sans accent_escape, comme le mode hôte, réduit le nombre de caractères qui peuvent créer une ambiguité d'interprétation s'ils sont précédés du caractère '\'. Ce mode peut donc éviter aux programmes qui utilisent beaucoup de '\', de masquer ce '\' (utile pour les programmes Edinburgh qui manipulent assez souvent des identificateurs ou *graphic_symbol* contenant des '\').

E.5. Conventions pour le mode ISO

Les données partagées entre Prolog et les langages externes, décrites dans les EXTERNAL_DESCRIPTOR, doivent être codées en ISO.

La fonction PRO_BIND, qui permet de créer dynamiquement une zone de données partagées, référence des données qui doivent être codées en ISO; par contre son premier argument qui est le nom Prolog de la zone de données est une chaîne hôte.

Les échanges de chaînes par pipe - entre Prolog et des coprocessus - sont considérés comme des entrées sorties et doivent donc être des chaînes hôtes sous forme imprimable. Les fonctions concernées sont *send_string* et *receive_string*.

Pour les transferts de données entre Prolog et les langages externes ou vice versa, les conventions sont les suivantes:

get_string, put_string, get_term, put_term:

passent et reçoivent des chaînes ISO.

get_O_string, put_O_string:

ont les mêmes arguments et le même fonctionnement que respectivement *get_string* et *put_string* mais passent et reçoivent des chaînes hôtes sous forme imprimable.

Et enfin voici quatre procédures de traduction de chaînes:

bin_str_iso_to_host(source,dest)[1]

bin_str_host_to_iso(source,dest)[1]

réalisent la conversion d'une chaîne ISO en une chaîne hôte et vice versa, en assimilant les chaînes à des données (binary string); c'est à dire la longueur est conservée, il peut y avoir des caractères non imprimables, et les caractères de la chaîne d'entrée qui n'existent pas dans le jeu de sortie sont remplacés par le '?'.

print_str_iso_to_host(source,dest)[1]

print_str_host_to_iso(source,dest)[1]

réalisent la conversion d'une chaîne ISO en une chaîne hôte et vice versa, où la chaîne hôte est la représentation (printable string) de la chaîne ISO; la longueur des deux chaînes n'est pas forcément la même, un caractère peut être représenté par une chaîne à quatre caractères.

---

[1] *source* est l'adresse d'une zone mémoire où se trouve la chaîne de référence terminée par le caractère nul. *dest* est l'adresse de la zone mémoire où Prolog va copier la chaîne résultat, à charge à l'utilisateur de réserver la place nécessaire.

Annexe F
Table ISO 8859-1

	0	1	2	3	4	5	6	7
0			SP	0	@	P	`	p
1			!	1	A	Q	a	q
2			"	2	B	R	b	r
3			#	3	C	S	c	s
4			$	4	D	T	d	t
5			%	5	E	U	e	u
6			&	6	F	V	f	v
7			'	7	G	W	g	w
8			(	8	H	X	h	x
9			)	9	I	Y	i	y
A			*	:	J	Z	j	z
B			+	;	K	[	k	{
C			,	<	L	\	l	|
D			-	=	M	]	m	}
E			.	>	N	^	n	~
F			/	?	O	_	o	

	8	9	A	B	C	D	E	F
0			NBSP	°	À	Ð	à	ð
1			¡	±	Á	Ñ	á	ñ
2			¢	2	Â	Ò	â	ò
3			£	3	Ã	Ó	ã	ó
4			€	´	Ä	Ô	ä	ô
5			¥	µ	Å	Õ	å	õ
6			Š	¶	Æ	Ö	æ	ö
7			§	·	Ç	×	ç	÷
8			¨	¸	È	Ø	è	ø
9			©	š	É	Ù	é	ù
A			ª	º	Ê	Ú	ê	ú
B			«	»	Ë	Û	ë	û
C			¬	1/4	Ì	Ü	ì	ü
D			SHY	1/2	Í	Ý	í	ý
E			®	3/4	Î	Þ	î	þ
F			¯	¿	Ï	ß	ï	ÿ

PROLOG II+

MANUEL D'UTILISATION
WINDOWS

Table des matières

Table des matières iii

Avant-Propos vii

1. Guide d'installation sous l'environnement Windows ..... U 1 - 1
1.0. Préambule ..... U 1 - 1
1.1. Matériel et logiciel requis ..... U 1 - 1
1.1.1. Pour lancer Prolog ..... U 1 - 1
1.1.2. Pour ajouter des extensions (méthode standard, en 32 bits) ..... U 1 - 1
1.1.3. Pour ajouter des extensions en code 16 bits ..... U 1 - 2
1.1.4. Pour créer des applications et les diffuser ..... U 1 - 2
1.2. Contenu du kit d'installation ..... U 1 - 2
1.2.1. Fichiers indispensables pour utiliser Prolog ..... U 1 - 2
1.2.2. Fichiers annexes pour l'utilisation de Prolog ..... U 1 - 3
1.2.3. Fichiers pour les extensions standards (32 bits) ..... U 1 - 4
1.2.4. Extensions en code16 bits (réservé Windows 3) ..... U 1 - 5
1.2.4.1. Principe des extensions 16 bits ..... U 1 - 5
1.2.4.2. Fichiers pour les extensions 16 bits ..... U 1 - 6
1.2.4.3. Exemples fournis ..... U 1 - 6
1.2.5. Fichiers d'exemples ..... U 1 - 7
1.2.6. Conventions pour les suffixes des fichiers ..... U 1 - 7
1.3. Procédure d'installation ..... U 1 - 8
1.4. Modification de l'environnement d'exécution ..... U 1 - 8

2. Utilisation de Prolog II+ Windows ..... U 2 - 1
2.1. Lancement et arrêt de Prolog ..... U 2 - 1
2.1.1. Activation, fichiers par défaut ..... U 2 - 2
2.1.2. Arrêt de Prolog ..... U 2 - 2
2.2. Espaces et tailles par défaut ..... U 2 - 2
2.3. Syntaxe de la ligne de commande ..... U 2 - 3
2.4. Création et exécution d'un programme ..... U 2 - 10
2.5. Interruption d'un programme ..... U 2 - 11
2.6. Construction et lancement de Prolog avec graphisme ..... U 2 - 11

2.7. Compilation et édition de liens ..... U 2 - 12

3. Spécificités de Prolog II+ Windows ..... U 3 - 1
3.1. Valeurs extrêmes des constantes arithmétiques ..... U 3 - 1
3.2. Les caractères dans la syntaxe Prolog II+ ..... U 3 - 1
3.3. Personnalisation d'une Application ..... U 3 - 2
3.4. Modification du module principal ..... U 3 - 4
3.5. Extensions en mode 16 bits (Réservé Windows 3) ..... U 3 - 4
3.6. Utilisation de PrologII+ sous forme de DLL ..... U 3 - 6

4. Communication avec une application, en utilisant le protocole DDE ..... U 4 - 1
4.1. Aperçu du protocole DDE ..... U 4 - 1
4.1.1. Les trois niveaux d'identification du protocole DDE ..... U 4 - 2
4.1.2. Les possibilités du protocole DDE Prolog ..... U 4 - 3
4.2. Utilisation simple de Prolog II+ comme serveur DDE ..... U 4 - 4
4.2.1. Initialisation du serveur Prolog II+ ..... U 4 - 4
4.2.2. Transmission d'une requête à Prolog II+ ..... U 4 - 5
4.2.3. Exécution d'un but Prolog avec récupération des solutions ..... U 4 - 6
4.2.4. Schéma du Serveur Prolog II+ ..... U 4 - 7
4.2.5. Exemples ..... U 4 - 7
4.3. Programmer en Prolog II+ un applicatif Client ou Serveur ..... U 4 - 8
4.3.1. Programmer un applicatif Client DDE ..... U 4 - 9
4.3.2. Programmer un applicatif Serveur DDE ..... U 4 - 11

5. Primitives Graphiques ..... U 5 - 1
5.1. Description du système graphique ..... U 5 - 2
5.1.1. Evénements ..... U 5 - 2
5.1.2. Objets graphiques ..... U 5 - 2
5.1.3. Configuration du graphique Prolog II+ au démarrage ..... U 5 - 5
5.1.4. Conventions utilisées dans ce chapitre ..... U 5 - 6
5.2. Primitives de gestion des fenêtres ..... U 5 - 7
5.2.1. Création, destruction d'une fenêtre ..... U 5 - 7
5.2.2. Configuration, manipulation d'une fenêtre ..... U 5 - 10
5.2.3. Rafraîchissement des zones graphiques ..... U 5 - 12
5.3. Primitives élémentaires de gestion des objets attachés à une fenêtre ..... U 5 - 13
5.3.1. Création, destruction d'objets ..... U 5 - 13
5.3.2. Configuration d'objets ..... U 5 - 17
5.3.3. Gestion des événements ..... U 5 - 20
5.4. Primitives spécifiques pour la gestion de menu ..... U 5 - 21

5.4.1. Sauvegarde et changement de la barre de menu..... U 5 - 21
5.4.2. Description d'un menu ..... U 5 - 22
5.5. Mode de dessin et d'écriture.....U 5 - 25
5.6. Dessin et positionnement.....U 5 - 29
5.7. Position de la souris dans une zone graphique.....U 5 - 33
5.8. Primitives spéciales de saisie.....U 5 - 34
5.8.1. Simulation de boutons..... U 5 - 34
5.8.2. Affichage de message avec validation..... U 5 - 35
5.8.3. Saisie de texte..... U 5 - 35
5.8.4. Boites de choix..... U 5 - 37
5.8.5. Choix de fichier..... U 5 - 39
5.8.6. Choix d'un noeud d'un arbre ..... U 5 - 41
5.9. Règles pour gérer des objets structurés.....U 5 - 41
5.10. Envoi d'événements à Prolog depuis un objet externe.....U 5 - 48

Avant-Propos

La documentation concernant Prolog II+ a été regroupée en deux manuels :

- le manuel de référence, qui décrit de manière précise le langage et son utilisation. Ce manuel est valable pour toutes les implantations.

- le manuel d'utilisation, qui décrit tout ce qui est dépendant d'un ordinateur ou d'un système d'exploitation spécifique. Il y a donc un manuel d'utilisation par matériel. Celui-ci concerne la version WINDOWS.

Le manuel d'utilisation précise comment se réalisent de manière pratique sur votre ordinateur, certaines fonctions décrites dans le manuel de référence.

Le chapitre premier décrit l'installation de Prolog sur votre machine.

Le deuxième chapitre concerne le lancement de Prolog, la gestion des états sauvés et la structure des noms de fichiers.

Le troisième chapitre de ce manuel décrit ce qui est spécifique à cette version: ce qu'il y a en plus et en moins par rapport à la version de base, ainsi que les valeurs extrêmes des constantes.

Le quatrième chapitre décrit la communication avec une autre application en utilisant le protocole DDE.

Le dernier chapitre décrit la bibliothèque de primitives graphiques, spécifique à chaque implantation.

Dans les exemples de ce manuel, ce qui a été affiché par l'ordinateur apparaît en caractères "`machine à écrire`" droits, tandis que ce qui doit être tapé par l'utilisateur apparaît en *`caractères machine à écrire penchés`*.

1. Guide d'installation sous l'environnement Windows

1.0. Préambule
1.1. Matériel et logiciel requis
1.2. Contenu du kit d'installation
1.3. Procédure d'installation
1.4. Modification de l'environnement d'exécution

Ce chapitre décrit la procédure d'installation du compilateur Prolog II+ en tant qu'Application dans l'environnement Windows. Il est recommandé de le lire en entier avant de commencer l'installation.

1.0. Préambule

Prolog II+ pour Windows est maintenant une véritable application 32 bits qui fonctionne aussi bien sur Windows NT, Windows 3.1 étendu avec le kit Win32s, et Windows 95.

Pour les utilisateurs Windows 3, Prolog II+ offre toujours la possibilité d'extensions en C 16 bits, extensions désormais isolées dans une DLL, donc ne nécessitant que le SDK 16 bits classique.

1.1. Matériel et logiciel requis

1.1.1. Pour lancer Prolog

Au minimum un ordinateur PC 386 équipé avec 4 Mo de mémoire, et doté de Windows version 3.1 augmenté du kit Win32s, sachant que le système standard standard actuel serait plutôt Windows 95 ou Windows NT.

1.1.2. Pour ajouter des extensions (méthode standard, en 32 bits)

Tous les moyens d'extension de Prolog sont utilisables ici. L'outil de développement recommandé est le "MicroSoft Visual C++ version 2.xx". Tout autre SDK compatible est utilisable.

1.1.3. Pour ajouter des extensions en code 16 bits

Cette méthode est destinée aux utilisateurs de Windows 3, la DLL d'extension n'étant pas reconnue par Windows NT et Windows 95. Seule la méthode des parasites décrite en annexe est disponible ici, le code d'extension prenant place dans une DLL chargée automatiquement par Prolog. Pour construire cette DLL, une librairie statique qui inclut les fonctions de communication (*get_integer*...) est fournie. Elle est compilée avec le "CL version 8.00" du "MicroSoft Visual C++ version 1". Tout autre SDK compatible est utilisable.

Cette DLL a accès à l'ensemble de l'API 16 bits de Windows.

1.1.4. Pour créer des applications et les diffuser

Pour créer des applications et les diffuser, il vous faut commander le kit suivant:

Run Time Prolog II+ sous Windows. Pour Windows 3, vous devrez aussi redistribuer le kit Win32s.

1.2. Contenu du kit d'installation

Note concernant Windows 3

Dans le cas de Windows 3, la première chose à faire est d'installer le kit Win32s fourni. Il est doté de sa procédure d'installation autonome et devra être redistribué tel quel à vos propres clients utilisant Windows 3. Le jeu "FreeCall" fourni avec ce kit vous permettra de vérifier son bon fonctionnement.

Le *kit* d'installation de Prolog II+ sous Windows consiste en un volume comportant les fichiers suivants:

1.2.1. Fichiers indispensables pour utiliser Prolog

err.txt

Fichier contenant les messages d'erreurs en anglais, et consulté par Prolog lors de l'appel de la règle *ms_err* si l'option -r n'a pas été utilisée pour en choisir un autre. Pour avoir les messages en français, il faut choisir le fichier *fr_err.txt.*

fonts.usr

Fichier de définition des fontes utilisées dans l'environnement graphique par PrologII+. L'utilisateur peut intervenir sur ce fichier avant le démarrage pour définir les fontes *:ft(1), :ft(2)*... (utilisées par exemple par le prédicat *gr_font*).

initial.po

Fichier contenant le code sauvé du superviseur.

prolog.exe

Fichier exécutable de la version de développement de Prolog II+.

1.2.2. Fichiers annexes pour l'utilisation de Prolog

*customiz.dl_*

DLL contenant des ressources de personnalisation de Prolog II+. Automatiquement reconnue si renommée en *customiz.dll*.

dde.mo

Module objet Prolog à charger pour l'écriture d'un applicatif client ou serveur utilisant le protocole DDE.

dessin.m2, dessin.mo

Fichiers source et objet du module de dessin d'arbres.

edinburg.mo

Module objet chargé automatiquement lorsque l'on active la syntaxe Edinburgh. Contient toutes les règles prédéfinies spécifiques à la syntaxe Edinburgh. Le chargement de ce module dans une session Prolog en syntaxe Marseille provoque le passage en mode de compatibilité Edinburgh.

fr_err.txt

Fichier contenant les messages d'erreurs en français.

graphic.mo

Module objet Prolog contenant toutes les primitives graphiques. Fichier à charger pour lancer l'environnement graphique de Prolog.

graphstr.mo

Module objet Prolog utilisé automatiquement par l'environnement graphique en fonction de la configuration choisie au démarrage de Prolog.

obdialog.mo

Module objet Prolog à charger pour utiliser la primitive de gestion d'objets graphiques structurés : *gr_dialog*. Est automatiquement chargé par le chargement du fichier *graphic.mo*.

prolog2.pre

Fichier texte contenant la ligne de commande Prolog II+. Si ce fichier existe, les options qui y sont spécifiées remplacent les options par défaut. Ce fichier est d'abord recherché dans le répertoire courant, puis dans le répertoire défini par la variable d'environnement *PrologDir2*. Il est éditable par n'importe quel processeur de texte.

tools

Répertoire d'outils comme:

dbgbase.mo

Module objet à recharger pour avoir des messages d'erreur plus complets concernant les règles prédéfinies de la syntaxe Prolog II.

dbgedin.mo

Module objet à recharger pour avoir des messages d'erreur plus complets concernant les règles prédéfinies de la syntaxe Edinburgh. Peut être chargé directement à la place de *edinburg.mo*.

dbggraph.mo

Module objet à recharger pour avoir des messages d'erreur plus complets, concernant les règles prédéfinies graphiques. Peut être chargé directement à la place de *graphic.mo*.

int_edit.mo

Module objet Prolog à recharger (prédicat *reload*) pour pouvoir utiliser l'éditeur intégré, à la place de l'éditeur externe défini par la variable d'environnement *PrologEdit*.

1.2.3. Fichiers pour les extensions standards (32 bits)

Tous les fichiers cités ci-dessous, servent à la reconstruction de Prolog II+. Les fichiers objets et la librairie sont indispensables et constituent l'exécutable final : *prolog.exe.*

princip.c, princip.obj

Fichiers source et objet contenant le lancement du but principal de Prolog.

proentry.c, proentry.obj

Fichiers source et objet contenant la routine d'entrée standard de Prolog II+ (WinMain). Le fichier *proentry.obj* peut être remplacé par un programme utilisateur utilisant Prolog comme un sous-programme.

proext.h

Fichier source C, contenant des déclarations de macros et de structures, à inclure dans les modules utilisateur écrits en C.

prolink.bat

Fichier de commandes pour refaire l'édition de liens et reconstruire l'exécutable. Il peut être modifié par l'utilisateur.

prolog.def

Fichier de description de module. Est utilisé lors de l'édition de liens. L'utilisateur doit y rajouter les noms des Callbacks qu'il désire exporter s'il y a lieu (voir SDK Windows).

prolog.lib

Librairie contenant les fonctions de Prolog.

prolog.res

Fichier compilé des ressources de Prolog.

prouser.c, prouser.obj

Fichiers source et objet de l'interface externe C.

use_win.h

Fichier source à inclure obligatoirement dans les modules utilisateur écrits en C pour assurer leur compatibilité avec l'environnement fenétré de prolog.

1.2.4. Extensions en code16 bits (réservé Windows 3)

1.2.4.1. Principe des extensions 16 bits

Les extensions 16 bits sont localisées dans une DLL réalisable au moyen du SDK standard de Windows 3, sans modification de l'exécutable 32 bits. Cette DLL est chargée automatiquement dès l'invocation d'une règle d'extension située dans la plage réservée, et déchargée lorsque Prolog se termine. Sa localisation obéit aux règles suivantes:

La DLL doit se trouver au même endroit que l'exécutable 32 bits de Prolog, et son nom **doit être le même, sauf le dernier caractère qui est remplacé par un '-', et l'extension qui est ".DLL"**.

exemple: à c:\prolog\prolog.exe correspond: c:\prolog\prolo-.dll

Attention: il est possible de lancer plusieurs instances d'une application Prolog, la DLL d'extensions 16 bits sera alors partagée. Ceci peut entraîner des conflits au niveau des données privées que la DLL maintient.

Lors de l'utilisation des primitives de communication avec Prolog, il est utile de savoir que les pointeurs passés sont convertis mais que les données pointées ne sont pas copiées. L'opération est donc relativement performante.

1.2.4.2. Fichiers pour les extensions 16 bits

usermpty.c

Fichier d'extension vierge, à compléter par vos fonctions, qui peuvent également figurer dans un source séparé ou des librairies statiques. Ce fichier fournit aussi les fonctions standards d'initialisation et de terminaison de la DLL, permettant d'y placer éventuellement les votres.

callpro.h

Fichier fournissant les prototypes des primitives de communication, à inclure.

callpros.lib, callprol.lib

Librairie statique fournissant le code de liaison à Prolog, disponible en modèle Small (*callpros*) et Large (*callprol*). L'une des deux versions doit être liée à votre code.

callpro.def

Fichier de définition de module standard pour la DLL, utilisé par l'éditeur de liens. Vous devrez éventuellement en modifier les paramètres et y rajouter vos propres exports. Les ordinaux sont sans importance, les fonctions de connexion étant reconnues par leur nom.

userdll.mak

Exemple de makefile pour la DLL, à adapter.

1.2.4.3. Exemples fournis

Les DLLs d'extensions réalisées en exemple doivent donc être copiées auprès de l'exécutable et renommées pour être reconnues. Une seule est donc utilisable à la fois:

tutorial.c, tutorial.mak, tutorial.p2 -> tutorial.dll

Exemple didactique minimal décrivant les règles essentielles à respecter.

userfunc.c,, userfunc.mak, userfunc.p2 -> userfunc.dll

Exemple conséquent fournissant diverses règles qui exploitent la plupart des primitives de communication.

userxtrn.c, userxtrn.mak, userxtrn.p2 -> userxtrn.dll,

Exemple connectant lui-même une seconde DLL 16 bits, et faisant appel à l'API Windows. Il s'agit de la transposition en 16 bits de l'exemple des objets externes, tel que fourni en 32 bits. Les sources de la DLL chargée sont regroupés dans un sous répertoire.

1.2.5. Fichiers d'exemples

Les exemples se trouvent dans le sous-répertoire *examples* du Kit Prolog.

Exemples de programmation en prolog commentés en Annexe:

automat.p2	sur les arbres infinis
database.p2	sur l'évaluation d'une formule
deriv.p2	sur la dérivation formelle
grammar.p2	sur les grammaires
menu.p2	la composition d'un menu
money.p2	un casse-tête
mutant.p2	les mutants
path.p2	la construction d'un chemin

Autres exemples :

D'autres exemples concernant l'environnement graphique sont donnés, ainsi que des exemples d'ajout de prédicats externes auto-documentés.

1.2.6. Conventions pour les suffixes des fichiers

Pour les fichiers et leur signification les conventions suivantes sont utilisées:

.c	Suffixe des fichiers contenant des sources écrits en C.
.obj	Suffixe des fichiers contenant des fichiers objet .
.d2	Suffixe des fichiers ne contenant que des définitions de macros Prolog II+.
.m2	Suffixe des fichiers contenant un module source Prolog II+ écrit en syntaxe Prolog II.
.m2E	Suffixe des fichiers contenant un module source Prolog II+ écrit en syntaxe Edinburgh.
.mo	Suffixe des fichiers contenant le code d'un module Prolog compilé (attention, ils ne sont pas compatibles avec un éditeur de liens du système, seul Prolog peut les manipuler).
.po	Fichiers de format identique au précédent, mais contenant un état complet (fichiers sauvés avec la commande *exit*). Seuls ces fichiers peuvent être utilisés comme état initial pour le démarrage d'une session Prolog.
.p2	Suffixe d'un fichier source contenant des règles Prolog II+ écrites en syntaxe Prolog II. Le fichier est chargé avec la commande *insert* lorsque la syntaxe courante est la syntaxe Prolog II.

.p2E Suffixe d'un fichier source contenant des règles Prolog II+ écrites en syntaxe Edinburgh. Le fichier est chargé avec la commande *consult* lorsque la syntaxe courante est la syntaxe Edinburgh.

.pex Suffixe d'un fichier source contenant des commandes Prolog. Doit être utilisé par la commande *input*.

.txt Suffixe de fichiers contenant du texte quelconque.

Les fichiers livrés dont les noms se terminent par *.p2* contiennent les textes sources des programmes donnés en exemple en Annexe.

1.3. Procédure d'installation

Le volume de distribution de Prolog II+ a été confectionné par compression d'une arborescence. Exécuter la commande de chargement indiquée sur la feuille de description accompagnant le kit de distribution après s'être placé dans le répertoire où l'on désire installer les fichiers.

Il est conseillé de créer un répertoire réservé à Prolog, par exemple *c:\prolog,* dans lequel seront installés les fichiers du kit. Il faut alors définir la variable d'environnement qui est utilisée par Prolog pour accéder à ces fichiers (voir § 1.4).

1.4. Modification de l'environnement d'exécution

Pour une meilleure utilisation de Prolog, les variables suivantes doivent être définies dans votre environnement d'exécution:

PrologDir2

Chemin utilisé par Prolog si les fichiers suivants ne sont pas définis dans le répertoire courant: *initial.po*, *edinburg.mo, err.txt*. La variable concaténée au nom du fichier doit donner son chemin complet.

PrologEdit

Nom de votre fichier de commande qui lancera l'éditeur de textes devant être utilisé par Prolog (prédicats *edit*, *editm*) lors de l'édition avec un éditeur de la machine hôte.
La commande suivante doit pouvoir être comprise par l'interpréteur de commandes:

%PrologEdit% *nom-de-fichier*

Il n'y a pas de valeur par défaut pour cette variable.
Note: Sous Windows3, votre fichier de commande devra en plus, à la fin, créer un fichier de nom *file.end* dans un répertoire défini par la variable d'environnement de nom *TEMP*, ceci afin d'indiquer la terminaison de l'édition. Si vous désirez par exemple utiliser l'éditeur *edlin* du DOS, vous devez construire un fichier de commande (*myedit.bat*) ayant la forme:

```
rem begin
edlin %1
echo >> file.end
rem end
```

Si vous désirez par contre utiliser l'éditeur intégré plutôt qu'un éditeur externe, vous devez exécuter le prédicat suivant:

```
> reload("int_edit.mo");
```

2. Utilisation de Prolog II+ Windows

2.1. Lancement et arrêt de Prolog
2.2. Espaces et tailles par défaut
2.3. Syntaxe de la ligne de commande
2.4. Création et exécution d'un programme
2.5. Interruption d'un programme
2.6. Construction et lancement de Prolog avec graphisme
2.7. Compilation et édition de liens

Prolog II+ est le compilateur d'une version avancée du langage Prolog. Le compilateur Prolog II+ Windows vous permet de charger, d'exécuter, d'étendre ou de modifier interactivement un programme Prolog.

Le compilateur est lancé avec un état initial. Lorsque l'on sort de Prolog avec la commande *exit*, l'état courant du programme est sauvegardé dans un fichier de code binaire, qui pourra être utilisé plus tard comme nouvel état initial. Le fichier *initial.po* qui vous est livré est un état initial neuf, c'est-à-dire constitué uniquement d'un superviseur et des règles prédéfinies.

Ce chapitre fournit une description de la façon de lancer le compilateur Prolog II+ Windows, d'y créer un programme et d'en sortir.

Ce chapitre est destiné également à décrire la façon d'utiliser les autres outils du kit Prolog II+.

2.1. Lancement et arrêt de Prolog

Pour commencer une session, lancez l'exécutable Windows *prolog.exe* par un moyen quelconque (à travers le Gestionnaire de Programmes, à partir d'un shell...).

Pour terminer une session, lancez l'un des buts Prolog: *quit;* ou *exit;*, ou utilisez l'option Quitter du menu principal.

2.1.1. Activation, fichiers par défaut

Si aucun nom de fichier n'est donné dans la commande de lancement, le nom de fichier *initial.po* est pris par défaut pour le fichier état initial. Exemple sous MS/DOS:

win prolog

Cette commande lance le compilateur avec l'état initial *initial.po* recherché d'abord dans le répertoire courant, puis avec le chemin indiqué dans la variable d'environnement *PrologDir2*. Une erreur survient si un tel fichier n'existe pas. (La commande Prolog *exit* sauvera l'état du programme dans un fichier appelé *prolog.po* dans le répertoire courant.)

win prolog myfile.po

Cette commande lance le compilateur avec l'état initial *myfile.po*. Une erreur survient si un tel fichier n'existe pas.

Au lancement, Prolog crée sa fenêtre Application et la fenêtre Console, puis affiche sa bannière et des informations sur sa configuration puis charge l'état binaire de démarrage.

Le chargement terminé, Prolog se met en attente de commande. Pour cela il affiche un prompt et attend une suite de buts pour l'exécuter. Prolog fonctionnera dans ce mode jusqu'à sa terminaison, produite par l'exécution du but *quit* ou *exit*.

2.1.2. Arrêt de Prolog

Lorsque l'on sort de Prolog en exécutant la règle prédéfinie *quit* ou par un menu, aucun état n'est sauvegardé.

Lorsque l'on sort de Prolog en exécutant la règle prédéfinie *exit*, l'état courant est sauvegardé sous le nom *prolog.po*.

Lorsque l'on sort de Prolog en exécutant la règle prédéfinie *exit("nom-de-fichier")*, alors c'est ce dernier nom qui sera utilisé dans tous les cas pour l'état sauvé produit.

2.2. Espaces et tailles par défaut

L'espace de travail de Prolog est subdivisé en plusieurs espaces. Par défaut, les tailles de ces espaces sont lues dans le fichier binaire initial, à moins d'utiliser l'option de la ligne de commande qui permet de changer le moyen de déterminer ces valeurs initiales qui sont alors:

espace du code

C'est l'espace où est chargé l'état initial et le code généré par le compilateur. La taille par défaut est de 500 Ko. Il n'y a pas de limite sur la taille des modules, hormis la contrainte que la somme des tailles des modules soit inférieure à l'espace total pour le code.
Lorsque des règles (à fortiori des modules) sont supprimées, l'espace est automatiquement récupéré pour les règles qui ne sont pas en cours d'utilisation.

espace du dictionnaire

Dictionnaire des identificateurs Prolog. Taille par défaut: 200Ko.

espace de la pile de récursion (stack)

Contient les variables locales de la démonstration en cours. Taille par défaut: 100Ko. Il n'y a pas de consommation d'espace dans cette pile dans le cas de récursion terminale (voir chapitre *Contrôle de l'effacement des buts* du manuel de référence).

espace de la pile de copie (heap)

Le compilateur Prolog II+ utilise une technique de copie de structures, permettant la mise en oeuvre d'un "garbage collector" très performant. La taille par défaut est de 100Ko.

espace de la pile de restauration (trail)

Pile de mémorisation des valeurs à restaurer lors des retours arrière (backtracking) dans l'exécution en cours. Taille par défaut: 50 Ko. Le "garbage collector" permet la récupération d'espace dans cette pile en cas de débordement.

espace pour les appels directs en C

C'est un espace de travail pour le codage des arguments lors des appels directs de procédures ou fonctions C par la primitive *callC*. Taille par défaut: 1 Ko.

Ces tailles par défaut peuvent être modifiées individuellement par des options sur la ligne de commande ou dans le fichier *prolog2.pre*.

2.3. Syntaxe de la ligne de commande

Dans toutes les descriptions syntaxiques de ce manuel, les éléments facultatifs sont mis entre crochets: [ ], ceux qui peuvent apparaître un nombre quelconque de fois -éventuellement aucune- sont écrits entre accolades: { }.

Il est possible de mémoriser la commande de lancement de Prolog en l'incluant dans un fichier nommé *prolog2.pre* qui peut se trouver dans le répertoire courant ou dans le répertoire d'installation de Prolog. En effet, si un tel fichier existe, Prolog traitera toutes les options de ce fichier ne figurant pas déjà sur la ligne de commande (qui a donc priorité). Seule la première ligne est lue, les autres lignes sont ignorées. Le premier mot est aussi ignoré (nom du programme).

La syntaxe de la commande Prolog est:

```
win (sous MS-DOS) prolog [parametres]  [etat_initial]
```

Le nom de fichier *etat_initial*, donné dans la syntaxe du système hôte, désigne le fichier d'état avec lequel on souhaite démarrer la session Prolog. Si rien n'est spécifié, le nom *initial.po* est pris par défaut.

Les paramètres peuvent être spécifiés dans un ordre quelconque, ils permettent de modifier les tailles et les options par défaut du compilateur. La liste des paramètres possibles est:

```
[-H] [-c n] [-C file] [-d n] [-E] [-f fv{fv}] [-h n] [-i file] [-
j file] {-m file} [-M n] [-o file] [-P parameter] [-q n] [-Q] [-r
file] [-R n] [-s n] [-t n]
```

Signification des paramètres:

-H

(Help) Affiche au démarrage de Prolog, l'aide de la ligne de commande. Pour chaque option affiche également la configuration par défaut.

-c entier

(code) Définit la taille en Ko utilisée pour l'espace des règles, des variables statiques, et des tableaux Prolog.

-C file

(Compilation) Compile le fichier source de nom *file*, génère un fichier objet et quitte Prolog. La détermination du nom du fichier objet suit les règles suivantes :

- Si l'option *-o* est utilisée simultanément, c'est le nom désigné dans cette option qui sera utilisé ;
- Si le fichier source ne comporte aucun module, c'est le nom *empty.mo* qui sera utilisé ;
- Si le fichier source ne comporte qu'un seul module, c'est ce nom de module, suffixé par *.mo* qui sera utilisé (si cet unique module est le module vide, c'est le nom *user.mo* qui sera utilisé ;
- Si le fichier source comporte plusieurs modules, c'est le nom du fichier source suffixé par *.mo*. qui sera utilisé.

-d entier

(dictionnaire) Définit la taille en Ko utilisée pour le dictionnaire.

-E

(Edinburgh) active Prolog en mode de compatibilité Edinburgh (cf Chapitre 10 du manuel de référence). Est équivalent à l'option *-m edinburg.mo*. Change la syntaxe et charge les nouvelles règles prédéfinies spécifiques au mode Edinburgh. La syntaxe Edinburgh suppose que la syntaxe des variables n'est pas la syntaxe Prolog, et la syntaxe des réels est la syntaxe standard. Par conséquent l'utilisation de cette option, modifie la valeur des flags *s*, *v*, et *r* qui vaudra respectivement *E*, *E* ou *U* et *S*. Le traitement de cette option est réalisé en dernier. Par conséquent, si par l'option *-f* les flags *s* et *r* sont spécifiés, ils seront ignorés et le flag *v* défini à *P* sera également ignoré.

-f fv{fv}

(flags) Définit les valeurs des options de comportement de Prolog. Les valeurs par défaut des options sont équivalentes au paramètre:

-f a1A0cIe0g1G0i1o1rPsSuWvPw1zU

fv{fv} est une suite de couples de caractères où dans chaque couple, le premier caractère définit l'option concernée, le second définit sa valeur.
Les options disponibles sont:

option	*valeurs possibles*
a	0,1
A	0,1,c,C,d,D,h,H,s,S,t,T
c	I,M
e	0,1
g	0,1
G	0,1,c,C,d,D,s,S
i	0,1
o	0,1
r	P,S
s	A,C,I,S
u	E,F,W
v	E,P,U
w	0,1,2
z	U,D

a: (atomes préfixés)
La valeur détermine si l'on doit attribuer automatiquement le préfixe vide ("") aux atomes en position d'argument et non explicitement préfixés :

0 Ces atomes ne seront pas préfixés (attribution du préfixe vide).

1 Ces atomes seront préfixés en fonction du contexte.

A: (réallocation)
La valeur détermine si l'on doit interdire ou permettre la réallocation de l'espace concerné:

1 Toutes les réallocations sont interdites

0 Toutes les réallocations sont permises

c (resp. C) La réallocation du code est interdite (resp. permise)

d (resp. D) La réallocation du dictionnaire est interdite (resp. permise)

h (resp. H) La réallocation de la pile de copie (heap) est interdite (resp. permise)

s (resp. S) La réallocation de la pile de récursion (stack) est interdite (resp. permise)

t (resp. T) La réallocation de la pile de restauration (trail) est interdite (resp. permise)

c: (codage des caractères)
La valeur détermine le type de codage interne des caractères:

I codage selon le jeu ISO 8859-1.

M codage selon le jeu de la machine hôte.

e: (escape mode)
Détermine le mode d'impression des caractères accentués n'appartenant pas au jeu de la machine hôte. Ce flag n'a de signification qu'en mode de codage ISO. Les valeurs possibles sont:

0 le caractère est écrit comme une séquence *format_escape* -\code octal- (cf R1.29). Cette option a la particularité de réduire l'ensemble des caractères qui peuvent créer une ambiguité pour l'écriture du "\" (cf R1-7).

1 le caractère est écrit comme une séquence *accent_escape* (cf R1.29 à R1.31).

g: (garbage collection information)
Si la valeur est 1 (option par défaut), un message est imprimé chaque fois qu'il y a récupération mémoire, sinon aucune information n'est imprimée.

G: (garbage collection)
La valeur détermine si l'on doit interdire ou permettre la récupération de mémoire dans l'espace concerné:

1 Toutes les récupérations sont interdites

0 Toutes les récupérations sont permises

c (resp. C) La récupération dans le code est interdite (resp. permise)

d (resp. D) La récupération dans le dictionnaire est interdite (resp. permise)

s (resp. S) La récupération dans les piles est interdite (resp. permise)

i: (interprète la lecture du caractère spécial '\')
Si la valeur est 1 (option par défaut), le ou les caractères suivant le '\' sont spécialement interprétés (par exemple la séquence *t* signifie le caractère de tabulation). Si la valeur est 0, le caractère '\' perd sa valeur de caractère spécial, il n'est plus possible de le composer avec un *format_escape* ou un *accent_escape* pour représenter un autre caractère, quelle que soit la valeur de l'option *e*.

o: (optimisation)
Optimisation du code généré (pour: les instructions arithmétiques, les instructions de comparaison, les tests de type, *block*, *val*, *assign*.). Lorsque cette option est activée, les opérations arithmétiques peuvent donner, lorsqu'on décompile une règle (avec *rule* par exemple), une suite de termes différente de la suite originale. Il en est de même pour le prédicat *block*, pour lequel la décompilation donnera, dans certains cas, un terme équivalent mais non identique. Le debugger peut ne pas visualiser certaines opérations arithmétiques, ou certains tests de type, ou encore les arguments de certains prédicats prédéfinis optimisés tels que *val*, *assign* ou *block*. La valeur détermine si l'optimisation est active ou pas.

0 (pas d'optimisation) Le programme apparaît sous sa forme originale lorsqu'on décompile ou lorsqu'on utilise le debugger.

1 (optimisation) C'est la valeur par défaut. L'option s'applique pour tout nouveau code produit par *insert*, *module* ou *assert*.

r: (real)
La valeur détermine la syntaxe à utiliser pour les réels.

P (Prolog) Notation Prolog II (permet l'utilisation simultanée avec la notation infixée des listes).

S (Standard) Notation standard (les réels y sont écrits sous forme abrégée et les listes en notation "crochets". Ex: [1, 2, 3]).

s: (string)
Cette option n'est prise en compte qu'en mode Edinburgh. La valeur définit l'interprétation syntaxique de l'unité lexicale *string* décrite au § 1.4 du manuel de référence.

A (Atom) L'unité représente un identificateur. Il sera préfixé en fonction du contexte courant de lecture.

C (Character) L'unité représente une liste d'identificateurs d'une lettre. Ils seront préfixés en fonction du contexte courant de lecture.

I (Integer) L'unité représente une liste d'entier égaux respectivement au code (Machine ou ISO, en fonction de l'option choisie) de chaque caractère de la chaîne.

S (String) L'unité représente une chaîne de caractère Prolog.

u: (undefined)
La valeur détermine le comportement lors d'une tentative d'effacement d'une règle non définie. Les valeurs possibles sont:

E (Error) Impression du but en cause et arrêt.

F (Fail) Même comportement que l'interpréteur: échec.

W (Warning) Impression d'un message et échec.

v: (variables)
La valeur détermine la syntaxe à utiliser pour les variables et les identificateurs.

E (Edinburgh) Notation type Edinburgh.

P (Prolog II+) Notation type Prolog II+.

U (Underlined variables) Seules les suites alpha-numériques commençant par "_" dénotent des variables. C'est la forme de base (voir le manuel de référence, chapitre 1).

w: (warning)
Paramètre l'avertissement de certaines configurations étonnantes. N'agit pas sur les messages qui résultent de l'option uW, quand un prédicat appelé n'est pas défini.

0 aucun message n'est donné.

1 prévient en cours d'exécution de l'effacement de :
- *block_exit/1* ou *block_exit/2* avec une variable libre à la place du numéro d'erreur,
- *load/2* ou *reload/2* avec une demande de substitution d'un préfixe inexistant,
- *save/2* ou *kill_module/1* pour un module inexistant,
- *suppress/1* ou *suppress/2* pour une règle inexistante,
- *kill_array/1* d'un tableau non défini.

2 donne les mêmes avertissements qu'en mode 1 et en supplément signale :
- pendant la compilation, les variables non muettes qui n'ont qu'une seule occurence dans la règle (susceptibles de résulter d'une faute de frappe).

z: (lectures des tailles de Prolog)
La valeur détermine la manière de définir les tailles de l'espace de Prolog:

U lecture des tailles des espaces dans le fichier binaire initial.

D tailles par défaut définies au chapitre 3.

-h entier

(heap) Définit la taille en Ko réservée pour la pile de copie.

-i file

(input) Définit le nom du fichier sur lequel est dirigée l'unité courante de lecture dès le démarrage. Par défaut la lecture se fait à la console.

-j file

(journal) Définit le nom du fichier journal créé par la primitive *paper* de Prolog (défaut: "*prolog.log*").

-m file

(module) Charge au démarrage de Prolog après chargement de l'état binaire le module objet de nom *file*. Si le fichier n'est pas trouvé dans le répertoire courant, il est cherché dans le répertoire défini par la variable d'environnement *PrologDir2*. Si plusieurs de ces options (maximum 20) sont utilisées, elles sont traitées dans l'ordre d'apparition.

-M entier

Temps de réponse maximal accepté, en millisecondes, pour un échange DDE, avant de signaler une erreur.

-o file

(output) Définit le nom du fichier sur lequel est dirigée l'unité de sortie courante dès le démarrage. Par défaut l'écriture se fait dans la "console". La bannière de Prolog apparait toujours dans la console.

-P parameter

Permet de passer un paramètre sur la ligne de commande. Ce paramètre est récupérable depuis Prolog dans le tableau prédéfini de chaînes de caractères *tab_user_param*. Si une chaîne vide est rendue, c'est qu'il n'y a plus de paramètres à récupérer. Au maximum 20 paramètres peuvent être transmis sur la ligne de commande. Exemple:

```
prolog -P 1 -P foo
   > val(tab_user_param[1], V);
   {V = "1"}
   > val(tab_user_param[2], V);
   {V = "foo"}
   > val(tab_user_param[3], V);
   {V = ""}
```

-q entier

Définit la taille en Ko réservée pour le codage des arguments des appels directs de procédures C.

-Q

Permet le démarrage de Prolog à l'état iconique (Quiet).

-r file

Définit le nom du fichier d'erreurs utilisé par Prolog (défaut: "*err.txt*"). Pour avoir les messages d'erreur en français, choisir le fichier *fr_err.txt*.

-R entier

(Realloc) Définit le pourcentage d'augmentation de taille pour la réallocation automatique (défaut: 25%).

-s entier

(stack) Définit la taille en Ko réservée pour la pile de récursion et des variables locales.

-t entier

(trail) Définit la taille en Ko réservée pour la pile de restauration (variables à restaurer après backtracking).

2.4. Création et exécution d'un programme

Voici un exemple complet de création et d'exécution d'un programme définissant une petite base de données, démarrant avec l'état de base. $ représente le caractère d'invite du système, > celui de Prolog. Par souci de concision, la règle *insert* a été utilisée directement, sans appeler l'éditeur.

```
$ win prolog (sous DOS)
PROLOG II+ ...
...                PrologIA
```

Prolog est lancé avec l'état initial, les seules règles (ou programmes) présentes en mémoire sont les règles prédéfinies.

```
>insert;
pere(Jean,Marie) ->;
pere(Jean,Pierre) ->;
;
```

Ici, le programme utilisateur est compilé, il est maintenant connu par Prolog.

```
{}
>pere(x,y);
```

Une exécution de ce programme est lancée.

```
{x=Jean, y=Marie}
{x=Jean, y=Pierre}
>exit;
```

On sauve un nouvel état binaire de démarrage contenant le programme utilisateur.

```
Bye......
$
```

Le fichier *prolog.po* a maintenant été créé, contenant un nouvel état (règles prédéfinies et votre programme). Pour repartir avec l'état précédent, relancer Prolog en donnant le nom de fichier correspondant:

```
$ win prolog prolog.po
PROLOG II+ ...
...                PrologIA
```

Prolog connait maintenant les règles prédéfinies et votre programme.

```
>pere(Jean,y);
{y=Marie}
{y=Pierre}
```

Vous pouvez lancer une exécution.

```
>quit;
Bye......
$
```

On sort sans sauver.

Pour sauvegarder l'état sous un autre nom de fichier, utilisez la commande *exit("nom-de-fichier")*.

2.5. Interruption d'un programme

A tout instant, un programme Prolog peut être interrompu par la frappe de la commande Ctrl+Pause. Cette interruption est traitée par le système Prolog de gestion des erreurs et correspond à l'erreur 16. Ainsi, elle peut être récupérée par un programme au moyen de la règle prédéfinie *block* (voir le manuel de référence). Si ce n'est pas le cas, l'erreur remonte jusqu'au niveau de la ligne de commande Prolog, et le message: INTERRUPTION UTILISATEUR est affiché.

2.6. Construction et lancement de Prolog avec graphisme

Pour la première exécution, exécuter la commande de lancement de Prolog avec le module graphique:

```
C:\> win prolog -m graphic.mo
```

Après avoir effectué les choix proposés dans le panneau de configuration, il est conseillé de sauver l'état courant afin de ne plus avoir à refaire ces choix:

```
> save_state("monini.po");
```

Pour les exécutions suivantes, il suffira alors de lancer la commande:

```
C:\> win prolog monini.po
```

2.7. Compilation et édition de liens

Pour une extension de Prolog, il est possible d'utiliser le fichier *prouser.c* ou bien créer un ou plusieurs autres modules. Il faut ensuite recompiler le(s) module(s) et faire l'édition de liens entre celui(ceux)-ci et la bibliothèque Prolog pour construire le fichier *prolog.exe.* Ceci est réalisé par le programme *prolink.bat*.

prolink.bat

Le programme *prolink.bat* permet de recréer un exécutable de Prolog modifié ou augmenté d'un certain nombre de modules pouvant contenir des descripteurs de données externes. La commande *prolink* est de la forme :

```
prolink [liste_modules_objets] : [liste_descripteurs]
```

liste_modules_objets

est une suite de noms (séparés par des blancs) de modules objets compilés auparavant, devant être inclus dans l'exécutable final.

liste_descripteurs

est une suite de noms (séparés par des blancs) de tables de descripteurs (voir le chapitre 7. du manuel de référence) se trouvant dans les modules objets indiqués en premier argument. Si l'on n'a pas créé de table de descripteurs, le deuxième argument peut être supprimé.

Attention: ne pas oublier le caractère blanc de part et d'autre du caractère séparateur ':'.

3. Spécificités de Prolog II+ Windows

3.1. Valeurs extrêmes des constantes arithmétiques
3.2. Les caractères dans la syntaxe Prolog II+
3.3. Personnalisation d'une Application
3.4. Modification du Programme principal
3.5. Extensions en mode 16 bits
3.6. Utilisation de PrologII+ sous forme de DLL

3.1. Valeurs extrêmes des constantes arithmétiques

Les booléens sont représentés par les entiers 0 et 1.

Les valeurs des entiers manipulés dans Prolog ne sont pas limitées à priori. Les valeurs des entiers qui peuvent être communiqués avec un langage externe sont comprises entre -2 147 483 648 et 2 147 483 647 (2^{31}-1) : ceci correspond au type *long int* de C.

Les réels manipulés dans Prolog correspondent au type *double* de C (IEEE 64 bits). Les valeurs des doubles sont comprises entre -1.79e308 et +1.79e308 et la plus grande valeur négative et la plus petite valeur positive sont respectivement -2.2e-308 et +2.2e-308. Certaines fonctions de communication acceptent uniquement des réels simple précision. Les valeurs des réels simple précision sont comprises entre -3.4e38 et +3.4e38 et la plus grande valeur négative et la plus petite valeur positive sont respectivement -1.2e-38 et +1.2e-38. Ces réels correspondent au type *float* de C (IEEE 32 bits).

3.2. Les caractères dans la syntaxe Prolog II+

Tout au long du premier chapitre du manuel de référence la syntaxe était plus spécialement décrite pour le jeu ISO, ce chapitre décrit une partie de la syntaxe relative aux caractères, adaptée au système Windows. Elle est valide pour chaque session Prolog qui utilise le jeu de caractères de la machine hôte.

Il faut noter que si MS-DOS utilise le jeu de caractères OEM, Windows utilise le jeu ANSI qui correspond pratiquement au jeu ISO. Il est donc possible que des fichiers écrits sous MS-DOS et contenant des caractères accentués affichent des rectangles noirs aux emplacements correspondants lorsqu'ils apparaissent dans une fenêtre Windows. Il est conseillé de créer les fichiers de texte avec un éditeur sous Windows.

```
big_letter =            "A" | … | "Z" ;
letter =                big_letter | "a" | … | "z"
                        | "À" … "ß" - "×"
                        | "à" … "ÿ" - "÷";

digit =                 "0" | … | "9" ;
alpha =                 letter | digit | "_";

separator =             "(" | ")" | "[" | "]" | "{" | "}"
                        | "|" | "," ;
separatorP=             ";"  |  "."  |  "<"  |  ">"  ;

special_char =          "%" | "'" | "`" | "_" | "!" | "`" |;
special_charE =         ";" ;

graphic_c =             graphic_char | "\",  graphic_char ;
graphic_charE =         "."   |  "<"  |  ">"   ;
graphic_char =          "#" | "$" | "&" | "*" | "+" | "-" | ":"
                        | "/" | "=" | "?" | "\" | "@" | "^" |  "~"
                        | NBSP … ¿ | × | ÷ ;

character =             letter | digit | separator |
                        graphic_char | special_char ;

string_char =           character - ( "'" | "\")  | "'"'"
                        | "\", format_escape ;

format_escape =         "b" | "f" | "n" | "r" | "t" | "\"
                        | newline
                        | octal_digit, octal_digit, octal_digit
                        | ("x" |"X"), hex_digit, hex_digit ;
```

3.3. Personnalisation d'une Application

Lors de l'initialisation de l'Application, la DLL *customiz.dll* est chargée si elle existe dans le répertoire qui contient l'exécutable de Prolog. Cette DLL ne contient pas de code exécutable, son rôle est de servir de support pour les ressources utilisateur. En effet, l'exécutable de Prolog contient déjà les ressources de Prolog. Les ressources utilisateur seront donc compilées et liées à ce module en utilisant un fichier *makefile* fourni.

Les ressources suivantes sont recherchées lors de l'initialisation et si elles existent, remplacent les ressources originales correspondantes:

10

est une ressource de type Icône, elle sera utilisée à la place de l'icône "Colonne" de Prolog pour représenter la fenêtre principale de l'Application. Elle peut également être utilisée pour représenter l'Application dans la fenêtre du Gestionnaire d'Applications: il faut pour cela la rechercher dans le module *customiz.dll* (au lieu de *prolog.exe*).

1, dans une ressource STRINGTABLE

est une ressource de type chaîne de caractères, qui sera utilisée comme titre initial de la fenêtre principale de Prolog, à la place de "Prolog II+". Attention, ne pas oublier que dans une ressource, le caractère '\0' final doit être explicite.

100

est une ressource de type Description de Boîte de Dialogue, qui pourra être écrite à la main ou en utilisant l'Editeur de Dialogues du SDK. Si elle est trouvée, l'item "A propos de Prolog ..." du menu "Application" disparaît et l'item "A propos ..." est validé, commandant l'ouverture de cette "About Box". Son contenu peut être quelconque, mais un seul bouton sera reconnu, et devra porter l'identificateur 1 (IDOK).

D'autres ressources utilisateur peuvent être fournies, mais elles ne seront pas gérées automatiquement. Suivre la procédure standard du système. Se rappeler seulement deux points essentiels:

- Votre fonction de dialogue doit aussi être exportée par l'Application, donc il faut la rajouter dans le module *prolog.def*.

- Votre ressource sera localisée dans le module *customiz.dll* et non dans *prolog.exe*, donc il faut utiliser le handle hUserDLL, disponible dans *prouser.c*, à la place du handle d'instance, dans toutes les primitives manipulant vos ressources (DialogBox(), ...).

Dans le cas de Windows 3 et pour utiliser des ressources au moyen d'extensions 16 bits, il sera plus judicieux de lier ces ressources directement à la DLL d'extension.

Le module *userrsc.rc*, fourni dans le kit, est un exemple de script de ressources de personnalisation. Il a été compilé et inclus dans le fichier *customiz.dl_* fourni dans le kit Prolog. Cette DLL doit être renommée en *customiz.dll* pour avoir une démonstration de cet exemple.

La compilation d'une DLL de personnalisation avec le SDK 16 bits fait l'objet d'une procédure particulière décrite en commentaires dans le fichier *userrsc.rc*.

3.4. Modification du module principal

Le fichier *proentry.c* contient le source de la routine principale de Prolog en tant qu'application. Cette routine est largement commentée et effectue des appels aux fonctions:

- *InitializeProlog (hInstance, hPrevInstance, lpCmdLine, nCmdShow)* qui va faire les initialisations nécessaires au lancement de Prolog. Cette fonction retourne *0* si l'initialisation a réussi, un numéro d'erreur sinon. Ses paramètres sont ceux de la fonction *WinMain*. . Elle remplace la fonction *ProStart* mentionnée au chapitre 8 du manuel de référence.

- *StartPrologMainGoal*() qui va lancer le but principal de Prolog, par appel à la fonction *promain()*, du fichier *princip.c*, dont le source est également fourni. Cette fonction retourne *0* si l'appel a réussi, un numéro d'erreur sinon. L'appel à la fonction *StartPrologMainGoal* n'est pas obligatoire pour utiliser la machine Prolog, il peut être remplacé par l'appel d'une procédure utilisateur qui installe son propre but et ainsi utilise Prolog en tant que runtime. Cependant, si cette procédure constitue une tache longue (notion intuitive), elle doit être lancée de la manière décrite dans le fichier afin de ne pas paralyser la machine.

- *TerminateProlog*() qui accomplira les tâches nécessaires à la terminaison de Prolog. Elle remplace la fonction *ProFinal* mentionnée au chapitre 8 du manuel de référence.

Dans ce fichier est également fournie la fonction dont se sert Prolog pour distribuer chaque message Windows lorsque sa lecture est faite de manière interne par l'environnement de Prolog.

3.5. Extensions en mode 16 bits (Réservé Windows 3)

Dans l'environnement 16 bits certaines fonctionnalités ne sont pas accessibles. Cet environnement ne permet pas l'appel direct de fonctions C (via le prédicat *callC*), la création de zones partagées, l'utilisation de descripteurs et l'utilisation des fonctions *new_pattern* et *get_formats*.

L'ajout de procédures externes est limité à la méthode des parasites décrite en Annexe D.

Dans la méthode des liens par parasites, un module utilisateur sert de relais entre Prolog et les routines utilisateur écrites en C. Les numéros de parasites 20000 à 29999 sont réservés aux routines C liées à ce mode. Ce module peut être développé comme pour une DLL Windows 3 standard. Le temps de commutation et de transfert de contrôle est très rapide et tout se passe comme si l'extension était directement liée à l'application 32 bits.

Exemple complet : ajout d'une règle en mode 16 bits

Voici un exemple complet, décrivant l'ensemble des opérations à exécuter pour créer une nouvelle règle prédéfinie implantée par une routine externe en code 16 bits.

Supposons que vous programmiez en C et que vous vouliez ajouter la règle prédéfinie *roots(a,b,c,x,y)* qui calcule les deux racines réelles *x* et *y* de l'équation: $ax^2 + bx + c = 0$ si elles existent et qui provoque un *backtracking* sinon.

Etape par étape, la marche à suivre est la suivante:

0. Copier les fichiers *proext.h, libentry.obj, usermpty.c, callpro.h, callpro.def, callproS.lib, userdll.mak* dans votre répertoire courant. Renommer *usermpty.c* en *roots.c*. Adapter *userdll.mak* en affectant la macro *SourceName* à *roots.*

1. Ajouter dans le fichier *roots.c* le code suivant:

```
    #include <math.h>
puis
   real_roots(perr_nb)
      int *perr_nb;
      {
      float a, b, c, d, x1;
      if ( !get_real(1, &a, perr_nb)
            || !get_real(2, &b, perr_nb)
            || !get_real(3, &c, perr_nb))
            return;
      if ( a == 0.)
            *perr_nb = -1;                /*backtrack*/
      else if ( (d = b * b - 4 * a * c) < 0.)
            *perr_nb = -1;                /*backtrack*/
      else
            {
            x1 = (-b + sqrt(d)) / (2 * a);
            if ( ! put_real(4, x1, perr_nb) )
                  return;
            put_real(5, -b/a - x1, perr_nb);
            }
      }
```

2. Modifier la fonction *user_rule* du fichier *roots.c* en ajoutant l'appel du nouveau programme externe. Par exemple, en donnant le numéro 20001 à la nouvelle règle:

```
user_rule(nb, err, err_nb)
   int nb, *err, *err_nb;
   {
   *err = *err_nb = 0;

    switch (nb)
         {
         ...
         case 20001:
               real_roots(err_nb);
               break;
         ...
         }
   }
```

3. Effectuer les compilations et l'édition de liens :

```
$ nmake -f userdll.mak
```

4. Renommer le fichier *roots.dll* en *prolo-.dll* et le copier au même endroit que l'exécutable prolog.

```
$ copy roots.dll  c:\prolog\prolo-.dll
```

5. Lancer une session Prolog:
```
$ win prolog
PROLOG II+, ...
...                  PrologIA
>
```

6. Insérer l'appel de la règle externe dans le module "":
```
> insert;
roots(a,b,c,x,y) -> /?20001;
;
{}
>
```

7. Essayer la nouvelle règle:
```
> roots(1.0e,-8.0e,15.0e,x,y);
{x=5.0e0, y=3.0e0}
> roots(+2.0e,+4.0e,+8.0e,x,y);
>
```

3.6. Utilisation de PrologII+ sous forme de DLL

Prolog II+ est également disponible sous la forme d'une DLL 32 bits (fichier *prolog.dll*) dénuée de tout environnement: graphisme, entrées/sorties. Cette DLL peut être utilisée par toute application Windows 32 bits, laquelle pourra aussi fonctionner sous Windows 3 étendu de l'API Win32s. La DLL Prolog exporte les routines décrites ci-après, ainsi que les routines classiques décrites au chapitre traitant des extensions en C à Prolog.

On retiendra seulement que toutes ces routines sont exportées par la DLL selon le protocole *cdecl*. L'utilisation de la librairie d'import (fichier *prodll.lib*) et de la définition des prototypes (fichier *exportsw.h*) évitera tout problème à ce niveau.

*ConnectDescriptors(EXTERNAL_DESCRIPTOR * paD[])*

Cette routine permet de déclarer dans le code de l'application le tableau de descripteurs *paD*, écrit comme on l'aurait fait pour une extension C directe. Ce tableau de descripteurs doit être persistant durant toute la session Prolog qui l'utilise (par exemple, qualifié de *static* s'il est en position de variable locale), et la déclaration doit être faite **avant** le début de la session (avant l'appel de *ProStart()*).
Si l'argument est NULL, les descripteurs seront supprimés dans la prochaine session. Si la routine est invoquée plusieurs fois, c'est le dernier tableau qui est pris en compte. La routine retourne 0 en cas de succès, -1 en cas d'échec (session déjà lancée).
L'exemple fourni avec la DLL utilise une fonction déclarée ainsi.

*ConnectUserRule(UserRuleFunction * pfUR)*

Cette routine permet de déclarer une fonction d'aiguillage appartenant à l'utilisateur *pfUR()* pour les extensions C utilisant la méthode des parasites. Cette fonction jouera le rôle de la fonction *user_rule()* du module *prouser.c*.
Le format de cette fonction (type *UserRuleFunction* défini dans le fichier des prototypes) est imposé: c'est le même que celui de la fonction *user_rule()*.
Si l'argument est NULL, la fonction d'aiguillage couramment installée est supprimée. Par sécurité, il est requis de supprimer la fonction courante avant d'en installer une nouvelle. La routine retourne 0 en cas de succès, -1 en cas d'échec (remplacement refusé).
La gamme de numéros de parasites qui peuvent être installés ainsi est celle qui correspondait aux extensions ordinaires: 1 à 9999.
Les mécanismes relatifs aux autres gammes de numéros restent inchangés: on peut par exemple appeler une fonction externe écrite en mode 16 bits de la même façon qu'avec la version standard de développement (DLL 16 bits).

*ConnectInString(InStringFunction * pfIS)*

Cette routine permet de compléter la DLL en y ajoutant les entrées, par la déclaration d'une fonction d'entrée de texte *pfIS()* à laquelle Prolog soumettra toutes les entrées (prédicats): avant l'installation d'une telle fonction, une chaîne vide est rendue pour chaque demande d'entrée (rendant donc celle-ci impossible). Le remplacement et la suppression de cette fonction fonctionnent comme la précédente.
Le format de cette fonction (type *InStringFunction* défini dans le fichier des prototypes) est imposé: son premier argument est l'adresse d'un buffer prêt à recevoir le texte (donc alloué), son second argument est la capacité maximale de ce buffer. Le code de retour de la fonction est ignoré: en cas d'erreur, elle doit rendre une chaîne vide dans le buffer.
Intérieurement, la fonction peut effectuer toute opération même bloquante (en traitant les événements) nécessaire pour obtenir le texte à retourner.

*ConnectOutString(OutStringFunction * pfOS)*

Cette routine permet de compléter la DLL en y ajoutant les sorties, par la déclaration d'une fonction de sortie de texte *pfOS()* à laquelle Prolog soumettra toutes les sorties (prédicats, messages): avant l'installation d'une telle fonction, aucune sortie ne peut être visible. Le remplacement et la suppression de cette fonction fonctionnent comme la précédente.
Le format de cette fonction (type *OutStringFunction* défini dans le fichier des prototypes) est imposé: son unique argument est l'adresse d'un buffer contenant le texte à imprimer. Le code de retour de la fonction est ignoré: aucune erreur n'est attendue.
Intérieurement, la fonction peut effectuer toute opération même bloquante (en traitant les événements) nécessaire pour afficher le texte. Toutefois, on remarquera que Prolog peut émettre des lignes vides, et donc un filtrage peut être nécessaire si par exemple des boîtes de messages sont utilisées.

InterruptProlog( )

Cette routine sans argument permet de déclencher une interruption utilisateur dans la session Prolog en cours d'exécution. Elle peut être invoquée à tout moment. L'exemple fourni avec la DLL en montre l'utilisation pour stopper l'exécution du prédicat Prolog *enum*.

4. Communication avec une application, en utilisant le protocole DDE

4.1. Aperçu du protocole DDE
4.2. Utilisation simple de Prolog II+ comme serveur DDE
4.3. Programmer en Prolog II+ un applicatif Client ou Serveur

L'environnement de Prolog II+ supporte le protocole DDE, d'une manière portable entre Windows 32 et OS/2-PM. Ces nouvelles fonctionnalités offrent deux types de possibilités:

- Lancer des buts Prolog et en récupérer les résultats depuis un applicatif Client externe, et ce, sans écrire aucun code d'interface,

- En chargeant un module Prolog spécifique, disposer d'un jeu complet de primitives permettant l'écriture en Prolog, sans aucune extension C, de tout applicatif Client ou Serveur DDE.

4.1. Aperçu du protocole DDE

Ceci n'est qu'une présentation très rapide du protocole DDE. L'utilisateur est invité à se reporter à la documentation du système hôte choisi pour approfondissement.

Le DDE est un protocole de communication unidirectionnelle entre deux applicatifs nommés **Client** et **Serveur**, à l'initiative du Client, de manière interne à la plateforme.

4.1.1. Les trois niveaux d'identification du protocole DDE

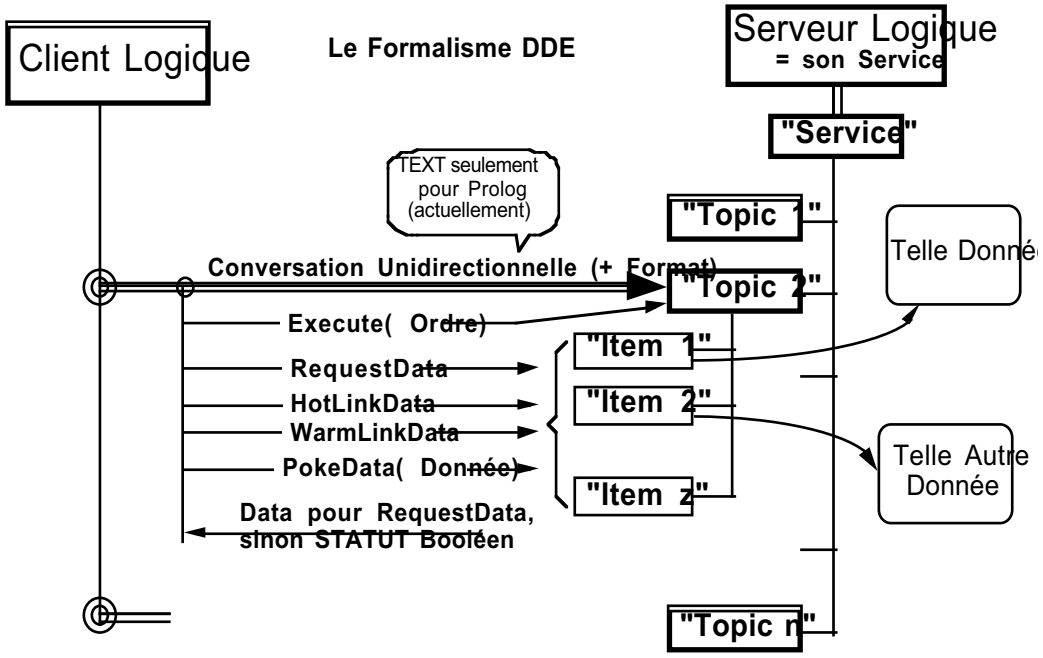
Le protocole DDE est fondé sur une identification à trois niveaux d'échanges, couramment nommés "Service", "Topic" et "Item". Chaque identifiant est une chaîne de texte, en général sensible aux minuscules/majuscules, limitée à environ 255 caractères, et à priori unique dans le contexte fixé par l'identifiant du niveau précédent.

Le **Service** identifie un Serveur logique. En pratique, c'est le plus souvent le nom générique de l'applicatif Serveur lui-même ("EXCEL", "WORD", "CLOCK", ...). Mais rien n'empêche un Serveur de déclarer plusieurs Services, s'il est capable de les gérer.

Le **Topic** indique un thème de conversation. C'est donc la base fondamentale de l'ouverture d'un échange DDE. En pratique, le Topic relève le plus souvent d'un Serveur précis, et donc il existe entre Topic et Service un lien implicite (par exemple: qui sait éditer la "Feuille de Calcul Excel nommée EXAMPLE.XLS" ?). Mais rien n'empêche un Serveur quelconque qui en a la compétence d'accepter un Topic donné.

Le nom d'**Item** indique un objet "donnée" précis relatif à une conversation ouverte sur un thème donné. En pratique, il s'agit donc souvent d'un "paragraphe" précis du "document" ouvert (par exemple: la plage "E7-K9" de la feuille de calcul précitée). Mais ce peut être toute entité conceptuellement subordonnée au Topic, compatible avec les opérations DDE à effectuer sur elle (par exemple: un signal "Reset" accédé forcément pour une opération "Poke", dans une conversation de Topic "Time" ouverte avec "CLOCK").

Indépendamment de ces identifications, un échange DDE se caractérise par le choix d'un **format de donnée**. Celui-ci, public ou propriétaire, est négocié implicitement lors de l'ouverture de la conversation: le Client est supposé s'être enquis des formats supportés par le Serveur pour le Topic visé, et avoir choisi le plus riche que lui-même supporte.



4.1.2. Les possibilités du protocole DDE Prolog

Afin de respecter le modèle de programmation standard de Prolog II+, le protocole DDE général a été adapté suivant les possibilités suivantes:

- Tous les échanges se traduisent par des appels de primitives ou bien l'appel automatique de buts par le système. Ces buts seront dans toute la suite appelés "buts-callbacks". Le retour d'une primitive correspond à la fin d'une transaction et fournit le statut associé.

- Les paramètres des communications lui sont présentés sous une forme classique en Prolog II+: les chaînes de texte elles-mêmes pour les chaînes d'identification (Service,Topic,Item), et un entier opaque pour les identificateurs de conversation. Une base de données interne assure la conversion et le contrôle de cohérence des paramètres fournis par l'utilisateur.

- Les capacités du DDE Prolog sont limitées à des données de type "Chaîne de Texte Standard".

- Le cas où ni le Service ni le TOPIC n'est spécifié par le Client potentiel suppose une phase de négociation où le Serveur décrit ce qu'il propose. Ce sous-protocole n'est pas implémentable sous forme de squelette extensible en Prolog. Donc il a été décidé que **seules les "Wild Connections" avec Topic spécifié seraient acceptées par Prolog**.

- Les buts-callbacks intervenant dans des mécanismes systèmes, avec les problèmes classiques que cela suppose, ne peuvent pas être mis au point "sur site" mais devront l'être avant raccordement, dans un contexte de simulation. De même, ils doivent fournir des traitements de durée "raisonnable", cette notion restant intuitive.

- La notification de la transmission d'une donnée se fait sur la base du Topic. La distribution à chaque client concerné par ce Topic est assurée de manière interne.

4.2. Utilisation simple de Prolog II+ comme serveur DDE

Pour utiliser Prolog II+ comme un Serveur DDE, il est nécessaire de disposer d'un applicatif Client capable de formuler des requêtes paramétrables au format "Texte Pur". A titre de test, il existe plusieurs possibilités:

- disposer déjà d'un tel applicatif, même surdimensionné pour cet usage,

- utiliser un "Sample DDE Client" fourni avec le Software Development Kit pour la plate-forme hôte (exemple: le SDK 32 bits de MicroSoft pour Windows),

- utiliser une autre instance de Prolog II+ programmée en Client DDE.

4.2.1. Initialisation du serveur Prolog II+

Afin de valider le Service DDE, la primitive suivante doit être invoquée:

ddePublishExternalEvent(_bActivate)

Active *(_bActivate* vaut 1) ou désactive (*_bActivate* vaut 0) le Service.
La désactivation du Service interdit toute future connexion, mais ne congédie pas les Clients éventuellement déjà connectés. Cependant, ils le seront de manière correcte si Prolog II+ se termine prématurément.

Après validation, Prolog II+ publie un Service (ou nom d'application) intitulé **Prolog2** et sous lequel il reconnaît le Topic **ExternalEvent**. Une "Wild Connection" sans Service mais spécifiant ce Topic sera aussi acceptée. Un nombre quelconque de Clients peuvent se connecter (chacun suivant sa méthode), jusqu'à limitation par le système.

Il est à remarquer que lors de la connexion et de toute opération ultérieure, la console de Prolog II+ ne manifeste aucune réaction, sauf pour une opération d'écriture explicite.

Après connexion, le Client peut utiliser librement les requêtes DDE standards avec la sémantique qui va être décrite ci-après.

4.2.2. Transmission d'une requête à Prolog II+

Il existe deux possibilités:

- Envoi d'une requête de type *Poke* sur un Item nommé *"texte",* sans donnée additionnelle (ignorée si présente): un événement de type *extern_event("texte", 0, 0)* est placé dans la queue d'événements de Prolog II+, et pourra être lu par le prédicat *get_event/3* du module graphique.

- Envoi d'une requête de type *Execute* (le paramètre Item n'existe pas), avec une donnée *but_prolog*: le but transmis est placé dans la queue d'exécution de Prolog II+, et sera lancé par la machine de manière asynchrone. **Le succès de la requête traduit celui de la transmission, et non celui du but transmis**. On ne considère pas ici la récupération des solutions.

Les messages d'erreur éventuels de Prolog se manifestent dans l'unité de sortie courante, par défaut la console, et ne sont pas transmis au Client. L'unification des variables du but transmis ne produit aucune trace. Seules les écritures explicites seront visibles.

Limitation: La donnée *but_prolog* est limitée à 32767 caractères et doit avoir la forme d'un but unique (et non pas une suite ou liste de buts) respectant la syntaxe en cours, sinon, une erreur est générée.

Exemples: réponses de la console de Prolog II+ à quelques requêtes:

- Requête *Poke* sur l'Item *"Lily was here"* avec la donnée *"You won't see me"*:

```
> get_event( 0, x, y);
{ x=extern_event( "Lily was here", 0, 0), y=0 }
>
```

- Requête *Execute* avec donnée *"please("anser me")"*:

```
-> please/1 : APPEL A UNE REGLE NON DEFINIE
>
```

- Requête *Execute* avec donnée *"enum(i, 15)"*:

```
> (il ne se passe rien de visible)
```

- Requête *Execute* avec donnée *"outml("Hello)"*:

```
Hello
> (mais pas d'affichage de {} )
```

4.2.3. Exécution d'un but Prolog avec récupération des solutions

La donnée transmise par la requête *Execute* accepte une seconde syntaxe dont la description suit:

but_prolog, Index, Identifiant où:

but_prolog obéit aux contraintes précédemment exposées,
Index est le numéro basé en 1 de l'argument de *but_prolog* à récupérer,
Identifiant a la syntaxe d'un identificateur abrégé Prolog (sans préfixe) et désignera la solution.

Lors de l'exécution du but, la valeur assignée à l'argument désigné sera convertie en une chaîne de caractères, et associée à son identifiant. Si le but échoue ou produit une erreur, la valeur assignée à la solution sera respectivement *"*Failure"* ou *"*Error nnn"*, *nnn* étant le code d'erreur Prolog. Le Client peut récupérer la ou les solutions de deux manières:

- Par l'envoi d'une requête *Request* sur l'Item *Identifiant* lancée **après** l'exécution du but: la valeur rendue est celle de la dernière solution courante du but auquel est associé l'identifiant, dans les conditions décrites précédemment. L'inconvénient est qu'il n'existe aucun moyen de synchronisation automatique permettant de savoir si le but a été exécuté. Si ce n'est pas le cas, la requête échoue, sauf si l'identifiant utilisé contient déjà la solution d'un autre but lancé précédemment. Le second inconvénient est que l'on récupère seulement la dernière solution.

- Par l'envoi d'une requête *HotLink* ou *Advise* sur l'Item *Identifiant*, lancée **avant** exécution du but: lors de l'exécution du but auquel est associé l'identifiant, chaque solution (au sens précédent) est transmise au Client en tant que mise à jour de l'Item. L'avantage de cette méthode est que la synchronisation est semi-implicite: elle convient très bien à un Client à structure événementielle. Il est à noter que si le Client veut différencier la dernière solution des autres, le but appelé devra être encapsulé afin de transmettre une solution différentiable. Après avoir eu satisfaction, le Client devra annuler le lien (Requête *Unadvise*).

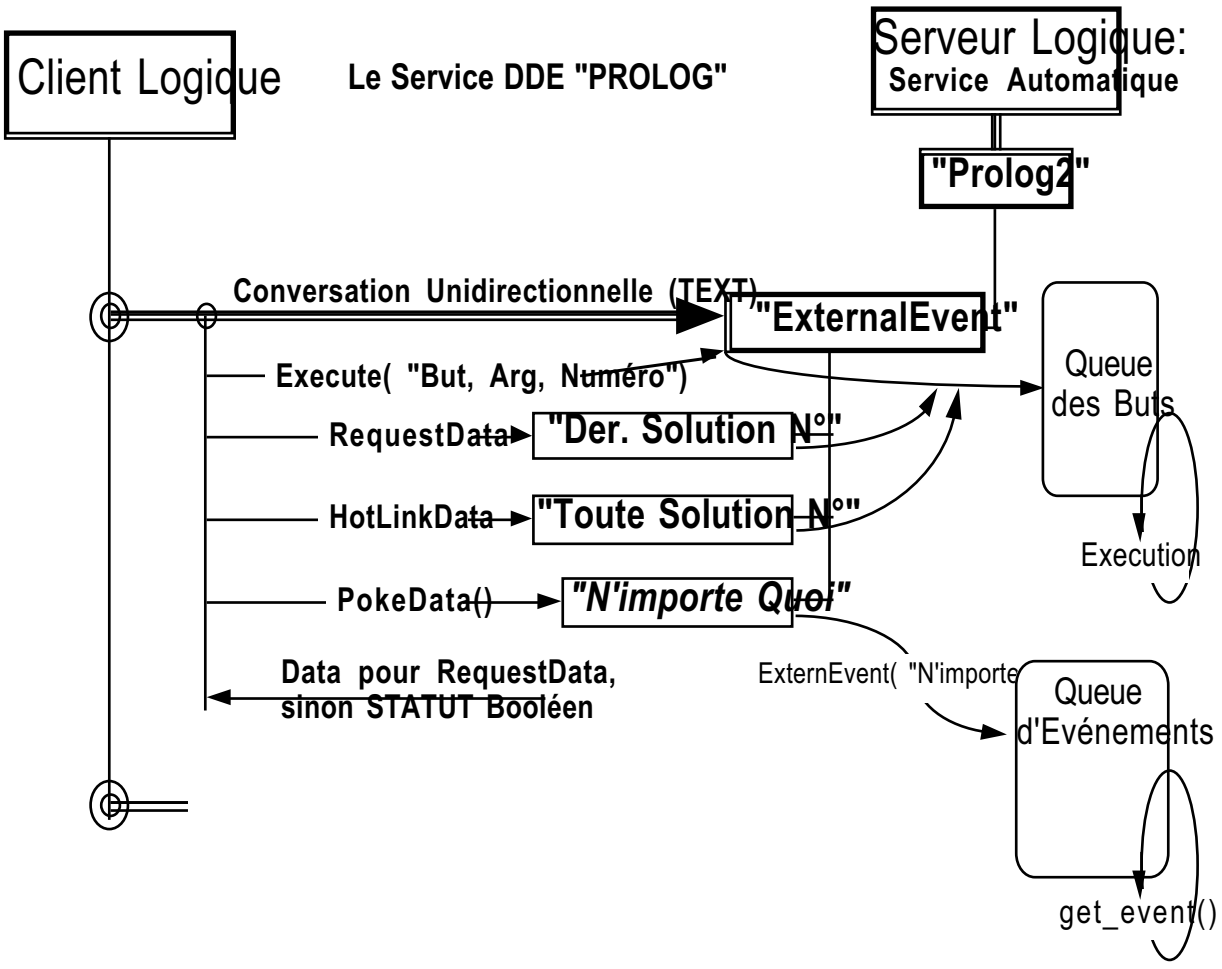
Remarques:

- Si le Serveur Prolog II+ est sollicité par plusieurs Clients, les buts transmis sont empilés par ordre chronologique, sans autre lien avec leur auteur que l'identifiant spécifié. Cette caractéristique permet de partager des données entre Clients, avec ou sans leur consentement. Dans le cadre d'une application reposant sur plusieurs applicatifs, il convient de bien choisir les identifiants utilisés.

- Dans le cas où l'un des Clients transmet le but *quit*, ou tout autre ayant un effet de bord quelconque, Prolog II+ l'exécute sans considération pour les intérêts des autres Clients. Exemple**:** un outil tentant périodiquement une connexion sur Prolog II+ pour y lancer une requête *Execute(quit)* constituerait un vrai vaccin anti-Prolog.

4.2.4. Schéma du Serveur Prolog II+

Ce schéma est à rapprocher de celui plus général donné précédemment:



4.2.5. Exemples

- La requête *Execute* avec la donnée *'eq(production, x), 2, my_var'* réussit et associe dans l'interface la valeur de *x* (ici *production*) à l'identifiant *my_var*.

- La requête *Execute* avec la donnée *'eq(production, x), 1, my_atom'* réussit.

- La requête *Request* pour l'item *my_var* réussit et rend ***production***

- La requête *Request* pour l'item *my_atom'* réussit et rend ***production*** (1er argument).

- La requête *Execute* avec la donnée *'arg2(X, Y, Z), 3, my_arg'* réussit, mais le message:

```
-> <v33> : ARGUMENT DE MAUVAIS TYPE
```

est affiché dans la console de Prolog II+

- La requête *Request* pour l'item *my_arg* réussit et rend ****Error 253***

- La requête *Advise* (ou *HotLinkData*) pour l'item *enumerate* réussit.

- La requête *Execute* avec donnée *'enum(i, 5), 1, enumerate* ' réussit, et le Client reçoit successivement: ***1***, ***2***, ***3***, ***4***, ***5***.

- La requête *Request* pour l'item *enumerate* réussit et rend ***5***.

4.3. Programmer en Prolog II+ un applicatif Client ou Serveur

Il est nécessaire dans les deux cas de charger le module *"dde.mo"* qui installe les primitives DDE. Sa désinstallation se fait par l'appel à la primitive:

ddeCleanup/0

> Termine toutes les conversations et Services en cours et élimine les primitives de Prolog relatives au DDE. Cette primitive n'est pas le bon moyen pour un applicatif Client ou Serveur de terminer son activité DDE. Elle ne devrait être utilisée qu'ensuite.

Selon les performances du système et des applicatifs, des problèmes de time-out peuvent survenir, produisant une erreur Prolog. Le délai général de time-out, bien que confortable (3 secondes) peut être ajusté par l'option *-M tttt* de la ligne de commande, où *tttt* est la durée en millisecondes. Cette valeur sera à déterminer par essais. Elle est talonnée par les valeurs 200 et 15000 ms.

Les paragraphes qui suivent étudient séparément les deux modes, mais rien n'interdit à un applicatif d'être simultanément Client et Serveur pour des conversations différentes. Il convient alors en l'écrivant d'accorder le plus grand soin à la gestion des numéros de conversations et éventuellement aux identificateurs DDE utilisés. Pour cela, la primitive suivante sera un auxilliaire précieux:

ddeRememberDatabase(_xData)

> Enumère les Services et les conversations en cours et les unifie avec l'argument *_xData*. Son format est le suivant:
> *dde_I_serve(_tService)* pour les Services (seulement pour le Serveur donc),
> *dde_I_am_Client(_hConversation, _tTopic)* ou
> *dde_I_am_Server(_hConversation, _tTopic,)* (selon le cas) pour les conversations.

4.3.1. Programmer un applicatif Client DDE

Les transactions DDE sont à l'initiative du Client, mais certaines prennent aussi la forme de notifications asynchrones de la part du Serveur. Le Client est supposé en tenir compte pour adapter sa conduite. De même, le Client est supposé connaître le mode d'emploi du Serveur qu'il prétend utiliser. Eventuellement, il y aura également lieu pour le Client de démarrer l'applicatif Serveur désiré.

Primitives de connexion et de déconnexion

ddeDeclareAsClient(_xService, _tTopic, _hConversation)

Tente d'ouvrir une conversation sur le Topic spécifié par la chaîne de caractères *_tTopic* avec le Serveur logique spécifié par la chaîne de caractères *_xService*, ou n'importe quel serveur si *_xService* vaut *nil*. Si la conversation est ouverte, l'argument *_hConversation* est unifié avec une valeur (entier opaque) unique identifiant cette conversation pour toute opération future. Si aucun serveur n'est trouvé, une erreur est signalée. Si plusieurs serveurs sont candidats, seule la première réponse sera retenue.

ddeEndConversation(_hConversation)

Termine la conversation désignée par *_hConversation* en congédiant le Serveur. Si *_hConversation* est une variable libre, termine par énumération (backtracking) toutes les conversations en cours en unifiant *_hConversation* avec la conversation concernée. Si la conversation désignée n'existe pas, l'unification échoue. L'usage de cette primitive est naturel pour le Client.

Primitive de transmission de requêtes

ddeTransmitRequest(_hConversation, _xRequest)

Effectue de manière synchrone la requête *_xRequest* auprès du Serveur lié à la conversation *_hConversation*. Le statut de terminaison de cette primitive (succès ou erreur) est lié à celui de la requête. Les requêtes reconnues sont les suivantes:

- *ddeRequestData*(*_tItem, _xValue)*: effectue une requête *Request* standard pour la donnée nommée par la chaîne *_tItem*, et tente d'unifier le résultat, une chaîne de texte Prolog, avec la valeur de *_xValue.*
- *ddeWarmLinkData*(*_tItem, ddeOpen)* ou
ddeWarmLinkData(_tItem, ddeClose): effectue une requête *Advise* standard pour la donnée nommée par la chaîne *_tItem*, demandant un abonnement de type notification de modification de cette donnée (lien tiède), ou résilie ce même abonnement.
- *ddeHotLinkData*(*_tItem, ddeOpen)* ou
ddeHotLinkData(_tItem, ddeClose): effectue une requête *Advise* standard pour la donnée nommée par la chaîne *_tItem*, demandant un abonnement de type mise à jour systématique de cette donnée (lien chaud), ou résilie ce même abonnement.

- *ddePokeData*(*_tItem, _xData)*: effectue une requête *Poke* standard pour l'Item nommé par la chaîne *_tItem*, adressant ainsi au Serveur la donnée *_xData*, qui est soit une chaîne de texte, soit *nil* (simple signal sans donnée jointe).
- *ddeExecute*(*_xArgument):* effectue une requête *Execute* standard au Serveur, lui passant une commande textuelle contenue dans la chaîne *_xArgument*.

Buts-callbacks

Les buts-callbacks (noms prédéfinis) sont dans le module global (ils ont le préfixe ""). Ils sont écrits par l'utilisateur et appelés par le système. Ils sont donc soumis à des contraintes relevant de la programmation système. On retiendra principalement les règles suivantes:

- Les buts-callbacks peuvent effectuer des retours arrière, mais la requète transmise sera validée (si toutefois la requête attend une validation) s'il existe au moins une solution.

- Les buts-callbacks ne doivent pas effectuer de traitements de taille déraisonnable (notion purement intuitive), ni à fortiori d'intèraction avec l'utilisateur incluant des phases modales. Il y a risque de time-out, voire même de réaction défensive du système.

- Ceci implique que les buts-callbacks ne peuvent pas être mis au point sur site. Il est nécessaire d'effectuer leur mise au point avec un simulateur.

:ddeCbServerClosing(_hConversation)

Signale la terminaison prématurée de la conversation *_hConversation* par le Serveur, quelle qu'en soit la raison. Il s'agit d'une notification à postériori, et le statut de terminaison de son traitement est sans effet sur le résultat transmis au serveur. La valeur de *_hConversation* doit être considérée comme désormais invalide et recyclable (donc *ddeEndConversation(_hConversation)* échouera si tenté).

:ddeCbServerUpdate(_hConversation, _tItem, _xArgument)

A la suite d'une requête *Advise*, notifie un changement dans la valeur de l'Item auquel le Client s'est abonné pour la conversation *_hConversation*. Cet Item est désigné par la chaîne de caractères *_tItem*. La nouvelle valeur est soit transmise dans *_xArgument* (lien chaud) soit disponible pour une future requête de type *ddeRequestData* (lien tiède). Dans ce dernier cas, *_xArgument* vaut *nil*. Le statut de terminaison de ce but n'a aucun sens ni aucun effet sur le résultat transmis au serveur.

Exemple

Le scénario suivant décrit, en Prolog, un dialogue avec un Serveur de type horloge.

```
> load( "dde.mo") ;
{}
> insert;   % Code minimum nécessaire pour suivre le dialogue
```

```
ddeCbServerUpdate( _hC, "Now", "24:00:00")     ->
   ddeTransmitRequest( _hC, ddeExecute("Close") )
   ! ;
ddeCbServerUpdate( _, "Now", _tTime)     ->
   outm( "\tIt is ") outml( _tTime) ;
ddeCbServerClosing( _)  ->
   outml( "\t\tEnd !") ;
;
{}
> ddeDeclareAsClient( nil, "Time", _hC);
{ _hC=1 }
> ddeTransmitRequest( 1, ddeRequestData("Now", _t));
{ _t="15:20:33" }
> ddeTransmitRequest( 1, ddePokeData("Now", "23:59:30"));
{}
> ddeTransmitRequest( 1, ddeHotLinkData("Now", ddeOpen));
{}
   It is 23:59:55
   It is 23:59:56
   It is 23:59:57
   It is 23:59:58
   It is 23:59:59
      End !
> ddeEndConversation( 1); % Failure, number 1 is out-of date
>
```

4.3.2. Programmer un applicatif Serveur DDE

Ce mode est plus contraignant, car un Serveur doit savoir être à la complète disposition de Clients quelconques, et aussi éviter tout conflit d'intérêts avec d'autres Serveurs. Donc il importe en premier lieu de définir formellement puis de doter de noms pertinents toutes les commandes qu'il pourra recevoir. Son mode d'emploi devra être très complet.

Primitive de déclaration

ddeDeclareAsServer(<_nMode, _tService>)

Déclare (si *_nMode* vaut *ddeOpen*) ou supprime (si *_nMode* vaut *ddeClose*) le Service DDE nommé par la chaîne de caractères *_tService*. Si *_tService* est une variable libre et *_nMode* vaut *ddeClose*, termine chaque Service déclaré (backtracking) en l'unifiant avec *_tService*. La suppression d'un Service renvoie brutalement tous les Clients qui s'y sont connectés. Ceci devrait être évité, en terminant au préalable toutes les conversations concernées.

Primitives de connexion et de déconnexion

ddeAcceptClient(_tService, _tTopic, _hConversation)

Accepte la requête du Client en cours de connexion, cette primitive devant être invoquée durant l'effacement du but-callback *ddeCbClientWantsServer/2*. Les arguments *_tService* et *_tTopic* sont des chaînes de caractères transmises par ce but-callback. Si aucun problème système ne survient, l'argument *_hConversation* sera unifié avec une valeur entière unique identifiant cette conversation pour toute opération future.

ddeEndConversation(_hConversation)

Termine la conversation *_hConversation* en congédiant le Client. Si *_hConversation* est une variable libre, termine par énumération toutes les conversations en cours en unifiant *_hConversation* avec la conversation concernée. L'usage de cette primitive est naturel pour le Client. Bien que possible, l'usage de cette primitive est à priori très déplacé pour le Serveur. Il ne devrait être utile qu'en cas de problème.

Primitive de transmission des données aux Clients:

ddeTransmitData(_tTopic, _tItem, _xNewValue)

- Doit être invoquée à la demande d'un Client, donc durant l'effacement du but-callback *ddeCbClientRequest/2* pour une requête *ddeRequestData*, pour transmettre *_xNewValue*, la nouvelle valeur de l'Item *_tItem*.
- Peut être invoquée à l'initiative du Serveur pour transmettre soit la nouvelle valeur *_xNewValue* de l'Item *_tItem* (lien chaud) soit une notification aux Clients ayant demandé un abonnement à l'Item modifié (lien tiède), auquel cas l'argument *_xNewValue* est ignoré. Le choix de notifier sur la base du Topic *_tTopic* et non à chaque Client concerné est imposé par des contraintes d'implémentation. La base de données interne assure la bonne distribution des notifications.

Buts-callbacks

Mêmes remarques que pour le mode Client.

:ddeCbClientWantsServer(_xService, _tTopic)

Notifie une demande de connexion d'un Client pour le Topic désigné par la chaîne de caractères *_tTopic* et au Serveur logique désigné par la chaîne de caractères *_xService*, sauf si *_xService* vaut *nil*, ce qui signifie "sans préférence". Pour que la conversation soit établie, le Serveur doit effacer ce but et durant cette opération effacer la primitive *ddeAcceptClient/3,* sans quoi (erreur ou échec) le Client est débouté. A priori, le Client n'est pas tenu d'insister, et donc un refus du Serveur peut être considéré comme définitif.

:ddeCbClientRequest(_hConversation, _xRequest)

Notifie la requête *_xRequest* du Client, dans les conditions liées à la conversation *_hConversation*, et attend la réponse de manière synchrone. Le statut de terminaison de ce but conditionne celui de la requête. Une erreur Prolog aura ici le même effet qu'un échec. Les requêtes reconnues sont les suivantes:

- *ddeRequestData*(*_xArgument):* notifie une requête *Request* standard pour la donnée nommée par la chaîne *_xArgument*, et attend du Serveur l'exécution de la primitive *ddeTransmitData/3* passant sa nouvelle valeur durant l'effacement du but,
- *ddeWarmLinkData*(*_tItem, ddeOpen)* ou

ddeWarmLinkData(*_tItem, ddeClose)*: notifie une requête *Advise* standard pour la donnée nommée par la chaîne *_tItem*, demandant un abonnement de type notification à cette donnée (lien tiède), ou la résiliation de ce même abonnement, information que le Serveur est supposé enregistrer pour adapter son comportement futur,

- *ddeHotLinkData*(*_tItem, ddeOpen)* ou
ddeHotLinkData(*_tItem, ddeClose)*: notifie une requête *Advise* standard pour la donnée nommée par la chaîne *_tItem*, demandant un abonnement de type mise à jour systématique à cette donnée (lien chaud), ou la résiliation de ce même abonnement, information que le Serveur est supposé enregistrer pour adapter son comportement futur,
- *ddePokeData*(*_tItem, _xValue)*: notifie une requête *Poke* standard pour l'Item nommé par la chaîne *_tItem*, adressant ainsi la donnée *_xValue* (chaîne de texte Prolog) au Serveur, ou bien encore un simple signal sans donnée jointe, *_xValue* valant alors *nil*,
- *ddeExecute*(*_xArgument):* notifie une requête *Execute* standard au Serveur, lui passant une commande textuelle préconvenue contenue dans la chaîne *_xArgument*, que le Serveur est supposé comprendre et exécuter de manière synchrone, et dont le statut doit conditionner le succès de ce but,
- *ddeClose*: notifie la déconnexion du Client. La base de donnée interne est mise à jour automatiquement, et le Serveur est supposé faire de même pour ses données propres éventuelles. Il s'agit d'une notification à postériori, et le statut de terminaison du but est sans effet.

Exemple

Imaginons un Serveur représentant le garde-manger d'un restaurant. Il doit gérer le stock de différents plats, consommés de manière aléatoire. Il doit pour chacun pouvoir donner la quantité disponible et retrancher la quantité consommée. Il doit alerter le patron en cas de rupture de stock et ensuite pouvoir lister sur commande les plats à renouveler. Enfin, il doit pouvoir enregistrer une livraison.

On définit le nom du Service: **Fridge**.

On définit les noms des Topics: **Boss**, réservé au maître unique des lieux,

Waiter, pour les serveurs de la salle.

Les Items seront les noms des plats, associés à une quantité disponible dans une base de données interne. Un Item nommé "Rupture" et accessible au seul patron lui permet de connaître la liste des plats dont le stock est inférieur à 5. Le patron peut l'interroger par *Request* et/ou s'y abonner. Il recevra alors la liste à jour à chaque consommation d'un serveur de la salle. Seul le patron est autorisé à alimenter le Frigo par une requête *Execute* passant une liste chiffrée. Il a aussi accès à toutes les opérations permises aux serveurs de la salle.

Le code de cet exemple est donné dans le fichier *ddefrigo.p2* du kit. On remarque qu'il est nettement plus volumineux que le précédent. Pour l'utiliser de manière amusante, il est préférable de disposer de plusieurs (environ 3) applicatifs Clients pouvant utiliser (facilement) des Identifiants DDE arbitraires. pour le cas où des instances de Prolog sont utilisées comme clients, un fichier nommé *ddecligo.p2*, disponible dans le kit fournit des commandes simplifiées adaptées à cet exemple. Dans tous les cas, lancer la règle *to_begin* pour démarrer ces programmes. Ils listent alors leur mode d'emploi.

Les manipulations suivantes sont suggérées:

- En premier, remplir le garde-manger, initialement vide: le patron lance par exemple une requête *Execute* de la chaîne: *"4 Lapin, 15 Poulet, 7 Gateau Basque, end"* ,

- Essayer de connecter un second patron, surtout si le premier est déjà abonné à Rupture,

- Essayer de réclamer un plat indisponible: Requête Poke de "nom du plat, -1" par exemple,

- Vérifier que le patron est prévenu lorsque les stocks diminuent,

- ...

5. Primitives Graphiques

5.1. Description du système graphique
5.2. Primitives de gestion des fenêtres
5.3. Primitives élémentaires de gestion des objets attachés à une fenêtre
5.4. Primitives spécifiques pour la gestion de menu
5.5. Mode de dessin et d'écriture
5.6. Dessin et positionnement
5.7. Position de la souris dans une zone graphique
5.8. Primitives spéciales de saisie
5.9. Règles pour gérer des objets structurés
5.10. Envoi d'événements à Prolog depuis un objet externe

Le système Prolog II+ comporte un dispositif asynchrone de gestion des événements permettant le fonctionnement de l'environnement (fenêtrage, menus, ...) sans que l'utilisateur ait à réorganiser son programme sous la forme d'une boucle de gestion des événements. La plupart des événements tels que le redimensionnement d'une fenêtre, le défilement des fenêtres de texte ..., sont gérés automatiquement. L'utilisateur peut associer des procédures Prolog à certains événements comme la sélection d'un menu, ou le rafraîchissement d'une fenêtre graphique… Les événements de type clic souris ou action clavier peuvent être testés par programme, ce qui permet de faire réagir le programme en conséquence.

L'ensemble des primitives graphiques de Prolog II+ offrent également la possibilité de créer des fenêtres, supports d'objets graphiques n'étant pas eux-mêmes des fenêtres. Les primitives de création et manipulation d'objets agissent uniquement sur des objets Prolog. Tous ces objets doivent être créés par les prédicats Prolog, et non par un outil externe (de type générateur d'écran, par exemple). Par contre, les primitives de gestion des événements peuvent aussi manipuler des événements survenus sur des objets externes à Prolog.

Les descriptions des différents environnements ont été regroupées dans un seul document pour mettre en évidence les différences qui existent encore entre ces environnements.

Le signe dans la marge indique une primitive ou une caractéristique spécifique au Macintosh.

Le signe s signale les nouvelles primitives disponibles sous Window3, sous OS/2-PM et sous Motif[1] . Elles ne sont pas encore implémentées sur Macintosh.

---

[1] La partie graphique PrologII+ sous l'environnement Motif a été développée en collaboration avec Digital Equipment Corporation.

Vous ne devez pas utiliser les règles marquées, donc spécifiques à une machine ou un environnement graphique, si vous désirez porter vos programmes sur d'autres configurations.

5.1. Description du système graphique

Ce chapitre décrit une bibliothèque de primitives pour la réalisation d'interfaces graphiques. Il suppose du lecteur une certaine pratique des environnements multi-fenêtres et de leur boîte à outils. Plus précisément, il suppose du lecteur la connaissance des caractéristiques de l'environnement graphique de la machine hôte. En effet, l'aspect général du graphique ou le comportement des fenêtres et des objets graphiques, ainsi que le maniement de la souris sont standard à l'environnement hôte; pour toute information sur ce fonctionnement, se référer au manuel correspondant.

Tous les effets graphiques obtenus ici doivent être considérés, du point de vue de Prolog, comme des effets de bord. En particulier, un objet dessiné n'est jamais effacé au backtracking. De même, la position du point d'écriture est une variable globale.

5.1.1. Evénements

Les événements sont les conséquences d'interruptions matérielles générées par le clavier ou la souris, traités à différents niveaux (système d'exploitation, système graphique…) et donc propagés jusqu'à différents niveaux (…utilisateur final). Nous nous intéressons à ceux qui arrivent jusqu'à Prolog.

Un certain nombre d'objets graphiques Prolog sont sensibles aux événements et les transmettent au programmeur Prolog, qui peut décider trois traitements différents:

- les traiter immédiatement en attachant un programme Prolog sur l'objet;
- les envoyer automatiquement dans la queue d'événements de Prolog pour les traiter ultérieurement;
- les ignorer.

5.1.2. Objets graphiques

Tout d'abord parlons des objets principaux que sont les **fenêtres**. Toutes les fenêtres peuvent ou non avoir un cadre, une barre de titre, des menus. Elles peuvent êtres redimensionnées, iconifiées, invisibles. Les fenêtres de texte peuvent avoir des barres de défilement horizontal ou vertical, et peuvent sauvegarder leur contenu.

Il existe quatre types de fenêtre, avec des caractéristiques et des fonctions différentes:

Les fenêtres EDIT:

ce sont des fenêtres de texte, destinées à l'édition de texte. Tout le contenu de la fenêtre est modifiable directement, au moyen du clavier, de la souris, des menus; par programme, via les primitives d'entrées sorties de Prolog, le point d'écriture se trouve à la fin du texte; le point de lecture est initialement au début de la fenêtre, ensuite après la dernière position lue. Noter que si une lecture est demandée, Prolog copie dans un tampon toute une ligne.
Ces fenêtres ont une capacité limitée, certaines manipulations ayant pour effet l'augmentation de la taille du texte au delà de la limite seront ignorées.

Les fenêtres FRAMEPANEL:

ce sont des fenêtres "vides" dont le seul rôle est d'accueillir des objets. Elles ne possèdent pas de partie dessinable, éditable…

Les fenêtres GRAPHICS:

ce sont des fenêtres destinées au dessin, sensibles aux actions clavier et souris, qui peuvent éventuellement recevoir des objets. Elles peuvent contenir du texte par les moyens classiques d'écriture. Par défaut, l'image bitmap n'est pas sauvegardée, ce qui veut dire que le rafraîchissement n'est pas assuré. Elles peuvent être dotées d'un bitmap qui permettrait leur retraçage automatique.
Il existe toujours, par compatibilité, le type *MODAL* qui est un sous ensemble du type GRAPHICS et qui désigne les fenêtres GRAPHICS qui ont l'attribut *MODAL*, attribut qui donne le focus à la fenêtre, restreint les événements utilisateur à cette seule fenêtre, et donc rend la saisie obligatoire dans cette fenêtre.
Par convention, les primitives qui n'ont d'effet que dans un objet GRAPHICS sont préfixées par *gr_*.

Les fenêtres TTY:

ce sont des fenêtres de texte qui fonctionnent en écriture et en lecture, toujours à la fin de la fenêtre. Elles ont un rôle de trace ou listing et fonctionnent à la manière des terminaux. C'est à dire, tout ce qui est affiché n'est pas modifiable, la saisie est en mode ligne et donc, tant que la ligne de saisie n'est pas terminée, elle est modifiable.
Ces fenêtres ont une capacité limitée, une troncature automatique se fait lorsque c'est nécessaire par le haut de la fenêtre.

Voyons maintenant les **autres objets**; ils sont conçus pour être placés sur une fenêtre qui est alors appelée : parent. Ils se déplacent avec leur parent et ne peuvent exister après sa destruction. Leur taille et leur position sont définies par des coordonnées relatives au coin intérieur supérieur gauche du parent.

Qu'advient-il si le parent change de dimension?

Pour déterminer le comportement d'un objet lors du redimensionnement de son parent, on définira **l'attachement de l'objet à un bord de sa fenêtre parent** :

Un objet est attaché à un bord de son parent signifie que l'écart (défini au moment de la création) entre ce bord et l'objet est constant quelles que soient les évolutions des dimensions du parent.

Il faut noter que :

- un objet est toujours attaché dans les deux directions (au minimum à un bord pour chaque direction). Par défaut, il s'agit du bord haut et du bord gauche.

- les objets dont la dimension n'est pas fixée par le système, peuvent être attachés aux deux bords pour la même direction. Dans ce cas, l'objet sera redimensionné en même temps que son parent.

Les objets qui doivent avoir un parent sont les suivant :

Check button:

bouton de choix à deux états, quand il est activé, il change d'état et génère un événement.

Push button:

bouton qui génère un événement quand il est activé.

Radio button:

bouton de choix exclusif; c'est un bouton à deux états qui fait partie d'un groupe de boutons dont un seul peut être sélectionné à la fois. En particulier, dès qu'un bouton du groupe est activé, il génère un événement, il prend la sélection et celui qui l'avait la perd. Par défaut, à la création, c'est le premier bouton du groupe qui est sélectionné.

Drawing area:

zone graphique destinée au dessin, sensible aux actions clavier et souris. Elle peut contenir du texte par les moyens classiques d'écriture. Par défaut, l'image bitmap n'est pas sauvegardée, mais la zone peut être dotée d'un bitmap pour son retraçage automatique. Les primitives préfixées par *gr_* agissent dans cette zone si elle a été précédemment définie comme la zone graphique courante (voir *gr_window*).

Editfield:

zone de texte éditable monoligne ou multiligne, génère un événement lors de la frappe d'un caractère ou de la perte ou du gain de focus.

Label :

zone de texte non éditable, non sensible aux événements, sert uniquement à l'affichage d'un texte dont le retraçage est géré automatiquement. Si le texte est vide, cela permet d'obtenir un rectangle.

Listbox:

liste d'items, inscrits dans un rectangle, avec barre de défilement verticale. Chaque item peut prendre deux valeurs : sélectionné ou non. A chaque sollicitation d'un item de la listbox, un événement est généré.

Popup menu:
Pulldown menu:

Un menu regroupe une suite d'items disposés verticalement. Un item peut être soit terminal, soit lui-même un menu, on parlera dans ce cas de menu hiérarchique. La sélection d'un item terminal du menu génère un événement et fait disparaître le menu. On distingue deux formes de menus qui se différencient par la façon de les activer et par les caractéristiques des fenêtres auxquelles ils peuvent être attachés.
Il est possible d'attacher un '*popup menu*' particulier à n'importe quelle unité fenêtre. L'activation du '*popup menu*' (également appelé *menu flottant*) d'une fenêtre se fait, alors que le curseur de la souris est dans la fenêtre, soit en appuyant sur le bouton menu de la souris si la souris comporte plusieurs boutons, soit en appuyant simultanément une touche d'option et le bouton de la souris si la souris possède un seul bouton. Le menu apparaît sous le curseur de la souris.
Il est possible d'attacher un ou plusieurs '*pulldown menu*' (également appelé *menu déroulant*) à n'importe quelle unité fenêtre possédant une barre de titre (ne convient pas aux fenêtres de forme cadre). Le titre du menu s'insère alors dans la barre de menu de la fenêtre et l'activation du menu se fait en réalisant un clic sur son titre. La liste d'items est alors déroulée au dessous. Noter que le titre du menu n'est pas un item du menu.

Scrollbar:

barre de défilement dont l'ascenseur (curseur qui se déplace dans la barre) exprime par sa position une valeur comprise dans l'intervalle associé à la barre. Peut servir de "doseur". A chaque activation de la 'scrollbar', elle génère un événement.

5.1.3. Configuration du graphique Prolog II+ au démarrage

Au lancement Prolog compte une barre de menus, trois fenêtres prédéfinies dont une seule est visible, il s'agit de :

- la fenêtre "console" de type *TTY* qui est par défaut, l'unité courante de lecture et d'écriture. L'unité de lecture et l'unité d'écriture peuvent être changées respectivement par les primitives *input* et *output*.

Les deux autres fenêtres sont :

- la fenêtre "graphic" de type *GRAPHICS*, qui est l'unité graphique courante par défaut. L'unité graphique est l'unité sur laquelle agissent les primitives préfixées par *gr_*. L'unité graphique peut être changée par la primitive *gr_window*.
- la fenêtre "trace" de type *TTY* utilisée en mode *debug* ou *trace*.

La barre de menus comporte des menus prédéfinis:

- le menu File qui permet de créer des fenêtres d'édition et sortir de Prolog.
- le menu Windows qui permet de rendre visibles ou invisibles les fenêtres existantes.
- le menu Control qui permet de générer une interruption utilisateur, passer en mode debug, exécuter rapidement certains prédicats prédéfinis Prolog.
- le menu Tools qui permet d'activer certains outils.
- certains menus du système hôte.

Nous avons regroupé ici des primitives très générales du graphisme.

init_screen

initialise le mode graphique. Ce prédicat est automatiquement lancé au chargement du module *graphic.mo*.

end_screen

désinstalle le graphique, c'est à dire, ferme les fenêtres, supprime les menus, supprime l'accès aux prédicats graphiques.

get_screen(x,y)
get_screen(x,y,N)

Unifie *x* et *y* respectivement avec la largeur et la hauteur de l'écran en pixels. Dans la forme à trois arguments, *N* est unifié avec 1 si l'écran ne possède que deux couleurs (noir et blanc), un nombre supérieur à 1 sinon.

graphic_system(s)

Unifie *s* avec une chaîne de caractères qui est fonction du système graphique hôte : "M" pour Motif, "W" pour Windows3 ou "P" pour Presentation Manager.

Le fichier *int_edit.mo* permet, si on le désire, l'utilisation de l'éditeur intégré par les prédicats *edit* et *editm* à la place de l'éditeur défini par la variable d'environnement *PrologEdit*. Il faut pour cela effacer les buts suivants:

```
> reload("int_edit.mo");
> exit("initial.po");
```

(pour éviter de refaire la même commande à chaque lancement de Prolog).

5.1.4. Conventions utilisées dans ce chapitre

Dans les règles exposées dans ce chapitre, *x* et *y* représentent les coordonnées horizontales (*x*) et verticales (*y*). L'axe positif des *x* est dirigé *vers la droite*. L'axe positif des *y* est dirigé *vers le bas*. Se rappeler que l'origine par défaut (i.e. x=0, y=0) coïncide avec le coin intérieur supérieur gauche de l'élément de référence. Celui-ci est la fenêtre graphique courante pour un tracé, la fenêtre parente pour un objet qui n'est pas une fenêtre, et l'écran pour une fenêtre.

Les coordonnées *x* ou *y* peuvent être exprimées indifféremment par des nombres entiers ou réels.

De même les variables commençant par *r* représenteront des arbres définissant un rectangle. Cet arbre doit être de la forme:

<x1,y1>.<x2,y2> ou bien *<x1,y1,x2,y2>*

où *x1,y1* sont les coordonnées du coin supérieur gauche, et *x2,y2* celles du coin inférieur droit.

On appellera booléen, un entier qui prend les valeurs 0 ou 1 et qui sert de valeur de vérité pour une propriété (visibilité d'un objet,…).

L'utilisateur peut nommer un objet à l'aide d'une chaîne, cette identification pourra servir pour toutes les primitives de création ou gestion de l'objet. Il peut également laisser Prolog attribuer à l'objet une identification par défaut (ce sera un entier) au moment de sa création (en laissant une variable libre), cette identification pourra être utilisée ultérieurement pour toutes les primitives de gestion de l'objet. Ou encore, pour les objets qui ne sont pas des fenêtres, il peut les nommer à l'aide d'un identificateur qui sera valable pour toutes les primitives agissant sur ces objets.

5.2. Primitives de gestion des fenêtres

5.2.1. Création, destruction d'une fenêtre

new_window(s,t)
s *new_window(s,t,s')*
new_window(s,t,v,r)
s *new_window(s,t,v,s',r)*
new_window(s,t,v,x1,y1,x2,y2)
s *new_window(s,t,v,s',x1,y1,x2,y2)*

Créent une nouvelle fenêtre, et une nouvelle unité d'entrée/sortie de nom *s* et de type *t*, où *s* est l'identification de la fenêtre qui doit être différente de celles des unités existant déjà. *s* peut être une chaîne. Si *s* est une variable libre au moment de l'appel, elle sera unifiée avec une identification de fenêtre fournie par Prolog.

La chaîne de caractères *s'* est affichée dans le bandeau de la fenêtre, comme titre de la fenêtre; si *s'* n'est pas spécifiée c'est *s* qui est inscrite.

v est un booléen (1 ou 0) indiquant si la fenêtre doit être visible ou pas. Par défaut, la fenêtre est créée visible.

x1,y1 sont les coordonnées du coin intérieur supérieur gauche de la nouvelle fenêtre par rapport à l'écran.

x2,y2 sont les coordonnées du coin intérieur inférieur droit de la nouvelle fenêtre, l'origine étant située au coin supérieur gauche de l'écran.

t est de la forme *type*, ou bien *type.liste_attributs*, où *type* détermine le type de la fenêtre, et donc les opérations qui peuvent lui être appliquées.

Les valeurs possibles pour *type* sont:

"EDIT" ou *:edit*
pour la création d'une fenêtre EDIT.
s "FRAMEPANEL" ou *:framepanel*

pour la création d'une fenêtre FRAMEPANEL.

"GRAPHICS" ou *:graphics*
pour la création d'une fenêtre GRAPHICS.

"MODAL" ou *:modal*
pour la création d'une fenêtre GRAPHICS avec l'attribut MODAL.

"TTY" ou *:tty*
pour la création d'une fenêtre TTY.

Les valeurs possibles pour un attribut de *liste_attributs* sont:

<"DISTANCE", n> ou < :distance, n>
n est un entier et désigne une portion de la partie dessinable de la fenêtre réservée pour une zone de création de boutons. Cette zone se situe en haut de la fenêtre, sur toute la largeur et sur une hauteur égale à *n* % de la hauteur totale. Cet attribut est valable pour tout type de fenêtre, mais n'a pas d'utilité pour les fenêtres FRAMEPANEL.

- <"FONT", n > ou < :font, n >
Définit le numéro *n* de la police de caractères utilisée pour le texte d'une fenêtre type texte. La fonte associée à *n* est dépendante du système hôte.

- <"FONTSIZE", n > ou < :fontsize, n >
Définit la taille de la police de caractères utilisée dans une fenêtre de texte.

<"MODAL", b> ou < :modal, b>
b est un booléen indiquant si la fenêtre est bloquante ou pas. La valeur par défaut est *b* = 0. Cet attribut est valable pour tout type de fenêtre. Cette fenêtre si elle est bloquante, reste au dessus de toutes les autres fenêtres, et tous les événements extérieurs à la fenêtre sont désactivés (excepté l'interruption utilisateur). Les événements extérieurs à la fenêtre sont réactivés lorsque la fenêtre est tuée. Les fenêtres modales peuvent être empilées.

s <"NO_RESIZE", b> ou < :no_resize, b>
b est un booléen indiquant si le redimensionnement de la fenêtre au moyen de la souris (ou d'une combinaison de touches prédéfinie pour certaines machines) est interdit ou pas. Par défaut *b* vaut 0. Quelle que soit la valeur de *b*, une fenêtre peut toujours être redimensionnée par programme.

<"SHAPE", n> ou < :shape, n>
n est un entier déterminant la forme de la fenêtre. Les formes standard sont 0: fenêtre avec titre, barre de menus et barres de défilement; 1: fenêtre cadre. Par défaut les fenêtres graphiques modales ont l'attribut 1, et les autres types de fenêtres ont l'attribut 0. Cet attribut est valable pour tout type de fenêtre. On notera que les fenêtres GRAPHICS et FRAMEPANEL n'ont pas de barres de défilement.

- <"SAVE", r > ou < :save, r >
  Cet attribut ne peut être appliqué qu'à une fenêtre graphique ou modale. Il définit le rectangle *r* de la fenêtre auquel est associé un bitmap de rafraîchissement (dans le cas de machines couleur une grande zone peut demander une place mémoire importante).

  <"SAVE", b > ou < :save, b >
  b est un booléen indiquant si un bitmap de rafraîchissement est associé à la fenêtre. La valeur par défaut est *b* = 0.
  Si *b* =1, un bitmap de rafraîchissement de la taille de la fenêtre est créé.

  Exemple de création d'une fenêtre graphique avec sauvegarde automatique :

```
> new_window("query","GRAPHICS".<"SAVE",1>.nil);
```

  Les deux buts suivants sont équivalents:

```
> new_window("fen","MODAL");
> new_window("fen","GRAPHICS".<"MODAL",1>.nil);
```

create_window(s,t,s',r1)
create_window(s,t,v,s',r1)
create_window(s,t,v,s',r1,r2)

Similaire à *new_window*, mais permet de définir un système de coordonnées avec échelle.

r1 est de la forme <x1,y1,x2,y2> où x1,y1,x2,y2 sont des réels indiquant les coordonnées de la partie 'dessinable' de la fenêtre dans un écran [0,1] * [0,1].

r2 est de la forme <x1,y1,x2,y2> où x1,y1,x2,y2 sont des entiers définissant un système de coordonnées locales propre à l'utilisateur. Ce paramètre n'est à préciser que pour des fenêtres GRAPHICS et FRAMEPANEL. Si le paramètre *r2* n'est pas spécifié, les coordonnées graphiques dans la fenêtre seront données en pixels.

Par exemple si une fenêtre est créée par l'appel suivant:

```
> create_window("mywindow", "GRAPHICS", 1, "mywindow",
<0.1E0,0.1E0,0.6E0,0.7E0>, <0,0,100,100>);
```

quelle que soit la taille de l'écran, un rectangle dessiné par *gr_rect(1,0,0,100,100)* la remplira, *gr_moveto(50,50)* positionnera au centre etc...

Autre exemple:

```
> create_window("ww","GRAPHICS",1,"mywindow",
<1e-1,1e-1,5e-1,5e-1);
```

kill_window(s)

Détruit la fenêtre de nom *s*. Le nom *s* ne peut pas être celui d'une fenêtre prédéfinie, ce doit être une fenêtre créée par *new_window* ou *create_window*. Si la fenêtre détruite était l'entrée ou la sortie courante, l'unité d'entrée ou de sortie précédemment active redevient l'unité courante d'entrée ou de sortie. Si la fenêtre détruite était la fenêtre graphique courante, c'est la fenêtre prédéfinie "graphic" qui redevient l'unité graphique courante. Voir aussi le prédicat *kill_object*.

5.2.2. Configuration, manipulation d'une fenêtre

clear_window(s)

Efface le contenu de la fenêtre de nom *s*. Ne modifie pas ses objets éventuels et ne supprime pas les événements liés à la fenêtre qui pouvaient être dans la queue Prolog. Est valable pour tout type de fenêtre (Sans effet sur une fenêtre FRAMEPANEL).

file_window(s1)
file_window(s1,s2)

Créent, si elle n'existe pas déjà, une fenêtre EDIT de nom *s2* initialisée avec le texte contenu dans le fichier de nom *s1*. *file_window(s1)* équivaut à *file_window(s1,s1)*.

front_window(s)

Unifie *s* avec le nom de la fenêtre Prolog se trouvant au premier plan et ayant le focus.

get_window(s,v)
get_window(s,v,x1,y1,x2,y2)
s *get_window(s,t,v,x1,y1,x2,y2)*

Renseigne sur le type, les attributs, la visibilité et les coordonnées de la fenêtre *s*. La forme à deux arguments renseigne uniquement la visibilité. Unifie successivement par backtracking les paramètres *s,t,v,x1,y1,x2,y2* avec les valeurs correspondant à chacune des fenêtres. Voir la primitive *new_window(s,t,v,x1,y1,x2,y2)* pour la signification des paramètres. Le type et les attributs sont rendus sous la forme identificateur. La valeur de l'attribut *:distance* est donnée en pixel et non plus en pourcentage de la partie dessinable.

gr_window(s)

La fenêtre ou la zone graphique *s* devient l'unité graphique courante.

gr_window_is(s)

Unifie *s* avec le nom de l'unité graphique courante.

- *option fermeture*

  Lorsque l'on clique dans la case de fermeture d'une fenêtre, l'unité correspondante est fermée et son nom est retiré du menu contenant les noms des fenêtres, excepté pour les fenêtres prédéfinies. Il est possible de rendre une fenêtre invisible sans la fermer en appuyant sur la touche *option* en même temps que la case de fermeture est cliquée.

- *print_window(s)*
- *print_window(s,f,t)*

  Imprime le contenu de la fenêtre de texte *s* avec la fonte *f* de taille *t*. Mêmes conventions que *gr_text(f,t,l)* (Voir 5.5. Mode de dessin et d'écriture). La forme avec un seul argument imprime la fenêtre avec la fonte courante de la fenêtre.

- *gr_print(s)*
- *gr_print(s,_rx,_ry,x0,y0)*
- *gr_print(s,r1,r2)*

  Imprime le contenu de la fenêtre graphique *s*. On peut spécifier, grâce à la troisième forme, la zone à imprimer, qui est le contenu du rectangle *r1*, le résultat sur papier se situant dans le rectangle *r2*. Les réductions et translations éventuelles sont automatiques. La forme avec un seul argument imprime la fenêtre sans modification de format. La forme à 5 arguments permet de spécifier la réduction selon les axes des X, des Y sous la forme de deux nombres *_rx* et *_ry* compris entre 0e0 et 1e0. On peut donner, en *x0* et *y0* la position de l'origine de l'impression sur le papier.

  ```
  > gr_print("graphic",0.5e0,0.5e0,0,0);
  ```

  imprimera la fenêtre "graphic", réduite au quart (chaque dimension étant réduite de moitié).

reset_window(s)

Repositionne le pointeur de lecture au début de la fenêtre EDIT de nom *s*. Echoue si *s* n'est pas une fenêtre EDIT créée par *new_window* ou *create_window*. Le point d'écriture reste toujours à la fin du texte.

save_window(s1)
save_window(s1, s2)

Sauve le contenu de la fenêtre de texte de nom *s1* dans le fichier de nom *s2*. *save_window(s1)* équivaut à *save_window(s1,s1)*.

set_window(s,b)

Permet de modifier les attributs ou la visibilité d'une fenêtre. *s* est l'identification de la fenêtre. *b* peut être soit un booléen: il indique alors si la fenêtre sera visible (0 : invisible, 1 : visible), soit un attribut (voir *new_window*).
Les attributs possibles sont pour les fenêtres graphiques et modales:
<"SAVE", v > où *v* est un booléen

Ce prédicat avec cet attribut peut aussi être employé pour une 'drawing area'. *s* est alors son identification.
Si *v* =1, un bitmap de rafraîchissement de la taille de l'objet est créé.

- <"SAVE", r > où *r* est un rectangle.

Pour une fenêtre de texte:

- <"FONT", n >
- <"FONTSIZE", n >

Exemple:

```
> set_window("console",<"FONTSIZE",12>);
```

set_window(s,b,x1,y1,x2,y2)

Permet de modifier la visibilité, les dimensions et l'emplacement des fenêtres existantes.

s est l'identification (chaîne de caractères ou entier) de la fenêtre, notamment "console", "graphic", "trace" pour les fenêtres prédéfinies.

b est un booléen, indique si la fenêtre sera visible (0: invisible, 1: visible).

x1,y1 sont les nouvelles coordonnées, par rapport à l'écran, du coin intérieur supérieur gauche de la fenêtre.

x2,y2 sont les nouvelles coordonnées du coin intérieur inférieur droit de la fenêtre, l'origine étant située au coin supérieur gauche de l'écran.

Exemple:

```
> set_window("graphic",1,50,50,300,300);
```

5.2.3. Rafraîchissement des zones graphiques

Les fenêtres GRAPHICS ou les 'drawing area' peuvent être rafraîchies automatiquement en leur associant un bitmap auxiliaire (voir les primitives *new_window* ou *set_window* ou *new_drawing_area*).

Dans le cas contraire, il existe une coroutine attachée aux zones graphiques, et qui est toujours déclenchée lorsque ces zones nécessitent un rafraîchissement. Cette coroutine est:

```
exec(:gr_update(u))
```

où *u* représente le nom de l'unité à rafraîchir. La règle *:gr_update* n'est pas définie dans l'environnement standard, et n'est pas lancée si l'utilisateur ne la définit pas. L'utilisateur peut donc, en écrivant une règle *:gr_update/1*, définir par programme ce que doit être le rafraîchissement d'une zone graphique sans utiliser de bitmap auxiliaire.

5.3. Primitives élémentaires de gestion des objets attachés à une fenêtre

Nous les avons qualifiées de primitives élémentaires car elles servent à réaliser des opérations élémentaires telles que la création, la configuration et l'interrogation d'un objet. Ensuite elles manipulent des éléments très simples, à savoir: un objet ou un attribut d'objet ou un événement survenu sur un objet.

L'ensemble des ces primitives ne permet pas de manipuler et configurer complètement les menus, il y a en plus de celles-ci, des primitives spécifiques aux menus, décrites au paragraphe 5.4.

5.3.1. Création, destruction d'objets

L'identification d'un objet peut être soit une chaîne, soit un identificateur Prolog II+, soit, quand elle est générée automatiquement par Prolog, un entier. Pour ce dernier cas, il suffit d'appeler le prédicat de création de l'objet avec une variable libre à la place de l'identification, la variable sera alors unifiée avec un entier attribué à l'objet, qui permettra de l'identifier par la suite.

Certains objets attendent un titre ou un texte, il est déterminé par une chaîne de caractères Prolog qui peut être vide.

Une constante entière est utilisée pour spécifier la visibilité de l'objet à sa création : 0 signifie non visible, 1 signifie visible.

Il est possible de modifier l'attachement par défaut des objets, à l'aide des attributs *:top_attach, :left_attach, :bottom_attach, :right_attach* (qui définissent respectivement l'attachement de l'objet au bord haut, gauche, bas, droit) à ajouter dans la liste des options de création. Si l'un des (ou les deux) attributs *:top_attach* et *:bottom_attach* est mentionné, cela redéfinit l'attachement vertical et(ou) si l'un des (ou les deux) attributs *:left_attach* et *:right_attach* est mentionné, cela redéfinit l'attachement horizontal.

Un programme Prolog peut être lié aux événements (s'il en existe) de l'objet créé, qui dès qu'ils se produisent provoquent la suspension du programme Prolog en cours pour lancer le prédicat associé à l'objet, avec en premier argument, l'identification de l'objet et éventuellement en deuxième argument une information supplémentaire (caractère tapé…). Lorsque le prédicat se termine, la démonstration suspendue reprend son cours normalement. Cela permet de gérer "en direct" la saisie dans les écrans graphiques.
Il est également possible de différer le traitement des données saisies ou des événements. Pour cela il suffit de spécifier *sys:get_event* à la place du prédicat, ce qui aura pour effet de mettre simplement cet événement dans la queue d'événements de Prolog (elle peut être scrutée par *get_event* ou vidée par *clear_events*).
Ou encore, il est possible de ne rien faire. Pour cela il suffit de spécifier *nil* à la place du prédicat, ce qui aura pour effet d'ignorer les événements de l'objet.

s *kill_object(o)*

Détruit l'objet *o*. Est valable pour tout type d'objet.

s *new_check_button(o,p,v,s,b,<x1,y1,x2,y2>)*

Crée un *'check button'* identifié par *o*, dont le texte est *s*. Le bouton et le texte seront inscrits dans un rectangle de coordonnées *x1*, *y1*, *x2*, *y2* dans la fenêtre parente *p*. Il sera visible selon la valeur de *v*.
x2 et *y2* doivent être libres ils seront alors unifiés avec les coordonnées effectives du coin inférieur droit de l'objet après création.
b peut être *nil*, *sys:get_event* ou un identificateur de règle. Dans ce dernier cas, à chaque activation du bouton le but *b(o)* est lancé.

s *new_drawing_area(o,p,v,<x1,y1,x2,y2>,r2,l)*

s *new_drawing_area(o,p,v,<x1,y1,x2,y2>,l)*

Crée une '*zone graphique'* identifiée par *o*. Elle est inscrite dans un rectangle de coordonnées *x1*, *y1*, *x2*, *y2* dans la fenêtre parente *p*. Elle sera visible selon la valeur de *v*. *r2* a la même signification que dans le prédicat *create_window* (système de coordonnées locales). *l* est une liste d'options, éventuellement vide (*nil*), formée à partir des identificateurs :
:no_border si l'on ne veut pas de bordure autour de la zone.
:save pour avoir un bitmap de sauvegarde.
:top_attach, *:left_attach*, *:right_attach*, *:bottom_attach*

s *new_push_button(o,p,v,s,b,<x1,y1,x2,y2>)*

s *new_push_button(o,p,v,s,b,<x1,y1,x2,y2>,l)*

Crée un '*push button'* identifié par *o*, dont le texte est *s*, dont les coordonnées sont *x1*, *y1*, *x2*, *y2* dans la fenêtre parente *p*. Il sera visible selon la valeur de *v*. Si *x2* et *y2* sont libres ils seront alors unifiés avec les coordonnées effectives du coin inférieur droit de l'objet après création. *b* peut être *nil*, *sys:get_event* ou un identificateur de règle. Dans ce dernier cas, à chaque activation du bouton, le but *b(o)* est lancé. *l* est une liste d'options, éventuellement vide (*nil*), formée à partir des identificateurs :
:buttonD permet de le définir comme 'bouton par défaut', c'est à dire que tout *carriage return* qui ne sera pas capté par un autre objet activera ce bouton. Ces boutons sont conçus pour qu'il n'y en ait qu'un seul par fenêtre. Si ce n'est pas le cas, le comportement n'est pas garanti et est dépendant du système hôte.
:top_attach, *:left_attach*, *:right_attach*, *:bottom_attach*

s *new_radio_button(o,g,p,v,s,b,<x1,y1,x2,y2>)*

Crée un *'radio button'* identifié par *o*, dont le texte est *s*. *g* est un identificateur Prolog qui représente le groupe auquel appartient le bouton. Le bouton et le texte seront inscrits dans un rectangle de coordonnées *x1*, *y1*, *x2*, *y2* dans la fenêtre parente *p*. Il sera visible selon la valeur de *v*.

x2 et *y2* doivent être libres ils seront alors unifiés avec les coordonnées effectives du coin inférieur droit de l'objet après création.
b peut être *nil*, *sys:get_event* ou un identificateur de règle. Dans ce dernier cas, à chaque activation du bouton le but *b(o)* est lancé.

s *new_edit_field(o,p,v,b,<x1, y1, x2, y2>,l)*

Crée un *'editfield'* identifié par *o*, dont les coordonnées sont *x1*, *y1*, *x2*, *y2* dans la fenêtre parente *p*. Il sera visible selon la valeur de *v*.
b peut être *nil*, *sys:get_event* ou un identificateur de règle. Dans ce dernier cas, à chaque caractère tapé dans l'editfield, le but *b(o,c)* est lancé où *c* est le caractère tapé, et à chaque événement de gain ou perte de focus le but *b(o)* est lancé.
l est une liste d'options, éventuellement vide (*nil*), formée à partir des identificateurs :

:multiple	si le champ peut comporter plusieurs lignes.
:hscroll	pour avoir une barre de défilement horizontale.
:vscroll	pour avoir une barre de défilement verticale.
<chars_nb,n>	si le champ est limité à *n* caractères (*n* entier).

:top_attach, *:left_attach*, *:right_attach*, *:bottom_attach*
Par défaut l'*editfield* est monoligne, sans barre de défilement, et le nombre de caractères est limité à 1024.

s *new_label(o,p,v,s,<x1,y1,x2,y2>,l)*

Crée un 'label' identifié par *o*, dont le texte est *s*, inscrit dans un rectangle de coordonnées *x1*, *y1*, *x2*, *y2* dans la fenêtre parente *p*. Il sera visible selon la valeur de *v*.
Si *x2* et *y2* sont libres ils seront alors unifiés avec les coordonnées effectives du coin inférieur droit de l'objet après création.
l est une liste d'options, éventuellement vide (*nil*), formée à partir des identificateurs :

:left	si le texte doit être cadré à gauche dans le rectangle.
:right	si le texte doit être cadré à droite dans le rectangle.
:no_border	si le texte ne doit pas être encadré.

:top_attach, *:left_attach*, *:right_attach*, *:bottom_attach*
Pour obtenir un texte centré, il faut les deux attibuts *:left* et *:right*.
Par défaut le texte est encadré et son alignement est dépendant du système graphique hôte.

s *new_listbox(o,p,v,b,<x1, y1, x2, y2>,l)*

Crée une *'listbox'* identifiée par *o*, dont les coordonnées sont *x1*, *y1*, *x2*, *y2* dans la fenêtre parente *p*. Elle sera visible selon la valeur de *v*. La définition des items de la listbox devra se faire par la primitive *set_attribute*.
b peut être *nil*, *sys:get_event* ou un identificateur de règle. Dans ce dernier cas, à chaque action de sélection dans la listbox, le but *b(o,n,m)* est lancé où *n* est le numéro de l'item affecté et *m* le modifieur utilisé.
l est une liste d'options, éventuellement vide (*nil*), formée à partir des identificateurs :

:multiple	si la sélection peut comporter plusieurs items.

:hscroll si la barre de défilement horizontale est souhaitée.
:top_attach, *:left_attach*, *:right_attach*, *:bottom_attach*
Par défaut la listbox est en mode sélection simple (au plus 1 item sélectionné) et possède une barre de défilement verticale qui peut être invisible si le nombre d'items de la boite est insuffisant.

s *new_popup_menu(o,p)*

Crée un support de *'popup menu'* identifié par *o*, attaché à la fenêtre *p*. Aucun item n'est encore créé, il faut pour cela utiliser la primitive *set_menu.*
Remarque : il n'est pas nécessaire de créer un support de *'popup menu'* avant d'utiliser la primitive *set_menu*. En effet si *set_menu* est utilisé avec un nom de fenêtre, cela fait référence au *'popup menu'* de cette fenêtre. S'il n'existe pas, *set_menu* le crée, mais son identification n'est pas connue.

En supposant que mygraph est l'identification d'une fenêtre graphique déjà créée, la suite de buts ci-dessous crée un popup menu composé de la liste d'items : "doLine", "doRect", "doOval" avec, attachée à chacun des items, respectivement la procédure : doLine/1, doRect/1, doOval/1.

```
> new_popup_menu(popgraph,mygraph);
> set_menu(popgraph,nil,doLine.doRect.nil);
> set_menu(mygraph,"new".nil,doOval);
```

s *new_pulldown_menu(o,p,s)*

Crée un support de *'pulldown menu'* identifié par *o*, de nom *s* (chaîne de caractères), attaché à la fenêtre *p*. Ses items seront ensuite créés par la primitive *set_menu*. N'est pas valable pour les fenêtres de forme cadre.

La suite de buts ci-dessous crée un pulldown menu de titre "custom" composé de la liste d'items : "myColor", "myMode", "myFont", "myPen" avec, attachée à chacun des items, respectivement la procédure : myColor/1, myMode/1, myFont/1, myPen/1.

```
> new_pulldown_menu(mymenu,mygraph,"custom");
> set_menu(mymenu,nil,myColor.myMode.myFont.miPen.nil);
> set_menu(mymenu,"miPen".nil,myPen);
```

s *new_scrollbar(o,p,v,b,<x1,y1,x2,y2>,l)*

Crée une *'scrollbar'* identifiée par *o*, dont les coordonnées sont *x1*, *y1*, *x2*, *y2* dans la fenêtre parente *p*. Elle sera visible selon la valeur de *v*.
Si *x2* et *y2* sont libres ils seront alors unifiés avec les coordonnées effectives du coin inférieur droit de l'objet après création.
b peut être *nil*, *sys:get_event* ou un identificateur de règle. Dans ce dernier cas, à chaque activation de la barre de défilement, le but *b(o)* est lancé.
l est une liste d'options, éventuellement vide (*nil*), formée à partir des identificateurs :
:vscroll si la barre doit être verticale.
:hscroll si la barre doit être horizontale.
:top_attach, *:left_attach*, *:right_attach*, *:bottom_attach*

L'orientation par défaut est déterminée par les coordonnées, sachant qu'une barre est plus longue que large.

5.3.2. Configuration d'objets

s *get_attribute(o,a,v)*

Permet de se renseigner sur l'état des objets ou leurs propriétés.
o est l'identification de l'objet.
a est l'identification de l'attribut.
v sera unifié avec la valeur de l'attribut interrogé.
Les attributs possibles sont les mêmes que ceux de la primitive *set_attribute,* excepté les attributs *focus* et *customColor*, complétés des attributs suivants qui sont des propriétés de création non modifiables :
:parent : *v* est unifié avec l'identification du parent de l'objet *o*.
:type : *v* est unifié avec un identificateur représentant le type de l'objet *o*. Les valeurs possibles sont: *:graphics*, *:tty*, *:edit*, *:framepanel*, *:label*, *:push_button*, *:check_button*, *:radio_button*, *:scrollbar*, *:edit_field*, *:popup_menu*, *:pulldown_menu*, *:listbox*, :*drawing_area*.
:*fontheight* : *v* est la hauteur en pixels de la fonte utilisée par l'objet. La hauteur de la fonte est la hauteur du rectangle dans lequel s'inscrivent les caractères (l'interligne est compris).
:group : *v* est unifié avec l'identificateur de groupe du 'radio button' *o*.
:items_nb : *v* est unifié avec le nombre d'items de la 'listbox' *o*.

s *set_attribute(o,a,v)*

Permet de modifier ou de préciser la configuration des objets.
o est l'identification de l'objet.
a est l'identification de l'attribut.
v est la valeur de l'attribut.
Les attributs possibles dépendent du type de l'objets. Les attributs généraux sont les suivants :
:activity : indique si l'objet ou l'item de menu réagit aux événements ou pas. *v* vaut respectivement 1 ou 0. N'est pas valable pour les labels.
:background : concerne la couleur de fond de l'objet. *v* est un entier ou un identificateur et suit les mêmes conventions que dans la primitive *gr_color2* (Voir 5.5. Mode de dessin et d'écriture).
:bottom_right : concerne la position du coin extérieur inférieur droit de l'objet. *v* est de la forme <*p1, p2*> où *p1* est l'abscisse, *p2* est l'ordonnée. A pour effet le redimensionnement de l'objet. N'est pas valable pour les menus.
:cursor : concerne la forme du curseur de la souris quand il est à l'intérieur de la fenêtre *o*. *v* est un des identificateurs suivants : *:default, :wait, :cross*. Pour *set_attribute*, si *o* vaut *nil*, la définition se rapporte à toutes les fenêtres.

:customColor : définit la couleur utilisateur pour l'objet *o*. *v* est de la forme *<_r, _v, _b>* où *_r*, *_v*, *_b* sont des entiers ou des réels et représentent la couleur en composantes rouge, vert, bleu comme défini dans la primitive *gr_color3* (Voir 5.5. Mode de dessin et d'écriture). Cet attribut n'est pas valable pour la primitive *get_attribute*.

:focus : donne le focus à l'objet *o*. *v* est ignoré. N'est pas un attribut valide pour la primitive *get_attribute*.

:font : concerne la fonte utilisée par l'objet. *v* a la même signification que dans la primitive *gr_font* (Voir 5.5. Mode de dessin et d'écriture). La fonte définit un style et une taille. La taille pourra être obtenue par la primitive *get_attribute* avec l'attribut *:fontheight*.

:foreground : concerne la couleur de traçage (texte ou dessin) de l'objet. *v* est un entier ou un identificateur et suit les mêmes conventions que dans la primitive *gr_color2* (Voir 5.5. Mode de dessin et d'écriture).

:predicate : *v* est le prédicat Prolog attaché à l'objet. N'est pas valable pour les labels (puisqu'ils ne sont pas sensibles), ni pour les menus et leurs items (voir pour ça la primitive *set_menu*).

Remarque : il peut être utilisé pour les unités graphiques, il correspond alors au prédicat qui doit être lancé, si l'unité n'a pas de bitmap, sur un événement "expose" (nécessité de retracer une partie de l'unité). La valeur attribuée par défaut est *gr_update*.

Remarque : ne pas confondre avec le prédicat qui est lancé sur un événement clic, il est lui lié à une portion de l'unité (Voir le prédicat *gr_sensitive*). Il n'est pas modifiable par *set_attribute*.

:protected permet de désactiver l'option de fermeture dans le menu par défaut d'une fenêtre.

:rank : concerne la numérotation des objets dans leur parent. *v* est le rang local de l'objet dans son parent, ce numéro est unique pour un parent donné. Implicitement c'est le numéro d'ordre de création. Cet attribut n'est pas valable pour les fenêtres.

:state : concerne l'état de l'objet: sélectionné ou non sélectionné. *v* vaut respectivement 1 ou 0. Est valable uniquement pour les check buttons et les radio buttons.

:text : concerne le texte d'un champ d'édition, le titre d'une fenêtre, le libellé d'un objet de type bouton ou d'un label. Dans le cas général, *v* est une chaîne de caractères. Pour un champ d'édition, *v* peut être de la forme *<s, 1>* ce qui signifie que la chaîne de caractères *s* doit être ajoutée (mode *append*) au contenu du champ d'édition.

:text_selection : concerne la sélection dans un champ d'édition. *v* est de la forme *<p1, p2>* où *p1* est l'indice du premier caractère sélectionné (débute à 0), *p2* est l'indice suivant le dernier caractère sélectionné, *p2 >= p1*.

:top_left : concerne la position du coin supérieur gauche de l'objet. *v* est de la forme *<p1, p2>* où *p1* est l'abscisse, *p2* est l'ordonnée. A pour effet le déplacement de l'objet. N'est pas valable pour les menus.

:user_field : concerne le champ utilisateur qui n'est pas du tout géré par Prolog. *v* est un entier. Le programmeur peut s'en servir par exemple, comme variable globale associée à l'objet.

:visibility : concerne la visibilité de l'objet. *v* vaut 0 si l'objet est invisible, 1 si l'objet est visible. N'est pas valable pour les menus.

:width_height : *v* est de la forme *<p1, p2>* où *p1* est la largeur de l'objet, *p2* sa hauteur. N'est pas valable pour les menus.

Les attributs spécifiques aux 'listbox' sont les suivants :

:items : concerne la liste des items d'une listbox. *v* est une liste de chaînes Prolog qui décrit, dans le même ordre, tous les items (par leur texte) de la listbox.

:selected_items : concerne la liste des items sélectionnés d'une listbox. *v* est la liste des rangs des items sélectionnés. Le premier item a le rang 1. S'il est mentionné un rang qui n'existe pas, il est ignoré. S'il est mentionné plus d'un rang pour une listbox en sélection simple, c'est la dernière sélection qui est prise en compte.

:state(i) : concerne l'état d'un item de listbox : sélectionné ou non sélectionné. *v* vaut respectivement 1 ou 0. *i* est un entier pour le numéro de l'item concerné dans la listbox. Le premier item de listbox est représenté par l'entier 1.

:text(i) : concerne le texte d'un item de listbox. *i* est un entier pour le numéro de l'item concerné dans la listbox. Le premier item de listbox est représenté par l'entier 1. *v* peut avoir différentes valeurs selon les situations suivantes:

– Interrogation sur le texte de l'item: *v* est une chaîne de caractères. Si l'item n'existe pas , *v* est la chaîne vide.

– Modification du texte de l'item : *i* est un numéro d'item existant, *v* est une chaîne de caractères non vide.

– Suppression de l'item : *i* est un numéro d'item existant, *v* est la chaîne vide.

– Création de l'item : *i* est un numéro d'item qui n'existe pas, *v* est une chaîne de caractères éventuellement vide.

– Insertion de l'item : *v* est de la forme *<p1, 1>* où *p1* est une chaîne de caractères éventuellement vide pour désigner le texte de l'item créé à la position indiquée par *i*.

:toplist : v est le numéro du premier item visible dans la listbox.

Les attributs spécifiques aux 'scrollbar' sont les suivants :

:scrollb_page : concerne l'amplitude du déplacement du curseur lorsqu'un saut est demandé (clic entre une flèche et le curseur). *v* est un entier.

:scrollb_pos : *v* désigne la position du curseur dans la 'scrollbar'. *v* est une des valeurs entières de l'intervalle de valeurs de la 'scrollbar'. Le 0 est en haut pour les barres verticales, à gauche pour les barres horizontales.

:scrollb_range : concerne l'intervalle de valeurs de la 'scrollbar'. *v* est un entier. S'il est positif, les valeurs seront comprises entre 0 et *v*. S'il est négatif, les valeurs seront comprises entre *v* et *–v*.

:scrollb_step : *v* est un entier, il désigne l'amplitude du déplacement élémentaire du curseur lorsqu'une flèche de 'scroll' est activée. L'amplitude minimum est 1, c'est aussi la valeur par défaut. Pour obtenir la nouvelle position il faut enlever ou ajouter *v* à la position actuelle.

s *get_local_object(p, n, o)*

Unifie *o* avec l'identification de l'objet de parent *p* et de rang local *n* dans *p*.

s *get_objects(p, l_o)*

Unifie *l_o* avec la liste des objets dont le parent est *p.*

5.3.3. Gestion des événements

s *get_event(b,e,o)*

Teste si un événement de type défini par *e* concernant l'objet *o* est présent dans la queue d'événements de Prolog. Si c'est le cas, le premier événement correspondant (le plus ancien dans le temps) est lu donc supprimé de la queue et l'unification des arguments est tentée. Sinon *get_event* agit en fonction de *b* qui doit être connu à l'appel.
Si *b* vaut 0, *get_event* n'est pas bloquant et génère un backtracking.
Si *b* vaut 1, *get_event* est bloquant et un tel événement est attendu, puis lu et enfin l'unification tentée.
o est soit l'identification d'un objet, soit une variable libre. Dans ce dernier cas tous les objets sont concernés.
e est soit une variable libre, dans ce cas tous les événements sont concernés, soit de la forme:

:char(c) : pour les 'edit_field' et les unités (fenêtres ou zones de dessin) GRAPHICS, où *c* est le code du caractère.

:click : pour un bouton ("push", "check", "radio") ou une 'scrollbar'. Aucun résultat n'est transmis, simplement un succès ou un échec.

:click(i,m) : pour les menus ou les listbox, où *i* est le numéro de l'item désigné et *m* donne l'état des principales touches de modification. m est la somme des valeurs :

1 si la touche SHIFT est appuyée,
2 si la touche CONTROLE est appuyée,
32 s'il s'agit d'un DO UBLE-CLIC

Remarque: Un double clic souris sur un item va générer la séquence d'événements suivante: *:click*(i,0) *:click*(i,32).

:click_down(x,y,m) : pour les fenêtres GRAPHICS et FRAMEPANEL et les 'drawing area', où *x,y* sont les coordonnées du clic relatives à l'objet, *m* donne l'état des principales touches de modification.

:click_up(x,y,m) : pour les unités (fenêtres ou zones de dessin) GRAPHICS, où *x,y* sont les coordonnées du clic relatives à l'objet, *m* donne l'état des principales touches de modification.

Remarque : Un double clic souris dans une unité graphique va générer la séquence d'événements suivante: *:click_down*(x,y,0) *:click_up*(x,y,0) *:click_down*(x,y,32) *:click_up*(x,y,0).

:focus_in : pour les 'edit_field'. Aucun résultat n'est transmis, simplement un succès ou un échec.

:focus_out : pour les 'edit_field'. Aucun résultat n'est transmis, simplement un succès ou un échec.

:update : pour les unités (fenêtres ou zones de dessin) GRAPHICS qui n'ont pas de bitmap et qui ont toujours le prédicat par défaut *sys:get_event* lié à l'unité.

:extern_event(s, i1, i2): pour les objets externes. s, i1 et i2 sont des données envoyées par l'utilisateur en même temps que l'événement, grace à la procédure *send_external_event*.

s *peek_event(b,e,o)*

Même principe et mêmes valeurs que la primitive *get_event(b,e,o)*, mais ne fait que consulter la queue d'événements, sans la modifier. L'événement *e* pour l'objet *o* n'est pas supprimé de la queue.

s *clear_events(e,o)*

Supprime tous les événements définis par *e* concernant l'objet *o*, aucune unification n'est faite, ce prédicat n'échoue jamais. *e* suit les mêmes conventions que pour *get_event(b,e,o)*.

5.4. Primitives spécifiques pour la gestion de menu

La primitive *set_menu* décrite dans cette section, permet de définir ou redéfinir des menus hiérarchiques.

Tous les menus, popup ou pulldown, sont attachés à une fenêtre. Il existe néanmoins un support particulier, pour accueillir des menus généraux à l'application, appelé la barre de menus de Prolog.

5.4.1. Sauvegarde et changement de la barre de menu

La barre de menu générale de Prolog est identifiée par *:prolog*. Pour connaître les identifications de ses menus, il faut effacer *sys:get_objects(:prolog,l)* qui en rendra la liste.

clear_menubar

Efface la barre de menus standard. Cette primitive permet la redéfinition complète de la barre de menus.

clear_menubar(s)

Efface la barre de menus de la fenêtre *s*.

s *restore_sysmenus(s,f,e)*

Ajoute dans la barre de menus de la fenêtre *s* les menus prédéfinis "File" et "Edit" en unifiant leur nom avec *f* et *e*. (La variable reste libre, si le menu n'existe pas sur la machine hôte).

- *save_menubar*

Mémorise la barre de menus courante. Cette règle ne peut être appelée qu'une fois.

restore_menubar

- Restaure la barre de menus sauvée avec *save_menubar*.

s Initialise la barre de menus de Prolog avec les menus généraux.

- *add_stdmenu(s)*

Cette primitive permet de réinsérer un ou plusieurs menus standard après un appel à *clear_menubar*. *s* doit être l'une des chaînes suivantes:

"apple"	: pour reinsérer le menu pomme
"file"	: pour reinsérer le menu Fichier
"edit"	: pour reinsérer le menu Editer
"find"	: pour reinsérer le menu Recherche
"control"	: pour reinsérer le menu Contrôle
"window"	: pour reinsérer le menu Fenêtres

5.4.2. Description d'un menu

set_menu(u,c,v)

Permet de définir ou redéfinir par *v* la valeur de l'item indiqué par *c*, dans le menu désigné par *u*.
Signification des arguments:

u (unité) Représente le menu concerné: 0 pour un menu de la barre de menu, l'identification d'une fenêtre pour le popup menu qui lui est associé ou encore l'identification d'un menu (popup ou pulldown).

c (chemin) Liste de chaînes indiquant un chemin de sélection dans le menu *u*. L'item correspondant dans ce chemin est remplacé par *v*. Si une telle hiérarchie n'existe pas, elle est créée. Si la liste est *nil*, c'est tout le menu qui est concerné. Attention: le titre d'un pulldown menu n'est pas un item, ne pas le mentionner dans le chemin.

v (valeur) Arbre de description de la valeur remplaçant l'item à la position indiquée par *c*, dans le menu *u*. Cette valeur peut être une feuille, ou une hiérarchie.

Valeur de type feuille

Une feuille est représentée par :

```
<"Nom item",Identificateur,n>
```

La chaîne *"Nom item"* vient remplacer le texte de l'item indiqué par le chemin. Ou bien, si la chaîne est vide, l'item désigné est supprimé. Un caractère '&' dans la chaîne *"Nom Item"* n'apparaitra pas dans le texte de l'item et désignera le caractère qui le suit, comme raccouci clavier pour activer l'item.
Identificateur est l'identificateur d'accès d'une règle d'arité 1 attachée à l'item.
n est un booléen indiquant si l'item est coché ou non.

Il est possible d'utiliser pour *v*, à la place de ce triplet, les formes simplifiées :

`nil` pour indiquer un trait de séparation horizontal
`Identificateur` pour `<"Identificateur",Identificateur,0>`
`<"Nom item",Identificateur>` pour
`<"Nom item",Identificateur,0>`

Lorsque le menu est sélectionné, le programme en cours est suspendu, et le but *Identificateur(u)* est immédiatement activé. Si le but se termine sans erreur, le programme reprend normalement, sinon l'erreur est propagé par le mécanisme *block_exit*.

Exemple, en supposant que la fenêtre d'édition "menu.p2" soit ouverte:

```
>
   set_menu( 0
          , ["Compilation","Compiler la fenêtre"]
          ,<"Compiler la fenêtre",compilation_fen>)
   set_menu( "menu.p2"
          , ["Compiler"]
          ,<"Compiler",compilation_fen>);
{}
>
insert;
compilation_fen(0) -> front_window(u) reinsert(u);
compilation_fen(u) -> string(u) reinsert(u);;
{}
>
```

Valeur de type hiérarchie

Une hiérarchie est représentée par un tuple à deux arguments de la forme:

```
<"Nom item", Liste_de_valeurs>
```

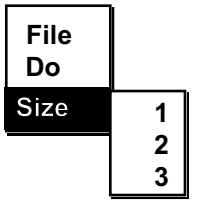
Chaque valeur représente elle-même une feuille ou une hiérarchie. Il est possible d'utiliser à la place de ce doublet une forme simplifiée:

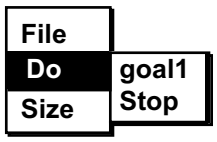
```
Liste_de_valeurs
```

Dans ce cas le nom de l'item est celui indiqué à la fin du chemin *c*. Ceci revient à apposer la hiérarchie sous l'item indiqué.
Exemple:

```
> set_menu("graphic",[],   [ <"File", [quit]>
                       , <"Do", [goal1,<"Stop",goal2>]>
                       , <"Size",  [ <"1",size1>
                                   , <"2",size2,1>
                                   , <"3",size3>
                                   ]>
                       ] );
insert;
size1(u) -> gr_window(u) gr_text(1,10,nil);
```

```
size2(u) -> gr_window(u) gr_text(1,12,nil);
...
```





Lorsque l'on utilise *set_menu* pour la barre de menu, c'est à dire avec *u* égal à 0, le menu créé sur la barre est un 'pulldown menu'. La première chaîne du chemin *c* ou bien, si *c* vaut *nil,* la première chaîne de la valeur *v*, représente le titre du pulldown menu. L'identification de ce menu est une chaîne: la chaîne de titre à la création du menu. Les modifications ultérieures du titre ne modifient pas l'identification du menu. L'identification du menu peut être utile pour d'autres primitives telles que *kill_object*, *get_attribute*…
Exemple:

```
>set_menu(0,"new".nil,<"colors",black.grey.white.nil>);
```

L'identification du menu créé par cet appel est "new", son titre est "colors".

```
>set_menu(0,nil,colors.black.grey.white.nil);
```

L'identification du menu créé par cet appel est "colors", son titre est "colors".

```
>set_menu(0,"colors"."black"."white"."else".nil,else);
```

L'identification du menu créé par cet appel est "colors", son titre est "colors".

Dans les primitives qui suivent, le premier et le deuxième argument suivent les mêmes conventions que pour *set_menu* :

check_item(u,c,v)

Cette primitive positionne ou enlève le cochage de l'item, suivant les valeurs de *v* : 0 indique non coché, 1 indique coché avec la marque standard de la machine.

- Si *v* est un entier plus grand que 1, le caractère de la machine hôte ayant le code *v* est utilisé comme marque.

Exemple:

```
> check_item(0,"colors"."black".nil,1);
```

- *command_menu(u,c,v)*

v est le code du caractère utilisé en conjonction de la touche de commande ( ), utilisée comme raccourci pour activer l'item spécifié par les deux premiers arguments.

Exemple de création du raccourci T pour activer la trace Prolog:

```
> char_code("T",n)
command_menu(0,"Contrôle"."trace".nil,n);
```

s Le caractère utilisé comme raccourci clavier est un caractère appartenant au texte de l'item du menu, il est désigné au moment de la définition du menu ou de l'item.

enable_menu(u,c,v)

Permet de désactiver (*v =0*) ou de réactiver (*v =1*) un menu ou un item. L'exemple suivant désactive tout le menu d'édition:

```
> enable_menu(0,"Editer".nil,0);
```

 style_menu(u,c,v)

Spécifie le style d'un item de menu. *v* est un entier dont chaque bit représente un des styles spécifiés avec l'ordre défini dans la primitive *gr_text*.
Exemple mettant en italique souligné l'item *trace* du menu *Fenêtres* (bit 1 + bit 2 positionnés) :

```
> style_menu(0,"Fenêtres"."trace".nil,6);
```

5.5. Mode de dessin et d'écriture

L'ensemble des primitives décrites ici, agit sur des propriétés de l'unité graphique courante (qui peut être une fenêtre de type GRAPHICS ou une 'drawing area').

gr_color3(r,v,b)

Définit la couleur du crayon de l'unité graphique courante en composantes *r* rouge, *v* vert, *b* bleu. Si les valeurs sont réelles, elles correspondent à des pourcentages d'intensité pour chaque composante, elles doivent être comprises entre 0e0 et 1e0, et sont portables d'une machine à l'autre. Si les valeurs sont entières, elles doivent être comprises entre 0 et 65535. Cette couleur est mémorisée comme la *:customColor* de l'unité graphique courante.

 gr_color(d,c)

Définit la couleur du fond (si *d*= 0) ou du crayon (si *d*=1) de l'unité graphique courante. L'argument *c* est un entier représentant une des 8 couleurs possibles parmi les valeurs: 33:noir, 30:blanc, 205:rouge, 341:vert, 409:bleu, 273:cyan, 137:magenta, 69:jaune.

s *gr_color1(d,c)*

Définit la couleur du fond (si *d*= 0) ou du crayon (si *d*=1) de l'unité graphique courante. L'argument *c* représente une couleur selon les mêmes conventions que celles de la primitive *gr_color2*.

gr_color2(f,c)

Définit la couleur *f* du fond et *c* du crayon de l'unité graphique courante. Les arguments *f* et *c* sont des entiers ou des identificateurs représentant une des couleurs possibles parmi les valeurs :

		couleur
0	:white	blanc
1	:red	rouge

2	:green	vert
3	:blue	bleu
4	:pink	rose
5	:orange	orange
6	:brown	marron
7	:magenta	magenta
8	:purple	violet
9	:cyan	cyan
10	:yellow	jaune
11	:lightGrey	gris clair
12	:darkGrey	gris foncé
13	:black	noir
14	:defaultColor	couleur système
15	:customColor	couleur utilisateur définie par *gr_color3*.

Remarque: la forme entière est moins explicite que la forme identificateur. Dans les primitives d'interrogation de la configuration (*get_attribute(ob1, :background, x)*), Prolog retourne les valeurs sous forme d'identificateur.

- *gr_choosecolor(p,s,i,f)*

  Affiche le dialogue de sélection de couleur au point *p* (de la forme <x,y>) de l'écran. *s* est une chaîne qui sera affichée dans le dialogue. *i* et *f* sont des triplets d'entiers compris entre 0 et 65535; un triplet <r, v, b> représentant une couleur en composantes r rouge, v vert, b bleu. *i* est la couleur initiale du dialogue et doit être connu au moment de l'appel, *f* sera unifié avec la couleur choisie. Echoue si l'utilisateur clique sur le bouton *Annuler*.

gr_pen(l,m)

- *gr_pen(l,h,m)*

  Définit la taille en pixels du crayon de l'unité graphique courante et son motif .
  l = largeur,
- *h* = hauteur,
  m = motif , est un entier ou un identificateur.

m	*m*	motif
0	:clear	trame du fond
1	:solid	trame pleine
2	:halfPattern	trame demi-teinte
3	:lightPattern	trame claire
4	:darkPattern	trame foncée

Remarque: la forme entière est moins explicite que la forme identificateur. Dans les primitives d'interrogation de la configuration (*gr_get_pen(l, h, m)*), Prolog retourne les valeurs sous forme d'identificateur.

s *gr_get_pen(l,m)*
s *gr_get_pen(l,h,m)*

Renseigne sur la taille et le motif du crayon de l'unité graphique courante. *l,h,m* ont la même signification que dans la primitive *gr_pen(l,h,m)*. Prolog retourne un identificateur pour le motif du crayon.

gr_stringwidth(s,n)

Unifie *n* avec la longueur en pixels de la chaîne *s* en fonction de la fonte, de la taille et du style définis pour l'unité graphique courante.

gr_mode(b,m)

Définit le mode d'application du motif sur le fond, pour l'unité courante graphique.

b=0 Concerne le dessin et donc le motif du crayon.

b=1 Concerne le texte et donc le motif correspondant à chaque caractère.

Il existe quatre opérations de base: Copy, Or, Xor et Bic.
L'opération Copy remplace simplement les pixels de destination par les pixels du motif ou de la source, dessinant sur la destination sans se préoccuper de son état initial.
Les opérations Or, Xor, et Bic laissent inchangés les pixels de destination situés sous la partie blanche du motif ou de la source. Elles traitent différemment les pixels situés sous la partie noire:

Or (ajout)	les remplace par des pixels noirs.
Xor (inversion)	les remplace par leur contraire.
Bic (effacement)	les remplace par des pixels blancs.

Suivant le mode, l'opération est réalisée à partir du motif lui-même (src), ou bien à partie de son inverse (notSrc).

Pour les machines couleur, la partie blanche du motif correspond à la partie couleur du fond, la partie noire du motif correspond à la partie couleur du crayon, la détermination de la couleur inverse est dépendante de la machine.

Voici les valeurs de *m* pour les différents modes.

m	*m*	modes d'application
0	:srcCopy	srcCopy
1	:srcOr	srcOr
2	:srcXor	srcXor
3	:srcBic	srcBic
4	:notSrcCopy	notSrcCopy
5	:notSrcOr	notSrcOr
6	:notSrcXor	notSrcXor
7	:notSrcBic	notSrcBic

Remarque: la forme entière est moins explicite que la forme identificateur. Dans les primitives d'interrogation de la configuration (*gr_get_mode(1, x)*), Prolog retourne les valeurs sous forme d'identificateur.

s *gr_get_mode(b,m)*

Renseigne sur le mode d'application du motif dans l'unité graphique courante. *b, m* ont la même signification que dans *gr_mode(b,m)*. Prolog retourne le mode comme un identificateur.

 gr_text(f,t,s)

Permet de définir la fonte, la taille et le style des sorties dans l'unité graphique courante. *f* = numéro de la fonte; t = taille de la fonte; *s* = liste de numéros de style.

N°	fonte	N°	style
0	systemFont	0	gras
1	applFont	1	italique
2	newYork	2	souligné
3	geneva	3	relief
4	monaco	4	ombré
5	venice	5	condensé
6	london	6	étalé
7	athens		
8	san Franscico		
...			

La liste vide représente le texte normal. La combinaison par défaut est *gr_text(4,12,nil)*.
Par exemple *gr_text(3,18,1.3.4.nil)* correspond à geneva taille 18, italique, relief et ombré.
Il est possible d'appeler cette règle en donnant en argument des identificateurs auxquels on a assigné les constantes appropriées :

```
> assign(Geneva,3) assign(Italic,1) assign(Relief,3);
{}
> gr_text(Geneva,18,Italic.Relief.nil)
 output("graphic") outm("Hello") output("console");
```

affiche *Hello* dans l'unité graphique.

s *gr_font(f)*

Charge la fonte représentée par *f* pour l'unité graphique courante. La fonte de base qui est la fonte proposée par le système est désignée par *:ftdefault* ; les autres fontes proposées sont de la forme*:ft(i)* ou bien *i*, où *i* est un entier compris entre 1 et 50. La correspondance entre l'entier et le nom de la fonte du système hôte est définie dans le fichier *fonts.usr*. La totalité des fontes disponibles sur la machine peut être examinée aprés un premier lancement de Prolog, dans le fichier *fonts.all* (créé au démarrage du graphisme).
Exemple:

```
> gr_font(:ft(2));
```

5.6. Dessin et positionnement

Les primitives décrites ci-dessous agissent sur l'unité graphique courante (qui peut être une fenêtre de type GRAPHICS ou une 'drawing area'). Elles permettent d'y définir un système de coordonnées, et d'y réaliser des dessins.

gr_setorigin(x,y)

Change l'origine des axes de sorte que *x,y* représentent les coordonnées du coin intérieur haut et gauche de l'unité graphique courante. L'aspect de l'unité n'est pas modifié.

s *gr_getorigin(x,y)*

Renseigne sur les coordonnées du coin intérieur supérieur gauche de l'unité graphique courante installées par *gr_setorigin*

gr_penloc(x,y)

Donne en coordonnées entières la position du crayon dans l'unité graphique courante.

gr_penlocr(x,y)

Identique à *gr_penloc(x,y)* mais donne un résultat de type réel.

gr_erase

Efface le contenu de l'unité graphique courante, repositionne le crayon en haut et à gauche et supprime tous les événements liés à la fenêtre et à ses objets.

gr_move(x,y)

Modifie la position du crayon de *x* pixels dans le sens horizontal et *y* pixels dans le sens vertical par rapport à la position courante.

gr_moveto(x,y)

Positionne le point d'écriture (crayon) de l'unité graphique courante en *x,y*.

gr_line(x,y)

Tire un trait depuis la position courante jusqu'à la nouvelle position du crayon, déplacé horizontalement de *x* pixels et verticalement de *y* pixels.

gr_lineto(x,y)

Tire un trait depuis la position courante jusqu'au point *x,y*, en déplaçant le crayon.

gr_rect(n,r)
gr_rect(n,x1,y1,x2,y2)

Permettent d'appeler les routines basiques de dessin inscrit dans un rectangle. *r* est l'arbre désignant le rectangle. *x1,y1* sont les coordonnées du coin supérieur gauche, et *x2,y2* celles du coin inférieur droit du rectangle.

Le premier argument *n*, où *n* est un entier ou un identificateur, détermine la procédure appelée:

0, :frameRect, :frame
Dessine le périmètre du rectangle.

1, :paintRect, :paint
Remplit le rectangle avec le motif associé au crayon.

2, :eraseRect, :erase
Efface le contenu et le périmètre du rectangle.

3, :invertRect, :invert
Inverse la couleur de chaque point de l'intérieur du rectangle. Si le rectangle a un coloriage autre que noir et blanc, le résultat est dépendant de la machine.

4, :frameOval
Dessine le périmètre de l'ovale inscrit dans le rectangle.

5, :paintOval
Remplit l'ovale avec le motif associé au crayon.

6, :eraseOval
Efface le contenu et le périmètre de l'ovale.

7, :invertOval
Inverse les bits de l'ovale. Si l'ovale a un coloriage autre que noir et blanc, le résultat est dépendant de la machine.

8, :frameRoundRect
Identique à *:frameRect* mais avec des coins arrondis.

9, :paintRoundRect
Identique à *:paintRect,* mais avec des coins arrondis.

10, :eraseRoundRect
Identique à *:eraseRect* mais avec des coins arrondis.

11, :invertRoundRect
Identique à *:invertRect* mais avec des coins arrondis.

12, :clipRect
Les dessins ultérieurs traceront uniquement ce qui est à l'intérieur du rectangle. Autrement dit, définit un rectangle de visibilité de tous les dessins à venir. Pour en annuler l'effet, le refaire avec les coordonnées de l'écran.

gr_polygon(n,L)

Dessine un polygone dans l'unité graphique courante. *L* est une liste de doublets <x,y> indiquant les coordonnées des points définissant le polygone.
Le premier argument *n*, où *n* est un entier ou un identificateur, détermine la procédure appelée:

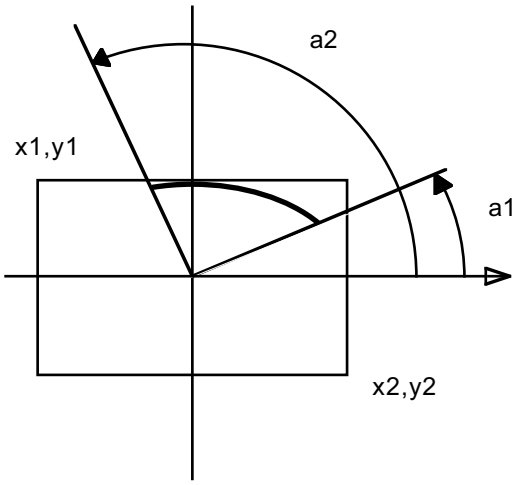
0, :frame
Dessine le périmètre du polygone.
1, :paint
Remplit le polygone avec le motif associé au crayon.
2, :erase
Efface le contenu du polygone (certaines des lignes brisées formant le contour du polygone peuvent subsister).
3, :invert
Inverse la couleur de chaque point de l'intérieur du polygone. Le résultat est dépendant de la machine si le polygone a un coloriage autre que noir et blanc.

gr_arc(n,r,a1,a2)
gr_arc(n,x1,y1,x2,y2,a1,a2)

Dessine un arc d'ellipse inscrit dans le rectangle *r* entre les angles *a1* et *a2*. Les angles *a1* et *a2* sont donnés en degrés.
Le premier argument *n*, où *n* est un entier ou un identificateur, détermine la procédure appelée:

0, :frame
Dessine l'arc d'ellipse.
1, :paint
Remplit le secteur défini par l'arc d'ellipse avec le motif associé au crayon.
2, :erase
Efface l'arc et le secteur.
3, :invert
Inverse la couleur de chaque point de l'intérieur du secteur d'ellipse. Le résultat est dépendant de la machine si le coloriage est autre que noir et blanc.

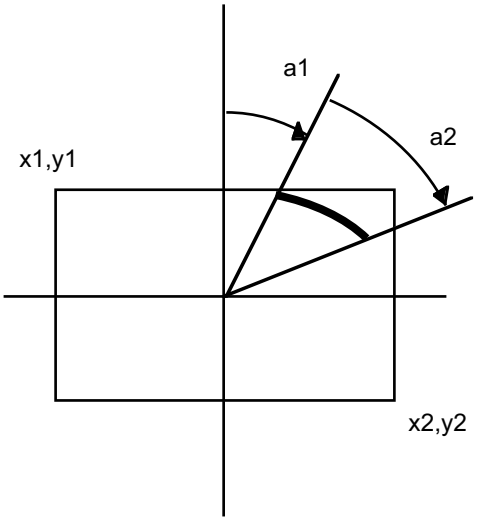
```
> gr_arc(:frame,100,100,200,170,20,110);
```



gr_arc'(n,r,a1,a2)
gr_arc'(n,x1,y1,x2,y2,a1,a2)

Dessine un arc d'ellipse inscrit dans le rectangle *r* entre les angles *a1* et *a2*. *a1* et *a2* sont donnés en degrés. Les arguments sont les mêmes que pour *gr_arc* mais l'origine est sur l'axe vertical, *a2* est relatif à la position *a1*, le sens positif est dans le sens des aiguilles d'une montre.

```
> gr_arc'(:frame,100,100,200,170,20,30);
```



- *gr_icon(n,x1,y1,x2,y2)*
- *gr_icon(n,r)*

Dessine dans l'unité graphique courante, et inscrit dans le rectangle indiqué une icône décrite dans une ressource 'ICON' de numéro *n*. Cette ressource est recherchée d'abord dans l'exécutable Prolog II+, puis dans le fichier système. Le système contient toujours les icônes :

0 Panneau stop.

1 Notification.

2 Panneau danger.

Le meilleur résultat est obtenu en donnant comme rectangle un carré de coté égal à 32.

gr_load(s)

Dessine dans l'unité graphique courante le contenu d'un fichier graphique de nom *s*. Ce fichier graphique est un fichier externe à PrologII+ réalisé à l'aide d'un quelconque éditeur graphique du système hôte. Il est possible de dessiner à des endroits différents des parties sélectionnées du fichier, en jouant sur les primitives *gr_setorigin* et *gr_rect(:clipRect,..)*.

5.7. Position de la souris dans une zone graphique

Un événement 'clic' peut être utilisé pour sélectionner, pour déclencher un traitement de manière asynchrone, ou encore, simplement pour désigner un point d'un objet de type GRAPHICS. Les primitives décrites ci-dessous concernent l'unité graphique courante (qui peut être une fenêtre de type GRAPHICS ou une 'drawing area').

 gr_click(b,x,y)

Teste si un bouton de la souris a été pressé depuis le dernier appel de *gr_erase* ou *gr_click*. Si c'est le cas, x et y sont unifiés avec les coordonnées, dans l'unité graphique courante, du point où se trouvait la souris lorsque cet événement, appelé "clic", s'est produit. Sinon agit selon *b*. *b* est un booléen qui doit être connu au moment de l'appel, avec la signification :

b=1 Attendre qu'un nouveau clic se produise dans l'unité graphique courante.

b=0 Ne pas attendre et réaliser un backtracking.

s *gr_click(b,x,y)*

Teste si un événement "clic" dans l'unité graphique courante est présent dans la queue des événements. Si c'est le cas, x et y sont unifiés avec les "coordonnées du clic" dans l'unité graphique courante et l'événement est supprimé de la queue. Sinon agit selon *b*. *b* est un booléen qui doit être connu au moment de l'appel, avec la signification :

b=1 Attendre qu'un nouveau clic se produise dans l'unité graphique courante.

b=0 Ne pas attendre et réaliser un backtracking.

Voir aussi la primitive *get_event*.

gr_click(b,x,y,m)

Identique à *gr_click(b,x,y)* mais donne en plus l'état des principales touches de modification en unifiant *m* avec la somme des valeurs :

1 La touche MAJ USC ULE est appuyée.
2 La touche CONTROLE[1] est appuyée.
 4 Une touche option est appuyée (OPTION sur Mac).
 8 La touche COMMANDE est appuyée.
 16 La touche MAJ USC ULES-VERRO UILLEES est enfoncée.
s 32 Il s'agit d'un DO UBLE_CLIC.

On a donc *m*=0 si aucune touche modificatrice n'est appuyée pendant le clic.

gr_clickr(b,x,y)

Identique à *gr_click(b,x,y)* mais *x* et *y* sont de type réel.

---

1 Cette touche est absente sur le clavier du Macintosh +.

s *gr_sensitive(x1,y1,x2,y2,p)*

Définit le rectangle de coordonnées *x1*,*y1*,*x2*,*y2* dans l'unité graphique courante *f*, comme zone sensible aux événements clic, à laquelle est attaché le prédicat *p*. Si *p* est un identificateur différent de *nil* et *get_event*, à chaque clic à l'intérieur du rectangle le but *p(f)* sera lancé. Par défaut toutes les zones sont définies avec *get_event* et donc tous les clics dans l'unité sont envoyés dans la queue d'événements de Prolog. *gr_sensitive(x,y,x',y',nil)* permet d'ignorer les clics dans la zone définie.
Les différents rectangles définis doivent être disjoints. Dans le cas contraire, pour un clic dans la partie commune, le comportement n'est pas garanti. C'est une des zones qui le capte, mais on ne sait pas laquelle.

gr_getmouse(x,y)
gr_getmouse(x,y,b)

Donne la position actuelle de la souris dans l'unité graphique courante en coordonnées entières relatives à l'origine de l'unité. *b*=0 si le(s) bouton(s) de la souris sont relevés au moment de l'appel, sinon le numéro du bouton pressé en commençant par la gauche pour une souris à plusieurs boutons (le premier bouton a le numéro 1).

gr_getmouser(x,y)

Identique à *gr_getmouse(x,y)* mais donne un résultat réel.

5.8. Primitives spéciales de saisie

5.8.1. Simulation de boutons

L'utilisation des deux primitives qui suivent nécessite le chargement préalable du module de dessin *dessin.mo*.

Elles permettent de simuler des boutons dans l'unité graphique courante.

gr_draw_buttons(b)

Dessine tous les boutons de la base de données dont l'identificateur d'accès est *b*. La base de données de boutons est construite par l'utilisateur. Il attribuera un nom de règle à chaque groupement de boutons. Les règles doivent avoir trois arguments:
- le rectangle dans lequel le bouton s'inscrit.
- la chaîne affichée dans le bouton.
- un terme à associer au bouton.

Par exemple les boutons suivants définis par l'utilisateur:

```
wtButton(<5,160>.<50,175>,"Edit",edit)->;
wtButton(<5,180>.<50,195>,"Set",nil)->;
```

sont dessinés en lançant le but:

```
> gr_draw_buttons(wtButton);
```

gr_button_hit(b,<x,y>,t)

b est l'identificateur d'accès à la base de données de boutons. Cette règle s'efface si *x,y* sont les coordonnées d'un point à l'intérieur d'un des boutons de la base *b*. *t* est alors unifié avec le troisième argument de la règle pour le bouton concerné.
La primitive *gr_click* combinée avec *gr_button_hit* permet d'écrire facilement une boucle de scrutation des boutons.

5.8.2. Affichage de message avec validation

s *message_box(t,s,m,b,x)*

Crée une fenêtre graphique modale pour afficher un message. Après saisie, la fenêtre est détruite et *x* est unifié avec le numéro du bouton de sortie. La fenêtre comporte:
- un titre donné par la chaîne de caractères *s*,
- une icone définie par l'identificateur *t* parmi les valeurs: *:warning*, *:info*, *:question*, :*wait*, *:error*, dont le dessin dépend de l'environnement graphique de la machine
- un message donné par la chaîne de caractères *m*,
- une ligne de boutons (au plus 3) telle que:
si *b* est un entier, il y aura *b* boutons dont les textes sont fonction de leur nombre et du type d'icone, et dépendent de l'environnement graphique de la machine.
si *b* est une liste de chaînes, chaque chaîne définit le texte d'un bouton. Ceci peut être sans effet sur certains environnements.

5.8.3. Saisie de texte

gr_editf(<s1,p1,p2>,r,s,k)
gr_editf(<s1,p1,p2>,x1,y1,x2,y2,s,k)

Créent un rectangle d'édition dans l'unité graphique courante, qui doit être une fenêtre. Dans le cas contraire (drawing area), une erreur est générée.
La fin d'édition est provoquée par un retour chariot, un caractère de tabulation, ou un clic dans la fenêtre en dehors du rectangle d'édition et en dehors d'un objet. Si *k* n'est pas une variable, le rectangle et son texte sont seulement dessinés sans édition.

s1 est la chaîne qui sert à initialiser l'édition.

p1, *p2* sont des entiers définissant les indices des caractères de début (inclus) et de fin (non inclus) de la sélection dans le texte (représentée en vidéo inverse). Le premier indice est 0. Initialiser avec un champ vide correspond aux valeurs:<"",0,0>.

r définit la position du rectangle englobant. Le texte est cadré à gauche.

x1,y1,x2,y2 sont les coordonnées des coins supérieur gauche et inférieur droit du rectangle dans la fenêtre.

s est la variable qui sera unifiée avec la chaîne éditée.

k si *k* est une variable, celle-ci est unifiée en sortie avec le mode de terminaison:

0 si retour chariot,
1 si clic souris dans la fenêtre[1] en dehors du rectangle d'édition,
2 si caractère TAB.

Si *k* est une constante en entrée, sortie immédiate après affichage de la zone.

Une forme plus simple de la règle est également utilisable pour une chaîne initiale vide:

```
gr_editf(<>,x1,y1,x2,y2,s,k)
```

s Remarque : cette fonction peut également être réalisée avec les primitives de création et manipulation des objets *editfield* ou *label*.

- *get_key(c)*
- *get_key(a,t,m)*

get_key sert à prendre un caractère au vol. Il n'y a aucun écho de ce caractère dans aucune fenêtre que ce soit. Cette primitive permet donc de saisir un mot-de-passe. Le fonctionnement détaillé est donné dans la description de *stty* .

```
get_key(c)
```

rend dans *c* un caractère.

```
get_key(a, t, m)
```

rend trois entiers :
- dans *a* le code ascii étendu
- dans *t* le numéro de touche (*virtual key code* décrit dans Inside Macintosh vol. V p.192))
- dans *m* l'état des touches de modifications (voir la primitive *gr_click*)

- *stty(" USEGETKEY", t)*

réglemente l'usage du prédicat *get_key*.

t = *0*

mode par défaut, dans lequel *get_key* échoue toujours. Les caractères tapés sont insérés dans les fenêtres texte le cas échéant. Dans ce mode, l'utilisateur peut éditer des fenêtres texte pendant que Prolog II+ travaille.

t < *0*

---

1 On peut lire sa position avec *gr_click(0,x,y)*.

get_key prendra un caractère qui ne peut être plus âgé que *t* ticks (1/60sec) avant l'appel. Echoue si, au moment de l'appel, aucun caractère n'est disponible, ou si le caractère est trop "*vieux*" .

t > *0*

get_key attendra un caractère pendant au plus *t* ticks. Echouera si aucun caractère n'est disponible au bout de ce laps de temps. *get_key* n'attend pas que le délai soit écoulé pour réussir, si on lui fournit un caractère dans les temps. Un caractère peut avoir été tapé à l'avance. Ce mode permet de saisir un mot-de-passe en un temps donné.

Exemples :

> *stty(" USEGETKEY", -60);*

Le prochain appel à *get_key* échouera si aucun caractère non lu n'a été tapé moins d'une seconde avant cet appel (1 s = 60 ticks).

> *stty(" USEGETKEY", 300);*

Le prochain appel à *get_key* échouera si aucun caractère non lu ne peut être obtenu dans les cinq secondes suivant cet appel (1 s = 60 ticks). Le caractère peut avoir été tapé à l'avance. *get_key* réussira de suite après avoir obtenu le caractère sans attendre que le délai soit écoulé.

- *stty("FL USH")*

  Enlève tous les caractères de la queue d'événements. Ils ne sont plus disponibles pour *get_key*. (permet à *get_key* de saisir une réponse qui n'a pas été tapée à l'avance.)

- *gtty(s, v)*

  Récupère la valeur d'un paramètre concernant le terminal :

  gtty(" USEGETKEY", v) *rend 0 ou un nombre signé de ticks.*

  gtty("TERM", x) *rend actuellement* "TTY" *ou* "MPW".

5.8.4. Boites de choix

gr_list(r, l, l_i, v_i, l_o, v_o, o)

La règle *gr_list* gère un choix d'élément(s) dans une liste. Cette liste s'affiche dans l'unité graphique courante, qui doit être une fenêtre. Dans le cas contraire (drawing area), une erreur est générée.

Description des paramètres :

r

Le rectangle englobant (ascenseurs compris) de la forme <x1,y1>.<x2,y2> ou <x1,y1,x2,y2>.

l

La liste d'items. Elle doit être une liste de constantes (éventuellement mixtes : chaîne, entier, réel, identificateur, <>).

l_i

Est la sélection initiale, sous la forme d'une liste éventuellement vide d'entiers qui sont les rangs (à partir de 1) dans *l* des éléments que l'on veut sélectionner.

v_i

C'est le rang de l'élément qui doit apparaître en première position en haut du rectangle.

l_o

Donne en sortie la liste des rangs des items sélectionnés (même convention que pour *l_i*).

v_o

Donne en sortie le rang du premier item visible dans le rectangle.

o

Décrit l'opération à réaliser parmi 4 possibles :

<0> Dessine seulement (pas de saisie possible).

<1,k> Dessine et prend la main jusqu'à ce qu'un événement de type frappe de caractère ou clic dans la fenêtre à l'extérieur de la liste ne se produise. NI LE CLIC, NI LE CARACTERE NE SONT L US. Rend pour k :

0 : Retour Chariot tapé
1 : clic en dehors de la liste
2 : TAB tapé

<2,x,y,m>

Dessine la liste et traite le clic x,y,m (donné par exemple par gr_click(?,x,y,m)). Sort tout de suite après.

<3,x,y,m,k>

Dessine, traite le clic x,y,m (donné par exemple par gr_click(?,x,y,m)) et prend la main jusqu'à ce qu'un événement de type clavier ou un clic dans la fenêtre à l'extérieur du rectangle se produise. Ni le clic ni le caractère ne sont lus.

Rend pour k :

0 : Retour Chariot tapé
1 : clic en dehors de la liste
2 : TAB tapé

On peut lire le clic de fin de sélection par *gr_click(0,x,y)*.

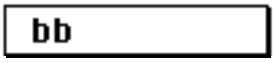
s Remarque : cette fonction peut également être réalisée avec les primitives de création et manipulation des objets *listbox*.

- *gr_popupItem(r,L,n1)*
- *gr_popupItem(r,L,n1,n2)*

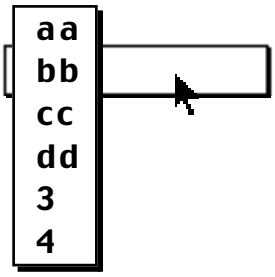
Crée un champ à valeurs proposées.
L est une liste de constantes. *r* est le rectangle dans lequel est affiché l'item numéro *n1* de la liste *L*. Si *n2* est absent, l'item est simplement dessiné. Si *n2* est présent, un menu présentant la liste des choix *L* vient recouvrir l'item, et l'inversion des champs est gérée tant que le bouton de la souris reste appuyé. Lorsque le bouton est relaché, le champ sélectionné est dessiné dans le rectangle à la place du précédent, et *n2* est unifié avec son numéro (*n1* si aucun champ n'est sélectionné). Exemple: rendre la fenêtre graphique visible et taper dans la console l'exemple suivant :

```
> gr_popupItem(<20,20,120,40>, ["aa","bb","cc","dd",3,4], 2)
```

```
gr_click(1,x,y)
gr_popupItem(<20,20,120,40>, ["aa","bb","cc","dd",3,4], 2,
i);
```



Lorsque vous cliquez dans la fenêtre en maintenant le bouton appuyé :



5.8.5. Choix de fichier

input(x)
output(x)

C'est une facilité ajoutée aux primitives *input* et *output*, avec *x* variable libre au moment de l'appel. Cette facilité consiste à demander à l'utilisateur, le nom du fichier par une boite de dialogue. Ensuite le changement d'unité d'entrée ou de sortie est réalisé normalement (voir *input output*), avec ouverture éventuelle du fichier. En sortie, unifie *x* avec le nom complet du fichier. Il y a échec et aucune action réalisée si, dans le dialogue de sélection de fichier, le bouton *Annuler* est cliqué.

sfgetfile(s)

s *sfgetfile(s,Filter,NomInitial,Point)*

Affiche une zone de dialogue permettant de visualiser, parmi les noms des fichiers se trouvant sur le(s) disque(s), ceux compatibles avec *Filter* et d'en sélectionner un.
Point est un point de format <x,y> indiquant l'endroit où est affiché le dialogue.
Ce dialogue comporte une liste de choix pour les fichiers proposés, et un champ d'édition initialisé avec la chaîne *NomInitial*.
s est unifié avec la sélection faite par l'utilisateur, il s'agit du nom complet du fichier (comprend le chemin d'accès et le nom selectionné). Le fichier n'est pas ouvert. La règle échoue si le bouton *Annuler* est pressé.

 sfgetfile(Point,Ltypes,Filter,S)

Affiche une zone de dialogue permettant de visualiser, parmi les fichiers se trouvant sur le(s) disque(s), les noms de ceux dont le type est mentionné dans la liste *Ltypes* et d'en sélectionner un.

Point est un point de format <x,y> indiquant l'endroit où est affiché le dialogue.
Ltypes est une liste de types précisant les fichiers à proposer. Un type est une chaîne de 4 caractères tel que défini dans *Inside Macintosh*. Exemple: *"TEXT"."APPL".nil* sélectionne tous les fichiers de type texte seul ou application.
Filter est une procédure de filtrage pour l'acceptation des fichiers sélectionnés. Ce doit être *nil* pour l'instant.
S est unifié avec la sélection faite par l'utilisateur. Il est de la forme : <C,T,N> où C est le créateur (chaîne de 4 caractères), T est le type du fichier (chaîne de 4 caractères), et N est une chaîne indiquant le chemin d'accès et le nom du fichier sélectionné (le fichier n'est pas ouvert).
La règle échoue si le bouton *Annuler* est pressé.

sfputfile(s)

Affiche une zone de dialogue permettant de choisir un nom de fichier à créer. Unifie *s* avec le nom avec chemin de ce fichier. Le fichier n'est pas ouvert. La règle échoue si le bouton *Annuler* est pressé.

- *sfputfile(Point,Prompt,NomInitial,s)*

Affiche une zone de dialogue permettant à l'utilisateur de choisir un nom, un volume, et un répertoire pour un fichier.
Le nom du fichier se saisit au clavier dans une zone d'édition qui est initialisée avec la chaîne (éventuellement vide) *NomInitial* qui doit être un nom de fichier sans chemin.
Si l'utilisateur tape un nom de fichier déjà existant, une alerte le prévient. Le fichier n'est ni créé ni ouvert.
s est unifié avec le nom avec chemin validé par l'utilisateur.
Point est un point de format <x,y> indiquant l'endroit où est affiché le dialogue.
Prompt est une chaîne (éventuellement vide) contenant un message destiné à l'utilisateur, ajouté en tête du dialogue.
La règle échoue si le bouton *Annuler* est pressé.

5.8.6. Choix d'un noeud d'un arbre

gr_tree_click(a,l,n)

L'utilisation de cette primitive nécessite le chargement préalable du module de dessin *dessin.mo*
Dessine dans l'unité graphique courante l'arbre *a*, et attend que l'utilisateur clique sur un des noeuds de l'arbre. Lorsque ceci est fait, *l* est unifié avec le chemin conduisant au noeud, et *n* est unifié avec le noeud. Le chemin est la liste des numéros des fils à parcourir en partant de la racine. Par exemple, le noeud dd de l'arbre *aa(bb,cc(ee(dd),ff))* est représenté par le chemin *2.1.1.nil*.

5.9. Règles pour gérer des objets structurés

gendialog(D,l1,l2)
gendialog(P,D,l1,l2)

L'utilisation de cette primitive nécessite le chargement préalable du module *obdialog.mo*. Cette primitive donne à l'utilisateur un moyen trés rapide de créer des sources d'écrans graphiques, pouvant être ensuite intégrés à des applications.
P, *D* et *l1* ont la même signification que dans la primitive *gr_dialog* décrite dans cette section.
Génère dans *l2* une liste de buts permettant de créer l'écran de dialogue défini par *D*.
Exemple :

```
gendialog(<100,100>.3.nil,"Hello".buttonD,nil, b_uts) b_uts;
```

va unifier *b_uts* avec la liste des buts à effectuer pour la création de ce dialogue et ensuite effacer ces buts.

gr_dialog(D,l1,l2)
gr_dialog(P,D,l1,l2)

L'utilisation de cette primitive nécessite le chargement préalable du module *obdialog.mo*. Ce module contient le gestionnaire de dialogue Prolog II+ permettant de créer et activer les dialogues décrits par l'argument *D*.
Dans sa deuxième forme, la primitive *gr_dialog* accepte un argument supplémentaire *P*, qui est une liste formée au choix par :

- un doublet d'entiers <x,y> pour déterminer la position du coin supérieur gauche du dialogue;
- un entier *i* pour choisir la fonte du texte du dialogue. *i* est comme précisé dans la primitive *gr_font*.
- le terme *:title(s)* où *s* est une chaîne de caractères désignant le titre de la fenêtre de dialogue.

D est soit un arbre décrivant un dialogue, soit l'identificateur d'une règle unaire avec un tel arbre en argument. *l1* est une liste (éventuellement vide) de paires

```
(nom_de_zone . valeur_initiale)
```

cette liste est utilisée pour définir ou redéfinir le contenu des zones. En sortie, si *l2* est une variable, elle est unifiée avec la liste des couples (*nom_de_zone* . *valeur_finale*) du dialogue. Si *l2* est une liste de paires (*nom_de_zone* . *X*), chaque paire est unifiée avec le couple correspondant.
Un dialogue est décrit avec des objets primitifs (zone de texte, zone d'édition, boutons, ...) et des primitives d'agencement en colonne ou en rangée. Ces primitives réalisent un ajustement dynamique de la taille des zones en fonction de leur contenu. Les zones dont la valeur peut être modifiée sont identifiées par un nom, ce qui permet de redéfinir leur valeur au moment de l'appel.

Exemple:

```
> load("obdialog.mo");
> gr_dialog("Hello".buttonD("OK"),nil,L);
```


```
> gr_dialog(      "Name:"
                .editf(10,field1)
                .buttonD("OK")
              , (field1."John").nil, L);
```

Les primitives de description d'objet sont les suivantes:

:text(s) , :text(s,f), :text(s,f,c) ou *s*

Décrit une ligne de texte non éditable représenté par *s* où *s* est une chaîne.*f* désigne le numéro de fonte à la manière du prédicat *gr_font*, et *c* la couleur du texte à la manière de l'attribut *:foreground* du prédicat *set_attribute.*

:text(i)

Décrit une ligne de texte non éditable dont le nom de zone est *i* où *i* est un identificateur. Si une valeur est associée à *i* dans *l1* c'est cette valeur qui est la valeur initiale du champ, sinon c'est la chaîne vide. Il n'est jamais associé de valeur finale à cette zone, puisqu'elle n'est pas modifiable.

:cb(i), :cb(i,v), :cb(i,v,s)

Décrit une boîte, dont le nom de zone est *i*, de type choix oui/non qui sont représentés respectivement par les deux valeurs possibles de *v*: 1/0. Si la valeur de *v* n'est fournie ni en argument, ni dans la liste *l1*, c'est la valeur 0 qui est donnée par défaut. Lorsque *v* est redéfini dans *l1*, c'est toujours cette dernière valeur qui est prise en compte. *s* est une chaîne qui définit le texte qui accompagne la boite.

:rb(<g,i>), :rb(<g,i>,v), :rb(<g,i>,v,s)

Décrit un radio-bouton *i* d'un groupe *g* (où *g* est un identificateur, et *i* une constante quelconque). Ce radio-bouton de valeur 0/1, fait partie du groupe *g*, avec au plus une valeur possible à 1 dans le groupe. Si la valeur de *v* n'est fournie ni en argument, ni dans la liste *l1*, c'est la valeur 0 qui est prise par défaut. Lorsque *v* est redéfini dans *l1*, c'est toujours cette dernière valeur qui est prise en compte. *s* est une chaîne qui définit le texte qui accompagne le radio-bouton. En sortie, *nom_de_zone* est représenté par *g(i)*, et *valeur_finale* par 0 ou 1, et ceci pour chacun des radio-boutons du groupe.

:rb1(<g,i>), :rb1(<g,i>,v), :rb1(<g,i>,v,s)

Sensiblement identique à *rb* mais avec la contrainte que les éléments du groupe ne peuvent être tous à zéro. Seul l'élément ayant la valeur 1 figure dans la liste de sortie. *nom_de_zone* est alors représenté par *g*, et *valeur_finale* par *i*. Ce format de sortie est donc différent et plus simple que celui fourni par *rb*.

:editf(n,i), :editf(n,i,s)

Décrit une zone d'une ligne de texte éditable de nom *i*, où *i* est un identificateur. *s* représente la valeur initiale de la zone d'édition. Cette valeur initiale est la chaîne vide si elle n'est spécifiée ni en argument, ni dans la liste *l1*. *n* est un entier représentant le nombre minimum de caractères de la zone. Si *s* est plus grand, la zone est dynamiquement expansée.

:button(s,i)

Décrit un bouton avec le label *s* auquel est associé l'action *i*. Lorsque le bouton est pressé, une action est réalisée:

- Si *i* est *fail*, le dialogue est terminé, et *gr_dialog* échoue.
- Si *i* est *nil*, rien n'est fait.
- Sinon, le but *i(l)* est activé, *l* étant la liste des paires *nom.valeur* des zones du dialogue au moment où le bouton est cliqué.

Lorsque le but *i* a été exécuté, la gestion du dialogue reprend en l'état. Pour arrêter le dialogue en cours, l'utilisateur peut programmer dans *i* un *block_exit(<fail,L>)* qui provoquera un backtracking de *gr_dialog*, ou un *block_exit(<nil,L>)* qui conduira à une terminaison normale, *l2* étant construite à partir de la liste des couples *nom.valeur* indiquée par *L*.

:button2(s,i)

Cet item a le même comportement que l'item *:button*, mais cette fois-ci la règle *i* est appelée avec *deux* arguments. Le premier est une liste décrivant l'état des items du dialogue. Le deuxième est à l'appel une variable qui doit être instanciée par la règle *i/2* avec la liste des items dont la valeur est modifiée. Cette règle est écrite par l'utilisateur, et permet donc d'attacher une sémantique quelconque aux boutons. La règle définit une relation entre le premier argument (argument d'entrée représentant la liste des valeurs actuelles), et le deuxième argument (argument de sortie représentant la liste des nouvelles valeurs).

:buttonD, :buttonD(s), :buttonD(s,i)

Décrit un bouton de terminaison de dialogue avec le label *s* auquel est associé l'action *i*. Usuellement *s* est la chaîne "OK", et *i* est *nil*. Ce bouton est activable par un clic ou par la frappe d'un Retour Chariot.

:glist(Lignes,Colonnes,Identificateur,ListeValeurs)

Définit un item de type liste de valeurs avec barre de défilement. La signification des arguments est :

Lignes

Entier représentant le nombre d'items devant être visibles. La hauteur du rectangle est calculée automatiquement en fonction de ce paramètre.

Colonnes

Entier représentant le nombre de caractères devant être visible pour un item. La largeur du rectangle est calculée automatiquement en fonction de ce paramètre.

Identificateur

Identificateur servant à donner un nom interne au gestionnaire. Il est utilisé pour donner le résultat en sortie.

ListeValeurs

Liste quelconque de constantes (chaînes, identificateurs, nombres) qui représente la liste des valeurs présentées, ou bien un triplet décrivant l'état initial du gestionnaire. Ce triplet doit alors avoir la forme :
<ListeValeurs, ListeDesNoSelectionnés, NoEnHaut>

Le premier argument est une liste quelconque de constantes (chaînes, identificateurs, nombres).

Le deuxième argument est la liste ordonnée des numéros d'ordre des items composant la sélection initiale. Cette liste peut comporter 0 (pas de sélection), 1, ou plusieurs éléments.

Le troisième argument indique le numéro de l'item à présenter en haut du rectangle d'affichage.

En sortie, donc dans *l2*, le résultat est donné sous la forme habituelle d'un couple :

(*identificateur.listeDesItemsSélectionnés*)

Tandis que dans la représentation interne, l'état du gestionnaire est désigné par le triplet décrit ci-dessus. C'est sous cette forme que doit être manipulée la valeur de la glist dans les programmes liés aux boutons. C'est à dire c'est le couple (*identificateur . < ListeValeurs, ListeDesNoSelectionnés, NoEnHaut >*) qui doit apparaître dans les listes manipulées par ces programmes.

Les primitives d'agencement sont basées sur la notion de combinaison de rectangles: mettre deux rectangles en colonne définit un nouveau rectangle englobant les deux autres.

:col(l) ou *l*

Décrit une colonne alignée à gauche, dont le contenu est décrit par *l*. Si *l* est une séquence, la primitive calcule la taille de chaque élément de la séquence, et constitue une colonne d'éléments disposés les uns au dessus des autres, et alignés sur leur côté gauche.

:ccol(l)

Décrit une colonne centrée verticalement, dont le contenu est décrit par la séquence *l*.

:row(l)

Décrit une rangée horizontale décrite par la (séquence de) primitive(s) *l*, et dont les éléments sont alignés sur leur bord supérieur: si *l* est une séquence, la primitive calcule la taille de chaque élément de la séquence, et constitue une rangée d'éléments disposés côte à côte alignés sur leur bord supérieur.

:crow(l)

Décrit une rangée d'objets décrits par *l* et centrés horizontalement, c'est à dire disposés de manière à ce que leurs points médians soient tous alignés sur une même horizontale.

:vfill(n)

Décrit un espacement vertical de *n* pixels, où *n* est un entier.

:hfill(n)

Décrit un espacement horizontal de *n* pixels, où *n* est un entier.

La primitive de groupement suivante permet de gérer l'activation ou la désactivation de zones entières pendant la saisie du dialogue.

:group(identificateur.valeur,DescriptionDesItems)

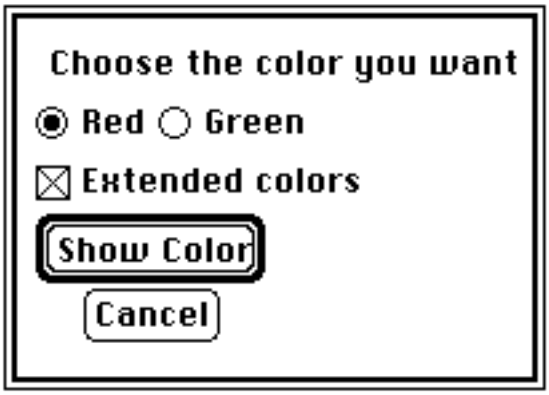
Permet d'associer un identificateur à un ensemble d'items regroupés géographiquement et sémantiquement. *valeur* indique si le groupe est actif ou non (0 pour inactif, 1 pour actif). Lorsque le groupe est inactif, il est dessiné en grisé et l'état de ses items ne peut être modifié. L'état d'un groupe ne peut être modifié dans la version actuelle que par les actions associées aux boutons. L'état d'un groupe n'est pas donné dans la liste de sortie, il est par contre fourni, dans la liste des états passée aux boutons, sous la forme d'un couple :

```
(identificateur.valeur)
```

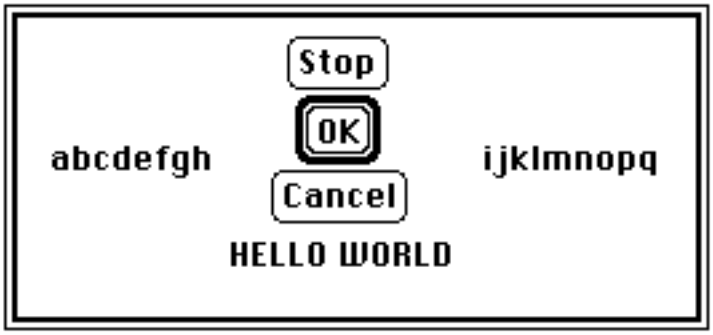
Exemples:

```
> gr_dialog( ccol("Voulez-vous recommencer le test?"
             .buttonD
             .button("Annuler",fail))
             , nil
             , L);
```


```
> gr_dialog( "Choose the color you want"
             .crow( rb(rb_1(red),1,"Red").
                    rb(rb_1(green),0,"Green"))
             .crow( cb(cb_1,0,"Extended colors"))
             .ccol( buttonD("Show Color")
                   .button("Cancel",fail)
                  )
           , (cb_1.1).nil
           , (cb_1.C).(rb_1(red).R).(rb_1(green).G).nil );
```

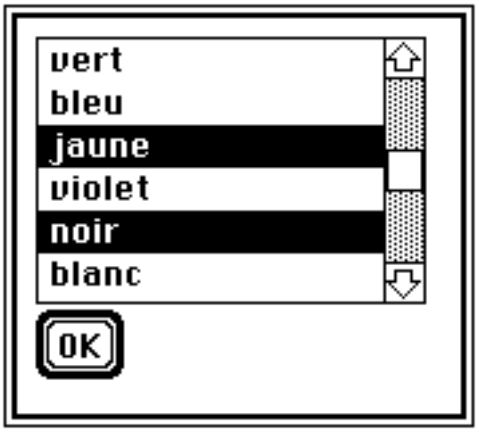



```
{C=1, R=1, G=0}
> gr_dialog( :crow( "abcdefgh"
                  .:ccol( :button("Stop",fail)
                         .:buttonD("OK",nil)
                         .:button("Cancel",fail)
                         ."HELLO WORLD"
                        )
                   ."ijklmnopq"
                )
             , nil,1);
```




```
> gr_dialog( glist( 5, 10, list1
                  , < [rouge,vert,bleu,jaune
                      ,violet,noir,blanc,orange]
                    , [4,6]
                    , 2
```

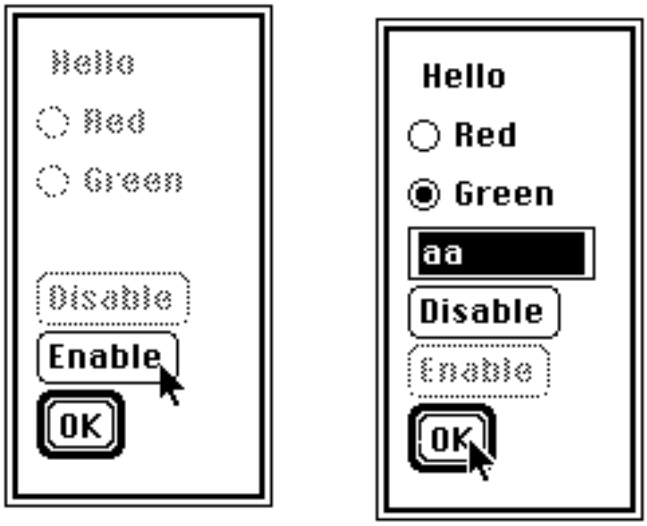
```
                        >)
                  .buttonD
               , nil
               , L);
{L=(list1.jaune.noir.nil).nil}
```




```
> insert;
enable( l1, l3 ) ->
   set_item(l1, zone1, 1, l2)
   set_item(l2, zone2, 0, l3);

disable( l1, l3 ) ->
   set_item(l1, zone1, 0, l2)
   set_item(l2, zone2, 1, l3);

set_item( nil, i, b, nil ) ->  block_exit(816, "set_item,
absent: ".i);
set_item( (i._).l, i, b, (i.b).l ) -> ! ;
set_item(e.l, i, b,  e.l' ) -> set_item( l, i, b, l' ) ;;
{}
> gr_dialog( group(zone1.0, "Hello"
                              .crow(rb1(color(red),0,"Red"))
                              .crow(rb1(color(green),1,"Green"))
                              .editf(5,ed1,"aa")
                              .button2("Disable",disable)
                              )
             .group(zone2.1,  button2("Enable",enable))
                               .buttonD
             , nil
             , L);
```




```
{L=(color.green).(ed1."aa").nil}
```

5.10. Envoi d'événements à Prolog depuis un objet externe

Nous appellerons "objet externe", tout objet graphique créé d'une autre manière que par l'appel des prédicats graphiques. Cela peut être fait durant l'appel à un prédicat externe ou bien dans le programme principal avant de démarrer Prolog, par exemple en chargeant un objet décrit dans un fichier de ressources généré par un outil externe (par exemple Dialog Editor sous Windows).

La possibilité qui est offerte ici, est de faire communiquer cet objet avec Prolog, en utilisant le principe des événements.

par l'intermédiaire de l'appel à une fonction C:

send_external_event(no,str,bool,int1,int2)

envoie un événement à Prolog.
Les deux modes de gestion de l'événement sont possibles: traiter l'événement immédiatement, ou bien le mettre dans la queue d'événements.
La signification des arguments est la suivante:
bool, qui est de type *int*, à valeur booléenne, indique le mode de traitement désiré:

- si *bool* est non nul, on désire traiter l'événement immédiatement, et dans ce cas:
  str, qui est de type *char* *, représente le but Prolog à appeler,
  no, *int1*, *int2*, sont ignorés.

- si *bool* vaut 0, on veut poster l'événement dans la queue de Prolog et dans ce cas:
  no, qui est de type *int*, pourra servir à identifier l'objet,

int1, *int2*, qui sont de type *int*, et *str* ont des valeurs dont les significations sont laissées libres à l'utilisateur.

Cet évenement sera accessible par les prédicats de lecture d'événements:
get_event(b, e, o) et *peek_event(b, e, o)* où :
o sera unifié avec *no*,
e sera unifié avec le terme Prolog *:extern_event(str, int1, int2)*

Index

^ R 1 - 23
^ R 2 - 4
~ R 4 - 6
^ R 10 - 3
* R 1 - 23
* R 10 - 3
** R 1 - 23
** R 4 - 6
** R 10 - 3
+ R 1 - 23
+ R 10 - 2; 3
, R 10 - 2; 3
- R 1 - 23
- R 10 - 2; 3
--> R 10 - 2
-> R 10 - 2; 3
/ R 1 - 23
/ R 10 - 3
// R 10 - 3; 7
/\ R 1 - 23
/\ R 4 - 6
/\ R 10 - 2
:- R 10 - 2
< R 1 - 23
< R 10 - 2; 7
<< R 1 - 23
<< R 4 - 6
<< R 10 - 3
= R 10 - 2; 3
=.. R 10 - 2; 8
=:= R 1 - 23
=:= R 10 - 2; 7
=< R 1 - 23
=< R 10 - 2; 7
== R 10 - 2; 11
=\= R 4 - 4
=\= R 10 - 23; 2; 7
> R 1 - 23
> R 10 - 2; 7
>= R 1 - 23
>= R 10 - 2; 7
>> R 1 - 23
>> R 4 - 6
>> R 10 - 3
?- R 10 - 2
@< R 10 - 2; 12
@=< R 10 - 2; 12
@> R 10 - 2; 12
@>= R 10 - 2; 12
abolish R 10 - 5
abs R 4 - 5
activity U 5 - 17
add R 4 - 4
add_implicit R 3 - 12
add_sentence_terminator R 5 - 7
add_stdmenu U 5 - 22
add_tlv R 2 - 14
alias R 10 - 13
alloc R 6 - 23
append R 5 - 2
arg R 4 - 11
arg R 10 - 8
arg2 R 4 - 10
assert R 2 - 11; 12
assert R 3 - 21
assert R 10 - 5

assert U 2 - 7
assert" R 3 - 22
asserta R 3 - 21
asserta R 10 - 5
assertn R 3 - 22
assertz R 3 - 22
assertz R 10 - 5
assign R 4 - 7
assign U 2 - 7
at R 10 - 14
atan R 4 - 6
atom R 10 - 6
atomic R 10 - 6
atom_chars R 10 - 8
atom_codes R 10 - 9
atom_concat R 10 - 9
atom_length R 10 - 9
attachement U 5 - 3
at_end_of_stream R 10 - 15
background U 5 - 17
backtrack_term R 4 - 8
bagof R 2 - 4
beep R 5 - 9
binary R 5 - 1
block R 2 - 4; 5
block U 2 - 7
block_exit R 2 - 4; 5
block_exit R 6 - 11
block_exit U 2 - 8
bottom_attach U 5 - 13
bottom_right U 5 - 17
bound R 2 - 6
bound R 4 - 2
bounded R 10 - 19
buttonD U 5 - 14
call R 10 - 4
callC R 7 - 26
callpro.def U 1 - 6
callpro.h U 1 - 6
callprol.lib U 1 - 6
callpros.lib U 1 - 6
cassign R 4 - 7
catch R 10 - 4
ceiling R 4 - 5
char U 5 - 20
chars_nb U 5 - 15
CHAR_ARRAY R 7 - 18; 21
char_code R 4 - 9
char_conversion R 10 - 19
check button U 5 - 4
check_button U 5 - 17
check_item U 5 - 24
chrono R 6 - 4
clause R 10 - 5
clear_events U 5 - 21
clear_input R 5 - 8
clear_menubar U 5 - 21
clear_window U 5 - 10
click U 5 - 20
click_down U 5 - 20
click_up U 5 - 20
close R 10 - 13
close_context_dictionary R 3 - 12
close_input R 5 - 8
close_output R 5 - 13
code R 6 - 25; 26
command_menu U 5 - 24
compound R 10 - 6
conc_list_string R 4 - 11
conc_string R 4 - 10
ConnectDescriptors R 8 - 10
ConnectInString R 8 - 10
ConnectOutString R 8 - 10
console R 5 - 1

consult R 10 - 5
copy_term R 4 - 11
copy_term_with_constraints R 4 - 11
cos R 4 - 6
coupure “!” R 2 - 2
cpu_time R 6 - 4
create_window U 5 - 9
cross U 5 - 17
current_context R 3 - 12
current_file R 5 - 2
current_input R 10 - 15
current_op R 10 - 3
current_output R 10 - 17
current_predicate R 3 - 22
current_prolog_flag R 10 - 20
cursor U 5 - 17
customColor U 5 - 18
customiz.dll U 1 - 3
customiz.dll U 3 - 2
customiz.dl_ U 1 - 3
customiz.dl_ U 3 - 3
C_FUNCTION R 7 - 18; 23
C_FUNCTION_BACKTRACK R 7 - 18; 23
C_FUNCTION_BACKTRACK_PROTECTED R 7 - 18; 23
C_FUNCTION_PROTECTED R 7 - 18; 23
date R 6 - 4
date_string R 6 - 4
date_stringF R 6 - 4
dbgbase.mo R 2 - 6
dbgbase.mo U 1 - 4
dbgedin.mo R 2 - 6
dbgedin.mo U 1 - 4
dbggraph.mo R 2 - 6
dbggraph.mo U 1 - 4
dde.mo U 1 - 3
ddeAcceptClient U 4 - 12
ddeCbClientRequest U 4 - 13
ddeCbClientWantsServer U 4 - 12
ddeCbServerClosing U 4 - 10
ddeCbServerUpdate U 4 - 10
ddeCleanup U 4 - 8
ddeDeclareAsClient U 4 - 9
ddeDeclareAsServer U 4 - 11
ddeEndConversation U 4 - 9; 12
ddeExecute U 4 - 10
ddeHotLinkData U 4 - 9
ddePokeData U 4 - 10
ddePublishExternalEvent U 4 - 4
ddeRememberDatabase U 4 - 8
ddeRequestData U 4 - 9
ddeTransmitData U 4 - 12
ddeTransmitRequest U 4 - 9
ddeWarmLinkData U 4 - 9
debug R 3 - 25
debug R 6 - 17; 24
debug R 10 - 19
default R 2 - 8
default U 5 - 17
def_array R 4 - 8
delay R 6 - 4
descripteur R 7 - 18
Dessin R 5 - 14
dessin.m2 R 5 - 14
dessin.m2 U 1 - 3
dessin.mo U 1 - 3
dictionary R 3 - 13
dictionary R 3 - 25
dictionary R 6 - 25; 26
dif R 2 - 7
DIRECT_C_FUNCTION R 7 - 18; 26
discontiguous R 3 - 23
discontiguous R 3 - 28
DISTANCE U 5 - 8

distance U 5 - 8
div R 4 - 4
dot R 4 - 1
double R 4 - 1; 5
DOUBLE_ARRAY R 7 - 18; 21
double_quotes R 10 - 19
Drawing area U 5 - 4
drawing_area U 5 - 17
draw_equ R 5 - 11
draw_mode R 5 - 12
draw_tree R 5 - 12
dynamic R 3 - 23
echo R 5 - 13
edinburg.mo R 6 - 24
edinburg.mo U 1 - 3; 4; 8
edinburg.mo U 2 - 5
edinburgh R 6 - 24
EDIT U 5 - 3; 7
edit R 3 - 27
edit R 6 - 3; 4
edit U 5 - 7; 17
Editfield U 5 - 4
editm R 3 - 14; 27
editm R 6 - 3
edit_area R 5 - 2
edit_field U 5 - 17
enable_menu U 5 - 25
end_module R 3 - 15
end_of_file R 10 - 15; 16
end_of_stream R 10 - 14
end_screen U 5 - 6
ensure_loaded R 3 - 23
enum R 4 - 12
eof R 5 - 3
eof_action R 10 - 13; 14
eof_code R 10 - 13
eol R 5 - 3
eq R 2 - 8
eql R 4 - 4
equations R 2 - 12
equations R 3 - 22
err.txt R 6 - 5
err.txt U 1 - 2
err.txt U 2 - 10
err.txt. U 1 - 8
ERROR R 8 - 2
error R 10 - 13
exit R 3 - 34
exit R 6 - 1
exit U 2 - 2
exp R 4 - 6
extern_event U 5 - 21; 49
fail R 2 - 8
fail_if R 10 - 4
fasserta R 3 - 23
fassertz R 3 - 23
fgetargtype R 7 - 3
fgetdouble R 7 - 5
fgetinteger R 7 - 5
fgetmaxstring R 7 - 5
fgetreal R 7 - 5
fgetstring R 7 - 5
fgetstrterm R 7 - 9
fgetterm R 7 - 17
file_dictionary R 3 - 13
file_name R 10 - 13
file_window U 5 - 10
findall R 2 - 8
find_pattern R 4 - 10
float R 4 - 5
float R 10 - 6
floor R 4 - 5
flush R 5 - 9
flush_output R 10 - 17

focus U 5 - 18
focus_in U 5 - 21
focus_out U 5 - 21
FONT U 5 - 8
font U 5 - 8; 18
fontheight U 5 - 17
fonts.all U 5 - 28
fonts.usr U 1 - 2
fonts.usr U 5 - 28
FONTSIZE U 5 - 8
fontsize U 5 - 8
force R 10 - 13
foreground U 5 - 18
fprefixlimit R 7 - 15
fprosymbol R 7 - 15
fputdouble R 7 - 7
fputinteger R 7 - 7
fputreal R 7 - 7
fputstring R 7 - 7
fputstrterm R 7 - 9
fputterm R 7 - 17
FRAMEPANEL U 5 - 3; 7
framepanel U 5 - 7; 17
free R 2 - 8
free R 4 - 2
freeze R 2 - 10
freplace R 3 - 24
fresetpermanentsymbol R 7 - 16
fretract R 3 - 24
fretractall R 3 - 24
front_window U 5 - 10
fr_err.txt U 1 - 3
fsetpermanentsymbol R 7 - 16
fsymbolstring R 7 - 15
functor R 10 - 9
gc R 6 - 25
gendialog U 5 - 41
gensymbol R 4 - 12
get R 10 - 15
get0 R 10 - 16
getenv R 6 - 5
get_arg_type R 7 - 3
get_attribute U 5 - 17
get_byte R 10 - 15
get_char R 10 - 15
get_code R 10 - 16
get_double R 7 - 5
get_error_complement R 8 - 11
get_event U 5 - 13
get_event U 5 - 20
get_formats R 8 - 8; 9
get_integer R 7 - 5
get_key U 5 - 36
get_local_object U 5 - 20
get_max_string R 7 - 5
get_objects U 5 - 20
get_option R 6 - 24
get_real R 7 - 5
get_screen U 5 - 6
get_string R 7 - 5
get_strterm R 7 - 9
get_term R 7 - 16
get_tlv R 2 - 15
get_window U 5 - 10
graphic.mo U 1 - 3; 4
graphic.mo U 5 - 6
GRAPHICS U 5 - 3; 8
graphics U 5 - 8; 17
graphic_area R 5 - 2
graphic_system U 5 - 6
graphstr.mo U 1 - 3
group U 5 - 17
gr_arc U 5 - 31
gr_arc' U 5 - 32

gr_button_hit U 5 - 35
gr_choosecolor U 5 - 26
gr_click U 5 - 33
gr_clickr U 5 - 34
gr_color U 5 - 25
gr_color1 U 5 - 25
gr_color2 U 5 - 25
gr_color3 U 5 - 25
gr_dialog U 1 - 3
gr_dialog U 5 - 41
gr_draw_buttons U 5 - 34
gr_editf U 5 - 35
gr_erase U 5 - 29
gr_font U 5 - 28
gr_getmouse U 5 - 34
gr_getmouser U 5 - 34
gr_getorigin U 5 - 29
gr_get_mode U 5 - 28
gr_get_pen U 5 - 27
gr_icon U 5 - 32
gr_line U 5 - 29
gr_lineto U 5 - 29
gr_list U 5 - 37
gr_load U 5 - 32
gr_mode U 5 - 27
gr_move U 5 - 29
gr_moveto U 5 - 29
gr_pen U 5 - 26
gr_penloc U 5 - 29
gr_penlocr U 5 - 29
gr_polygon U 5 - 30
gr_popupItem U 5 - 38
gr_print U 5 - 11
gr_rect U 5 - 29
gr_sensitive U 5 - 34
gr_setorigin U 5 - 29
gr_stringwidth U 5 - 27
gr_text U 5 - 28
gr_tree_click U 5 - 41
gr_window U 5 - 10
gr_window_is U 5 - 10
gtty U 5 - 37
halt R 10 - 20
heap R 6 - 26
hidden R 3 - 25
hidden_debug R 3 - 25
hidden_rule R 3 - 25
hscroll U 5 - 15; 16
ident R 4 - 1
if R 4 - 5
ignore_ops R 10 - 19
in R 2 - 11
in R 5 - 4
include R 3 - 25
index R 3 - 21; 32
indexation R 3 - 21
inf R 4 - 4
infe R 4 - 5
infinite R 2 - 11
infinite_flag R 6 - 12
initial.po R 3 - 4
initial.po U 1 - 3; 8
initial.po U 2 - 2; 4
initialization R 3 - 27
InitializeProlog U 3 - 4
init_fassert R 3 - 26
init_screen U 5 - 6
ini_module R 3 - 15; 17
ini_module R 6 - 1
inl R 5 - 4
input R 5 - 8
input R 6 - 5
input R 10 - 13
input U 5 - 39

input_is R 5 - 8
insert R 3 - 27; 28
insert R 6 - 5
insert U 2 - 7
insertz R 3 - 27; 28
integer R 4 - 1
integer_rounding_function R 10 - 19
Interruption R 2 - 6
interruption R 9 - 1
INT_ARRAY R 7 - 18; 21
int_edit.mo U 1 - 4
int_edit.mo U 5 - 6
in_char R 5 - 3
in_char' R 5 - 4
in_double R 5 - 5
in_ident R 5 - 6
in_integer R 5 - 5
in_real R 5 - 5
in_sentence R 5 - 6
in_string R 5 - 5
in_word R 5 - 6
is R 10 - 2; 7
is_array R 4 - 9
is_uncompiled R 3 - 28
items U 5 - 19
items_nb U 5 - 17
keysort R 4 - 16
kill_array R 4 - 9
kill_array R 7 - 18
kill_array U 2 - 8
kill_goal R 8 - 3
kill_module R 3 - 32
kill_module R 7 - 18
kill_module U 2 - 8
kill_object U 5 - 14
kill_window U 5 - 10
Label U 5 - 4
label U 5 - 17
left U 5 - 15
left_attach U 5 - 13
lg_buffer R 10 - 13
line R 5 - 11
line_width R 5 - 13
list R 3 - 14; 28; 29
Listbox U 5 - 5
listbox U 5 - 17
listing R 10 - 6
list_of R 2 - 8
list_string R 4 - 12
list_tuple R 4 - 12
lkload R 6 - 5
lkload R 7 - 27
ln R 4 - 6
load R 3 - 32
load R 6 - 5
load U 2 - 8
log R 10 - 7
max_arity R 10 - 19
member R 4 - 12
memory_file R 5 - 1
memory_file R 5 - 2
message_box U 5 - 35
mod R 1 - 23
mod R 4 - 4
mod R 10 - 2
MODAL U 5 - 8
modal U 5 - 8
mode R 10 - 13
module R 3 - 15
module U 2 - 7
month R 6 - 4
"MPW" U 5 - 37
ms_err R 6 - 5
mul R 4 - 4

multifile R 3 - 28
multifile R 3 - 29
multiple U 5 - 15
name R 10 - 9
new_check_button U 5 - 14
new_drawing_area U 5 - 14
new_edit_field U 5 - 15
new_goal R 8 - 3
new_label U 5 - 15
new_listbox U 5 - 15
new_pattern R 8 - 8
new_popup_menu U 5 - 16
new_pulldown_menu U 5 - 16
new_push_button U 5 - 14
new_radio_button U 5 - 14
new_scrollbar U 5 - 16
new_tlv R 2 - 14
new_window U 5 - 7
next_char R 5 - 4
next_char' R 5 - 4
next_solution R 8 - 3
nl R 10 - 17
nonvar R 10 - 6
not R 2 - 9
not R 10 - 14
not_defined R 3 - 29
no_border U 5 - 14; 15
no_debug R 6 - 18
no_echo R 5 - 14
NO_GOAL R 8 - 2
no_index R 3 - 33
no_infinite R 2 - 11
no_paper R 5 - 11
NO_RESIZE U 5 - 8
no_resize U 5 - 8
no_spy R 6 - 8; 18; 20
no_trace R 6 - 7; 18
number R 10 - 6
numbervars R 10 - 19
number_chars R 10 - 10
number_codes R 10 - 10
obdialog.mo U 1 - 3
OFFSET_ZERO_BASED R 7 - 18
omodule R 3 - 16
once R 10 - 4
op R 5 - 15
open R 10 - 13
opérateurs R 1 - 15; 22
opérateurs R 5 - 15
optimisations R 2 - 5; 11
optimisations R 3 - 20
optimisations R 4 - 3; 7; 8
optimisations R 6 - 6
option fermeture U 5 - 11
or R 2 - 9
out R 5 - 9
outl R 5 - 9
outm R 5 - 10
outml R 5 - 10
output R 5 - 13
output R 10 - 13
output U 5 - 39
output_is R 5 - 13
out_equ R 5 - 11
page R 5 - 11
paper R 5 - 11
paper U 2 - 9
parent U 5 - 3
parent U 5 - 17
past R 10 - 14
peek_byte R 10 - 16
peek_char R 10 - 16
peek_code R 10 - 16
peek_event U 5 - 21

phrase R 10 - 10
point d'arrêt R 6 - 7
Popup menu U 5 - 5
popup_menu U 5 - 17
position R 10 - 13
predefined R 3 - 29
predicate U 5 - 18
prefix_limit R 3 - 7
prefix_limit R 7 - 15
princip.c R 8 - 8
princip.c U 1 - 4
princip.c U 3 - 4
princip.obj U 1 - 4
print_window U 5 - 11
PROEDIT.PRO R 6 - 3
proentry.c U 1 - 4
proentry.c U 3 - 4
proentry.obj U 1 - 4
proext.h R 7 - 18
proext.h U 1 - 4
ProFinal R 8 - 3; 5
ProFinal U 3 - 4
prolink R 7 - 19
prolink.bat U 1 - 5
prolink.bat U 2 - 12
prolog U 5 - 21
prolog.def U 1 - 5
prolog.def U 3 - 3
prolog.exe U 1 - 3
prolog.lib U 1 - 5
prolog.log U 2 - 9
prolog.po R 6 - 1
prolog.po U 2 - 2; 11
prolog.res U 1 - 5
prolog2.pre U 1 - 4
prolog2.pre U 2 - 4
PrologDir2 U 1 - 4; 8
PrologDir2 U 2 - 2; 9
PrologEdit U 1 - 4; 8
PrologEdit U 5 - 6
prologII R 6 - 24
prologIIE R 6 - 24
ProStart R 8 - 3; 5
ProStart U 3 - 4
protected U 5 - 18
prouser.c R 4 - 10
prouser.c U 1 - 5
prouser.c U 2 - 12
prouser.c U 3 - 3
prouser.obj U 1 - 5
PRO_BIND R 7 - 19
pro_signal R 9 - 3
pro_symbol R 7 - 15
Pulldown menu U 5 - 5
pulldown_menu U 5 - 17
Push button U 5 - 4
push_button U 5 - 17
put R 10 - 18
put_byte R 10 - 18
put_char R 10 - 18
put_code R 10 - 18
put_double R 7 - 7
put_integer R 7 - 7
put_real R 7 - 7
put_string R 7 - 7
put_strterm R 7 - 9
put_term R 7 - 16
quit R 6 - 1
quit U 2 - 2
quoted R 10 - 19
rad R 4 - 6
Radio button U 5 - 4
radio_button U 5 - 17
rank U 5 - 18

read R 5 - 2
read R 10 - 16
read_line R 10 - 16
read_rule R 5 - 7
read_term R 10 - 16
read_unit R 5 - 7
real R 4 - 1
realloc R 6 - 25; 26
reconsult R 10 - 5
redef_array R 4 - 9
reinsert R 3 - 27; 28
reload R 3 - 33
reload U 2 - 8
rem R 1 - 23
rem R 4 - 4
rem R 10 - 2
remove_implicit R 3 - 12
remove_sentence_terminator R 5 - 7
repeat R 2 - 9
reposition R 10 - 13; 14
reset R 10 - 13
reset_chrono R 6 - 4
reset_cpu_time R 6 - 4
reset_permanent_symbol R 6 - 26
reset_permanent_symbol R 7 - 16
reset_window U 5 - 11
restore_C_backtrack_data R 7 - 23
restore_menubar U 5 - 22
restore_sysmenus U 5 - 22
retract R 3 - 31
retract R 10 - 6
retractall R 10 - 6
right U 5 - 15
right_attach U 5 - 13
round R 4 - 5
rule R 3 - 25; 30
rule_nb R 3 - 31
SAVE U 5 - 9
save R 3 - 33
save U 2 - 8
save U 5 - 9; 14
save_menubar U 5 - 22
save_state R 3 - 34
save_window U 5 - 11
Scrollbar U 5 - 5
scrollbar U 5 - 17
scrollb_page U 5 - 19
scrollb_pos U 5 - 19
scrollb_range U 5 - 19
scrollb_step U 5 - 20
see R 10 - 17
seeing R 10 - 17
seen R 10 - 17
selected_items U 5 - 19
send_external_event U 5 - 48
send_prolog_interrupt R 9 - 3
setarg R 4 - 12
setof R 2 - 9
set_alias R 1 - 29
set_attribute U 5 - 17
set_context R 3 - 11
set_cursor R 5 - 11
set_draw_mode R 5 - 12
set_import_dir R 6 - 5
set_input R 10 - 17
set_line_cursor R 5 - 11
set_line_width R 5 - 13
set_menu U 5 - 22
set_options R 6 - 24
set_output R 10 - 18
set_permanent_symbol R 6 - 26
set_permanent_symbol R 7 - 15; 21
set_prefix_limit R 3 - 7
set_prolog_flag R 10 - 20

set_stream_position R 10 - 13
set_tlv R 2 - 15
set_window U 5 - 11
sfgetfile U 5 - 39; 40
sfputfile U 5 - 40
SHAPE U 5 - 8
shape U 5 - 8
show_spy R 6 - 8; 17; 18
sign R 4 - 5
sin R 4 - 6
singleton R 3 - 27
singletons R 10 - 17
SINGLE_FLOAT_ARRAY R 7 - 18; 21
SOLUTION_EXISTS R 8 - 2
sort R 4 - 16
split R 4 - 13
sprintf R 5 - 11
sprintf R 7 - 27
spy R 6 - 8; 18; 20
sqrt R 4 - 6
sscanf R 5 - 8
sscanf R 7 - 27
stack R 6 - 26
stacks R 6 - 25
StartPrologMainGoal U 3 - 4
state R 6 - 24
state U 5 - 18; 19
statistics R 6 - 24
store_C_backtrack_data R 7 - 23
stream_property R 10 - 13
string R 4 - 1
STRING_ARRAY R 7 - 18; 21
string_double R 4 - 13
string_ident R 3 - 4
string_ident R 4 - 13
string_integer R 4 - 13
string_real R 4 - 13
string_term R 4 - 14
stty U 5 - 36
style_menu U 5 - 25
sub R 4 - 4
substring R 4 - 10
sub_atom R 10 - 11
sup R 4 - 5
supe R 4 - 5
suppress R 3 - 31
suppress R 7 - 18
suppress U 2 - 8
SYMBOL_ARRAY R 7 - 17; 18; 21
symbol_string R 7 - 15
sys_command R 6 - 5
tab R 10 - 18
tab_user_param U 2 - 9
tan R 4 - 6
tassign R 4 - 7
tell R 10 - 18
telling R 10 - 18
TerminateProlog U 3 - 4
term_cmp R 2 - 8; 9
term_cmp R 4 - 15
term_cmpv R 4 - 15
term_expansion R 10 - 21
term_vars R 4 - 14
text R 5 - 1
text U 5 - 18; 19
text_selection U 5 - 18
throw R 10 - 4
time R 6 - 4
told R 10 - 18
toplist U 5 - 19
top_attach U 5 - 13
top_left U 5 - 18
to_begin R 6 - 2
trace R 6 - 7; 18

trail R 6 - 26
true R 10 - 4
trunc R 4 - 5
truncate R 10 - 7
TTY U 5 - 3; 8
"TTY" U 5 - 37
tty U 5 - 8; 17
tty_area R 5 - 2
tuple R 4 - 1
tval R 4 - 3
type R 10 - 13
type U 5 - 17
unify_tlv R 2 - 15
unify_with_occurs_check R 10 - 5
unix_pipe R 5 - 1
unknown R 10 - 19
update U 5 - 21
userdll.mak U 1 - 6
usermpty.c U 1 - 6
userrsc.rc U 3 - 3
user_field U 5 - 18
use_win.h U 1 - 5
val R 4 - 3
val U 2 - 7
var R 10 - 6
variables R 10 - 16
variable_names R 10 - 16
var_time R 4 - 2
version R 6 - 24
visibility U 5 - 19
vscroll U 5 - 15; 16
wait U 5 - 17
warning U 2 - 8
week R 6 - 4
width_height U 5 - 19
write R 5 - 2
write R 10 - 18
writeq R 10 - 18
write_canonical R 10 - 18
write_term R 10 - 18
\ R 10 - 3; 7
\+ R 10 - 3
\+ R l0 - 2
\/ R 1 - 23
\/ R 4 - 6
\/ R 10 - 2
\= R 10 - 2; 3
\== R 10 - 2; 11

Ref. MPH-II/Fr/MMX

www.prolog-heritage.org