8

Environnement

DANS CE CHAPITRE on trouvera la description des options de la ligne de commande, de la compilation, de la mise au point des programmes, et d'autres fonctionnalités qui facilitent la vie du programmeur.

8.1 Les options de lancement

Le lancement depuis un shell de l'exécutable Prolog IV sans son environnement graphique s'effectue en lançant une commande de la forme suivante :

```
% prolog4 [ option ... option] [-- optionsUtilisateur...]
```

dans laquelle chacune des options est décrite plus loin, et où les options-utilisateurs sont des chaînes quelconques. Ce qui est entre crochets (qui ne doivent pas être tapés) est optionnel. C'est le double-tiret qui sépare les options destinées à Prolog IV de celles destinées au programme écrit en Prolog IV.

La totalité des options peut être récupérée par la primitive `argv/1` et la partie «utilisateur» peut être récupérée au moyen de la primitive `user_argv/1`.

8.1.1 Options

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

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

Dans ce qui suit, on appelle cellule un double mot-machine [1].

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

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

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

8.1.2 Primitives

argv(L), user_argv(L) récupèrent la liste de paramètres de la ligne de commande. Cette liste se compose d'atomes. `user_argv/1` ne récupère que les arguments situés après le séparateur `--` ; il rend `[]` s'il n'y en a aucun. Par exemple :

```
machine% prolog4 -heap 30000 -- carres 9
...
» argv(L).
L = ['prolog4','-heap','30000',--,carres,'9'].
» user_argv(L).
L = [carres,'9'].
```

8.2 Interruption utilisateur

A tout moment, un programme Prolog IV peut être interrompu au moyen d'une touche ou d'une action déterminée, dépendante du système d'exploitation utilisé (par exemple les touches `CTRL-C` sous Unix, ou le bouton «Stop» dans la fenêtre «Console» de l'environnement graphique de Prolog IV).

L'interruption de programme sert deux desseins :

- Arrêter un programme qui tourne (qu'il boucle ou qu'il dure trop longtemps).
- Passer dans un autre mode d'exécution (en mode débogueur).

Sous Unix, la même touche sert à arrêter un programme et à passer en mode débogueur. Ce qui discrimine est la façon d'utiliser cette touche :

- Si cette touche est pressée *deux fois* en moins d'une seconde, on passe en mode débogueur qui donne la main à l'utilisateur (en affichant le prompt du débogueur).
- Sinon, il s'agit d'une demande d'arrêt de programme : une erreur «`user_interrupt`» est générée par Prolog IV. Si cette erreur n'est pas récupérée par le programme, celui-ci termine et le prompt invitant à taper une nouvelle requête est affiché.

8.3 Compilation des règles

Après un bref rappel sur les primitives d'entrée de règles, il sera donnée une description des directives comprises par celles-ci.

8.3.1 Primitives de compilation

Il existe deux familles de primitives d'entrée de règles :

- Celles qui codent des paquets de règles dynamiques (famille `consult`).
- Celles qui codent des paquets de règles statiques (famille `compile`).

La plupart du temps, on préfèrera la compilation à la consultation : l'exécution des règles compilées est plus rapide, et le débogueur ne peut fonctionner qu'avec les règles compilées.

compile(F**), compile(**F**,** L**)** compilent le fichier de nom F (un atome). Les règles compilées sont statiques. L est une liste d'options portant sur le mode syntaxique et le mode débogueur de cette compilation.

recompile(F**), recompile(**F**,** L**)** fonctionnent comme les primitives `compile/`n, mais sans erreur de redéfinition.

consult(F**), consult** lisent et codent les règles du fichier de nom F (un atome). Les règles consultées sont dynamiques. Sans argument, les règles sont lues dans l'entrée courante.

reconsult(F**), reconsult** fonctionnent comme les primitives `consult/`n, mais sans erreur de redéfinition.

8.3.2 Directives de compilation

Les directives sont des annotations permettant de passer certaines informations au compilateur, d'altérer son comportement, de spécifier les propriétés des paquets de règles présents dans le fichier, d'inclure un fichier etc. Les directives ne sont comprises que du compilateur (primitives `compile`) et non pas des primitives `consult`.

Voici la liste de ces directives :

dynamic/1 Se comporte comme la primitive `dynamic/1`. Le compilateur ne pouvant que compiler, et non pas consulter, la présence dans le fichier d'une règle déclarée dynamique provoquera une erreur pendant la compilation.

op/3 Se comporte comme la primitive `op/3`. La portée des opérateurs déclarée est générale : ils ne sont pas éliminés en fin de compilation.

syntax/1 Informe le compilateur que les paquets de règles doivent être compilés en mode :

- `iso`, pour que les extensions syntaxiques majeures sont déconnectées,
- `prolog4` pour que la syntaxe soit augmentée afin d'alléger les notations décrivant les contraintes.

Par défaut, le mode courant (indiqué par le prompt de la console) sert de mode de compilation du fichier.

include/1 La directive `:-include(`$fichier$`)` indique au compilateur de lire le fichier $fichier$ (un atome) avant de poursuivre la compilation du fichier courant. Tout se passe comme si la directive était remplacée par le contenu du fichier indiqué en argument.

debug/0 Les paquets de règles qui suivent jusqu'à la fin du fichier ou jusqu'à la prochaine directive `:-no_debug` sont compilés d'une manière qui permet au débogueur de suivre l'exécution de ces paquets.

no_debug/0 Désactive la compilation en mode `debug` des paquets de règles qui suivent.

set_prolog_flag/2 Appelle simplement en ce point la primitive `set_prolog_flag/2`.

8.4 Le débogueur Prolog IV

8.4.1 Introduction

Tout programme (même écrit en Prolog IV), peut être faux !
En effet, il n'existe pas de langage de programmation permettant de rentrer en machine «le problème que l'on a en tête» sans laisser la porte ouverte aux erreurs. Bien sûr, le premier précepte à suivre est de relire son code quand il ne fonctionne pas comme on pense qu'il le «devrait». Il faut aussi corriger les éventuels avertissements lancés pendant la compilation ou la vérification syntaxique. Dans bien des circonstances, l'examen détaillé du déroulement d'un programme et le suivi interactif de son exécution peuvent apporter de précieux indices au programmeur laissé perplexe face à son «bug»[2]. Le mode de fonctionnement `debug` de Prolog IV répond à ce besoin de dépistage.

8.4.2 Le modèle des boîtes

Le débogueur de Prolog IV est fondé sur le «modèle des boîtes» que l'on retrouve dans des prologs plus classiques. Le concept important de ce modèle est que *chaque appel peut être considéré comme une boîte*, opaque ou transparente selon le désir de l'utilisateur d'avoir une trace d'exécution plus ou moins détaillée. Les boîtes de ce modèle comprennent quatre ouvertures, appelées ports. Deux autres ports sont disponibles en Prolog IV, et permettent de voir à l'intérieur des boîıtes.

2. Bogue.

8.4.3 Les ports

Ces ports correspondent à des endroits précis de l'exécution d'un but. Ils schématisent :

- l'activation d'un paquet de règles (`CALL`),
- la terminaison correcte d'une des règles du paquet (`EXIT`),
- le retour au sein de ce paquet par backtracking (`REDO`),
- la désactivation de ce paquet par épuisement de ses choix (`FAIL`),
- le choix de la n^{ieme} règle d'un paquet (`RULE`).
- l'unification avec succès de la tête de règle avec le but courant (`OK`).

Les deux derniers sont des ports secondaires, internes à la boîte.

Le comportement d'une boîte étant naturellement non-déterministe, une même boîte peut être sollicitée à plusieurs reprises pour fournir différentes réponses. De façon séquentielle, on entre pour la première fois dans une boîte par `CALL`. On en sort avec succès par `EXIT` qui indique qu'une solution a été trouvée. Lorsque des choix ont été laissés en attente, on revient dans la boîte pour demander une solution alternative par `REDO`. L'absence d'une (autre) solution est indiquée par `FAIL`, qui provoque la sortie définitive de la boîte.

L'affichage des ports respecte un même schéma. A partir de l'exemple,

```
4 [2] EXIT (r1) : dessert(glace)
```

voici la description des différents champs de ce schéma :

`4 :` Profondeur d'un appel : C'est le numéro d'ordre de la boîte dans laquelle on se trouve avant d'avoir effectué l'action.

`[2] :` Niveau de l'appel : il est défini par le niveau d'imbrication de la boîte correspondant à cet appel. Ce numéro est tel que tous les appels situés dans une même queue de règle (ou dans une requête) ont un même niveau.

`EXIT :` C'est le nom du port, l'une des chaînes `CALL`, `EXIT`, `REDO`, `FAIL`, `RULE` ou `OK`.

`(r1) :` Le numéro de la règle appliquée. Lorsqu'il est suivi d'une étoile, il indique qu'il s'agit de la dernière règle du paquet.

`dessert(glace) :` C'est le littéral qu'on traite. Il est affiché dans son état actuel d'instanciation.

Les ports sont affichés avant d'être franchis.

8.4.4 Sessions de déboguage

On verra dans ce qui suit diverses sessions montrant quelques possibilités du débogueur Prolog IV, en présentant au passage les principales commandes. Dans tout ce qui suit, ce qui doit être tapé par le programmeur est souligné.

Soit le petit programme suivant, qu'on place dans le fichier `dessert.p4` :

```
dessert(fruit, 2) .
dessert(ice_cream,6) .
```

Compilons-le en mode `debug` :

```
>> compile('dessert.p4', [debug(on)]).
true.
```

Maintenant, suivons le déroulement de l'appel au paquet `dessert` ; à chaque fois que le prompt du débogueur s'affiche, nous répondrons par la commande `s` (raccourci de la commande `step` qui permet la progression de port en port).

Pour simplifier la présentation, certains retours chariots ont été omis, ainsi que divers séparateurs de solutions.

```
>> debug.
true.

>> dessert(X,Y).

   4[4]CALL    : dessert(_267, _268) (DBG IV) s
    5[5]RULE(r1): dessert/2(_267, _268) (DBG IV) s
    5[5]OK(r1): dessert/2(fruit, 2) (DBG IV) s
    5[5]EXIT(r1): dessert/2(fruit, 2) (DBG IV) s
Y = 2,
X = fruit
  4[3]REDO(r1): dessert/2(_267, _268) (DBG IV) s
    5[5]RULE(r2*): dessert/2(_267, _268) (DBG IV) s
    5[5]OK(r2*): dessert/2(ice_cream, 6) (DBG IV) s
    5[5]EXIT(r2*): dessert/2(ice_cream, 6) (DBG IV) s
Y = 6,
X = ice_cream
    5[5]FAIL(r2*): dessert/2(_267, _268) (DBG IV) s

>>
```

Voici une explication de la session précédente, ligne par ligne :

- (`CALL`) On appelle `dessert`.
- (`RULE`) La première règle (`r1`) de `dessert` est essayée.
- (`OK`) La tête de cette règle est unifiée avec l'appel.
- (`EXIT`) Tous les littéraux de la queue de règle ont été exécutés (il n'y en avait aucun).
- La solution est affichée, car il n'y a plus de littéraux en attente d'exécution.
- (`REDO`) Il existe un point de choix pour l'appel à `dessert`.
- (`RULE`) On essaie la seconde règle (`r2`) de `dessert`.
- (`OK`) La tête de cette règle est unifiée avec l'appel.
- (`EXIT`) Tous les littéraux de la queue de règle ont été exécutés (il n'y en avait aucun).
- La solution est affichée, car il n'y a plus de littéraux en attente d'exécution.

– (FAIL) Il n'y a plus de choix à essayer pour l'appel à `dessert`.

– Retour au prompt Prolog IV. Tout l'arbre de recherche (un arbre avec des nœuds *et* et des nœuds *ou*) a été exploré.

Pour aller plus loin, utilisons maintenant le fichier `menu.p4` décrit ci-après.

```
repasleger(H,M,D) :-
    horsdoeuvre(H,I),
    plat(M,J),
    dessert(D,K).

horsdoeuvre(radis,1) .
horsdoeuvre(melon,6) .

plat(M,I) :- viande(M,I) .
plat(M,I) :- poisson(M,I) .

dessert(fruit, 2) .
dessert(glace,6) .

viande(poulet,5)  .
viande(porc,7)  .

poisson(sole,2) .
poisson(thon,4) .
```

Montrons avec cette session comment se déplacer de boîte en boîte sans voir le détail des appels.

```
>> compile('menu.p4', [debug(on)]).
true.

>> debug.
true.

>> repasleger(X,Y,Z).

    2[2]CALL    : repasleger(_267, _268, _269) (DBG IV) s
     3[3]RULE(r1*): repasleger/3(_267, _268, _269) (DBG IV) s
     3[3]OK(r1*): repasleger/3(_267, _268, _269) (DBG IV) s
     3[3]CALL    : horsdoeuvre(_267, _1213) (DBG IV) n
     3[3]CALL    : plat(_268, _1214) (DBG IV) s
     3[3]CALL    : dessert(_269, _1215) (DBG IV) s
      7[4]RULE(r1): dessert/2(_269, _1215) (DBG IV)
     ...
```

– Pour chacun des trois premiers ports, on fait du pas à pas (`step`).

– On veut ensuite sauter le détail de l'exécution des appels à `horsdoeuvre`, et à `plat`. On effectue pour ce faire la commande `next` (abrégée par `n`).

– Arrivé sur l'appel à `dessert`, on effectue du pas à pas pour voir le détail.

Montrons avec cette session comment progresser plus rapidement encore, par

le biais de point d'arrêts, toujours sans voir le détail des appels.

```
>> debug.
true.

>> repasleger(X,Y,Z).
    2[2]CALL    : repasleger(_267, _268, _269) (DBG IV) spy dessert

   dbg: spying: (dessert/2).
   (DBG IV) c
   **   3[3]CALL    : dessert(_269, _1215) (DBG IV) s
   **    7[4]RULE(r1): dessert/2(_269, _1215) (DBG IV) s
   **    7[4]OK(r1): dessert/2(fruit, 2) (DBG IV) c
   **    7[4]EXIT(r1): dessert/2(fruit, 2) (DBG IV) c
   Z = fruit,
   Y = poulet,
   X = radis
   **  6[2]REDO(r1): dessert/2(_269, _1215) (DBG IV) s
   **    7[4]RULE(r2*): dessert/2(_269, _1215) s
   **    7[4]OK(r2*): dessert/2(glace, 6)
   ...
```

– On utilise la commande `spy` pour installer un point d'arrêt sur les paquets de règles `dessert`.

– On effectue la commande `cont` (abrégée par `c`) pour continuer l'exécution du programme jusqu'à ce qu'un port faisant intervenir le littéral espionné (`dessert` dans notre cas) soit détecté. La machine s'y arrête alors.

– On peut ensuite au moyen de `step` détailler notre appel.

Les marques `**` qui figurent en marge indiquent que le port en question est un point d'arrêt.

Montrons quelques commandes affichant des informations diverses sur la démonstration et sur les variables :

```
(DBG IV) where
---\/---  NEW  ---\/---
   7[3](r2*) dessert/2(glace, 6)
   3[2](r1*) repasleger/3(radis, poulet, glace)
   2[1](r1*) sysh__query/1([’X’=radis,’Y’=poulet,’Z’=glace])
---/\---  OLD  ---/\---
(DBG IV) chpt
---\/---  NEW  ---\/---
   6[4](r1) viande/2(poulet, 5)
   5[3](r1) plat/2(poulet, 5)
   4[3](r1) horsdoeuvre/2(radis, 1)
---/\---  OLD  ---/\---
(DBG IV) up
(DBG IV) printvar
H = radis
M = poulet
D = glace
I = 1
J = 5
K = 6
(DBG IV)
```

– La commande `where` est utilisée pour montrer où l'on se trouve dans la démonstration courante (l'empilement des appels).

– La commande `chpt` sert à visualiser les points de choix qui restent à traiter.

– La commande `up` nous permet de retourner dans la procédure appelante. Ce changement de contexte nous permet d'interroger le débogueur sur les valeurs des variables de ce contexte-là.

– La commande `printvar`, sans argument, nous montre les valeurs de toutes les variables créées lors de la création de ce contexte.

8.4.5 Options du débogueur

Les options [3] sont des variables internes au débogueur. Elles servent essentiellement de paramètres implicites à certaines commandes.

Dans plusieurs commandes et options, les types de port sont repérés par une lettre. On a *in extenso* c pour CALL, e pour EXIT, r pour REDO, f pour FAIL, R pour RULE et Y pour OK. Une combinaison de ports est un suite de ces lettres, comme par exemple «`ceR`». La combinaison d'aucun port est donné par « { } ».

Les options du débogueur sont `stepports`, `nextports`, `contports`, `spyports`. Voici leur description :

stepports : est utilisée par la commande `step` pour la progression de port en port. Par défaut, cette option contient la chaîne «cerfRY» (tous les ports).

3. Le mot «option» est préféré au mot «variable» pour éviter toute confusion avec les variables prolog.

nextports : est utilisée par la commande `next` pour limiter l'affichage pendant la progression. Par défaut, cette option contient la chaîne vide (aucun port).

contports : est utilisée par la commande `cont` pour limiter l'affichage pendant la progression. Par défaut, cette option contient la chaîne vide (aucun port).

spyports : est utilisée par la commande `spy` pour installer un point d'arrêt sur un ou plusieurs paquet de règles, pour les ports décrits dans cette option. Par défaut, cette option contient la chaîne «cerfRY» (tous les ports).

8.4.6 Commandes

Les commandes de progression permettent d'avancer par pas variables dans un programme. Il faut noter qu'en prolog, suivant le sens du contrôle, le port suivant peut être à une profondeur plus grande ou plus faible que le port courant. Voici la liste de ces commandes : `step`, `next`, `cont`.

La gestion des points d'arrêt : `spy`, `unspy`.

L'affichage d'informations : `printvar`, `subdomain`, `chpt`, `where`, `up`, `down`.

Commandes diverses : `prolog`, `kill`, `help`.

On appelle *contexte* ou encore *contexte d'appel* un point de la démonstration courante. On peut voir un contexte comme une règle appliquée, ses variables étant dans un certain état d'instanciation, selon les appels qui ont déjà été effectués dans la queue de règle. Les contextes sont naturellement chaînés par la structure des appels imbriqués (un seul appel est actif à la fois.) On peut se déplacer parmi les contextes au moyen des commandes `up` et `down`.

step

C'est la commande de base pour suivre la progression pas à pas de la machine prolog. Avance pas à pas de port en port. Seuls les ports dont le type est indiqué par l'option `stepports` sont pris en compte. On peut donc ignorer certains types de ports et avancer plus vite. Son raccourci est `s`.

step ***ports***

Positionne l'option `stepports` avec la valeur *ports*, puis effectue la commande `step`. Raccourci : `s`.

next

Avance d'appel en appel en affichant au passage les ports dont le type est indiqué par l'option `nextports`. Seuls les appels de niveaux inférieurs ou égaux au niveau courant sont montrés. Autrement dit, on ne détaille pas les appels, que l'on peut considérer comme autant de boîtes noires. Si on ne peut aller à l'appel suivant (pour cause d'échec prolog par exemple), cette commande se comporte comme `step`. Raccourci : `n`.

next *ports*

Positionne l'option `nextports` avec la valeur *ports*, puis effectue la commande `next`. Raccourci : `n`.

cont

Continue l'exécution en affichant au passage les ports dont le type est indiqué par l'option `contports`. Si une point d'arrêt est détectée, la main est rendue au programmeur. Raccourci : `c`.

cont *ports*

Positionne l'option `contports` avec la valeur *ports*, puis effectue la commande `cont`. Raccourci : `c`.

where

Affiche l'empilement des appels, du plus récent au plus ancien. Raccourci : `w`.

source 1 *ou* 0

Active/désactive le mode «*source*» du débogueur. Ce mode n'a d'intérêt qu'en présence de l'environnement graphique de Prolog IV. Quand celui-ci est présent, le mode `source` est automatiquement positionné (à 1).

pbox 1 *ou* 0

Active/désactive l'affichage des lignes de statut du débogueur, c.a.d. ne montre plus les affichages textuels des ports rencontrés. Désactiver cet affichage (en mettant 0) n'a d'intérêt qu'en présence de l'environnement graphique, quand on ne veut plus se servir que du mode `source`. Par défaut, cet affichage est présent.

kill

Abandonne l'exécution de la requête courante. On revient au prompt Prolog IV. Le mode `debug` reste toutefois actif. Raccourci : `k`.

no_debug

Abandonne le mode `debug`. La prochaine commande tapée permet d'abandonner l'exécution du programme en cours (avec `kill`), ou de la poursuivre (avec `cont` par exemple).

halt

Termine l'exécution de Prolog IV. On revient au système d'exploitation.

up

Déplace le pointeur de contextes dans le contexte supérieur (celui de la règle appelante). Si on est dans le contexte le plus haut, un avertissement est affiché et rien n'est fait. Raccourci : `u`.

down

Déplace le pointeur de contextes dans le contexte inférieur (celui de la règle appelée) Si on est dans le contexte le plus bas, un avertissement est affiché et rien n'est fait. Raccourci : `d`.

printvar

Affiche toutes les variables du contexte courant. En conjonction avec des commandes `up` et `down`, permet d'examiner les variables qui participent à la démonstration courante. Il n'y a pas d'affichage de sous-domaine (voir la commande `subdomain` pour cela). Cette commande est basée sur la primitive Prolog `write`. Raccourci : `p`.

printvar *var* ...

Même chose, mais en se restreignant a certaines variables du contexte courant. Raccourci : `p`.

subdomain

Affiche les sous-domaines de toutes les variables appartenant au contexte courant (ou au contexte positionné par `up` ou `down`). La commande `subdomain` donne des informations sur les variables qui entrent en jeu d'une façon plus précise et plus élégante qu'avec `printvar`. Cette commande est basée sur le système d'affichage de la solution d'une requête. Raccourci : `sd`.

subdomain *var* ...

Même chose, mais en se restreignant a certaines variables du contexte courant. Raccourci : `sd`.

help

Affiche la liste des commandes disponibles, avec une description succincte. Raccourci : `h`.

help *cmd* ...

Affiche une description succincte de la commande *cmd*. Raccourci : `h`.

chpt

Affiche la pile des points de choix, du plus récent au plus ancien.

spy

Met un point d'arrêt sur le paquet dont le nom est celui de l'appel courant, pour les seuls ports contenus dans l'option `spyports`.

spy *nom* ...

Met un point d'arrêt sur le paquet de nom *nom*, pour les seuls ports contenus dans l'option `spyports`. Les noms peuvent avoir la forme *identificateur/arité* ou *identificateur*. Dans ce dernier cas, tous les paquets de noms *identificateur*, quelle que soit leur arité, sont espionnés.

unspy

Enlève un (éventuel) point d'arrêt sur le paquet dont le nom est celui de l'appel courant.

unspy ***nom ...***

Enlève un (éventuel) point d'arrêt sur le paquet de nom *nom*.

spyports ***ports***

Positionne l'option `spyports` (utilisé par la commande `spy`) à la valeur *ports*. Sans argument, affiche la valeur courante de l'option `spyports`.

prolog

Lance une nouvelle session Prolog IV imbriquée : le prompt de Prolog IV est affiché et on peut taper des requêtes (on sort temporairement du mode débogueur). Cette nouvelle session doit être terminée par le but « `stop.` ». On revient alors à la session de déboguage courante (sous le prompt du débogueur). Cette commande permet d'effectuer des vérifications, en lançant des buts, sans abandonner la session de déboguage courante, ni avoir à lancer une seconde exécution de Prolog IV en parallèle.

stepports ***ports***

Positionne l'option `stepports` (utilisé par la commande `step`) à la valeur *ports*. Sans argument, affiche la valeur courante de l'option `stepports`.

nextports ***ports***

Positionne l'option `nextports` (utilisé par la commande `next`) à la valeur *ports*. Sans argument, affiche la valeur courante de l'option `nextports`.

contports ***ports***

Positionne l'option `contports` (utilisé par la commande `cont`) à la valeur *ports*. Sans argument, affiche la valeur courante de l'option `contports`.

8.4.7 Primitives et compilation

Voici comment entrer ou sortir du mode `debug` d'une part, et comment compiler un programme pour que celui-ci soit déboguable pendant l'exécution.

Changement de mode

debug, no_debug La commande prolog `debug` fait passer la machine prolog dans le mode `debug` quel que soit le mode dans lequel on était précédemment. On peut parenthéser des sessions de surveillance au sein de son programme au moyen des primitives `debug` et `no_debug`, mais il faut savoir que l'on ne disposera pas de toutes les informations sur la démonstration en cours si la requête n'a pas été lancée sous le mode `debug`. De plus, les informations sur les niveaux sont relatives au point de l'exécution où `debug` a été lancé, et non plus par rapport à la requête. Il n'y a aucun intérêt à passer en mode `debug` quand le programme qui tourne n'a pas été compilé pour ce mode.

Compilation

Pour compiler un programme avec des informations utilisables par le débogueur, il faut utiliser :

- la directive `:-debug.` dans le fichier (avant le premier paquet de règles),
- l'option de compilation `debug(on)`

Il est possible de compiler des portions de fichier (certains paquets de règles) en mode `debug` à l'aide des directives `:-debug` et `:-no_debug` que l'on place entre certains paquets de règles.

8.4.8 Problèmes et limitations

- Seuls peuvent être pris en compte par le débogueur les programmes donnés à Prolog IV au moyen des commandes de la famille `(re)compile`, quand ils sont explicitement compilés en mode `debug` (par quelque moyen que ce soit).
- Il s'ensuit que les paquets de règles qui ont été entrés au moyen de la famille des primitives `(re)consult` et `assert` sont ignorés par le débogueur.
- Les méta-appels, comme ceux lancés par `call`, «`;`» ou `freeze`, sont ignorés par le débogueur.