Chapter 10

Connexion entre Prolog IV et C ou TCL

Ce chapitre décrit les moyens mis à la disposition du programmeur Prolog IV, C ou TCL pour appeler des programmes ou sous-routines écrits avec l'autre langage.
Un exemple complet est donné en appendice.

10.1 Appeler Prolog IV depuis C

10.1.1 Description des opérations de base

Cette section décrit une partie les moyens mis à la disposition du programmeur C d'appeler des programmes et requêtes écrites en Prolog IV. Etant donné du code écrit en C, on doit pouvoir effectuer les opérations de base suivantes :

- Transformer des données utilisées dans ce programme C en données utilisables par Prolog IV.
- Effectuer un "appel Prolog IV" lequel est une forme de requête limitée à un seul littéral.
- Examiner le(s) statut(s) de(s) retour(s) de cet appel (plusieurs retours sont possibles, du fait du non-déterminisme).
- Transformer les données au format Prolog IV en données utilisables en C.

Il faut avoir préalablement initialisé Prolog IV, afin qu'il soit en condition d'accepter des requêtes.

10.1.2 Initialisation de Prolog IV

```
void p4init(int argc, char *argv[]);
```

Cette fonction initialise la machine Prolog IV. Ses deux arguments *argc* et *argv* ont la même signification que les arguments de même nom de la fonction `main()` traditionnelle. Cette fonction doit être appelée avant toute autre fonction en rapport direct avec Prolog IV, qui est décrite dans ce chapitre.

10.1.3 Construction d'objets Prolog IV

typedef ... **P4SYMBOL**;
typedef ... **P4TERM**;

P4SYMBOL est le type C correspondant à un objet du dictionnaire des symboles Prolog IV. Un symbole n'est pas un terme, c'est un intermédiaire indispensable à la construction de certains termes comme les atomes ou les structures comme les foncteurs.
P4TERM est le type C correspondant à un terme Prolog IV.
Sauf indication contraire, on restreindra ici la notion de type prolog à l'aspect pragmatique lié à la programmation, par opposition au véritable mécanisme de typage faisant partie intégrante du système de résolution de Prolog IV.

Construction d'un symbole utilisable par Prolog IV

int **p4cstring_to_symbol**(char *cstring, P4SYMBOL *symbol);

Etant donnée *cstring* une chaîne de caractères C, cette fonction crée (ou retrouve) un symbole dans le dictionnaire Prolog IV; ce symbole est retourné dans l'argument *symbol*. S'il n'y a pas d'erreur, retourne 0, sinon retourne une valeur non-nulle.

char * **p4symbol_to_cstring**(P4SYMBOL symbol);

C'est en quelque sorte la fonction inverse de la fonction précédente.
Etant donné *symbol* un symbole préalablement construit par la fonction **p4cstring_to_symbol**, cette fonction retourne la chaîne de caractères associée dans le dictionnaire Prolog IV.
S'il n'y a pas d'erreur, retourne la chaîne de caractères, sinon retourne le pointeur `NULL`.

Construction de termes

Toutes les fonctions de construction de termes respectent le schéma suivant : **P4TERM p4make_*ppp*(arguments)**, où *ppp* est un type ou une construction prolog. Toutes ces fonctions retournent `NULL` en cas d'erreur ; la variable **p4errno** est alors positionnée.

P4TERM **p4make_atom**(P4SYMBOL symbol);

Cette fonction construit et retourne un terme de type atome (identificateur) à partir du symbole *symbol*.

P4TERM **p4make_lint**(long int cint);
P4TERM **p4make_ulint**(unsigned long int cint);

Ces fonctions construisent et retournent un terme de type entier à partir de l'entier *cint*.

P4TERM **p4make_rational_from_strfloat**(char *floatstring);

Cette fonction construit et retourne un terme de type "nombre rationnel" (nombre en précision parfaite) à partir de la chaîne de caractères *floatstring* contenant sa représentation décimale, pouvant comporter point-flottant et exposant. On pourra donner par exemple les chaînes `"4.5"`, `"8.12e1998"` `"1.000000000000000000000000000000001"`. L'argument ne doit pas comporter plus de mille caractères.

P4TERM **p4make_float**(double cfp);

Cette fonction construit et retourne un terme de type "nombre flottant" à partir de *cfp* .

P4TERM **p4make_functor**(int arity, P4SYMBOL symbol, ...);

Cette fonction (avec un nombre d'arguments variables) construit et retourne un terme de type foncteur ayant étiquette *symbol* et d'arité *arity*. L'argument *arity* indique le nombre de termes-fils de cette structure. Ils sont de type `P4TERM` et doivent impérativement être présents à l'appel de cette fonction.

`P4TERM` **p4vmake_functor**`(int arity, P4SYMBOL symbol, P4TERM terms[]);`

Cette fonction se comporte comme la précédente à la différence qu'elle a un nombre fixe d'arguments, les termes-fils de la structure devant être transmis au moyen du tableau *terms*. Ce tableau doit pouvoir contenir *arity* objets de type `P4TERM`.

`P4TERM` **p4make_nil**`(void);`

Cette fonction construit et retourne un terme [] (qu'on prononce nil); c'est en fait un cas particulier d'appel à la fonction **p4make_atom**().

`P4TERM` **p4make_dot**`(P4TERM car, P4TERM cdr);`

Cette fonction construit et retourne une paire pointée, c'est à dire une structure d'arité 2 et de foncteur spécial '.' qui sert à dénoter les listes prolog. Les deux éléments de cette paire pointée sont les termes *car* et *cdr*. Rappelons qu'une liste est soit vide ([]) soit une paire pointée dont le membre droit est lui-même une liste.

`P4TERM` **p4make_var**`(void);`

Cette fonction construit et retourne une nouvelle variable Prolog IV.

Note sur les constructeurs de termes :

- Dans cette version de l'interface C/PrologIV, il n'est pas possible de construire depuis C un "gros entier" Prolog IV, c'est-à-dire un entier plus grand que $2^{32} - 1$ sur une machine à mots de 32 bits.
- Quand l'un des arguments de type `P4TERM` d'une de ces fonctions vaut `NULL`, celle-ci retourne `NULL` à son tour. Il est donc possible d'imbriquer des appels C sans crainte de dysfonctionnements dus à la présence de `NULL` en lieu et place d'un terme. Par exemple :

```
T = p4make_dot(p4make_lint(100),           /* .(100,.(101,[])) */
               p4make_dot(p4make_lint(101),/* ==               */
                          p4make_nil()));  /* [100, 101]       */
if ( T == NULL ) erreur...
   ...
```

10.1.4 Appel d'un but Prolog IV

Tout appel de but prolog se décompose en les phases suivantes :

- préparer un nouveau but (ouvrir une session),
- construire un terme (dans certains cas, ce peut-être facultatif),
- installer un terme donné comme étant le but à invoquer,
- invoquer le résolveur Prolog IV et boucler (ou pas) pour obtenir la ou les solutions par backtracking,
- terminer avec le but en cours (terminer la session).

Préparer un nouveau but (ouvrir une session)

`void` **p4new_session**`(void);`

Cette fonction crée une nouvelle session en préparant Prolog IV à exécuter un nouveau but. Ce qui était précédemment actif dans Prolog IV est temporairement suspendu.

Construire un terme

Voir la section décrivant la construction des termes (la section qui décrit les primitives de la forme **p4make_ppp()**)

Installer un terme donné comme but à invoquer

`void` **p4make_call**`(P4TERM but);`

Prépare le terme *but* pour l'exécution.

Invoquer le résolveur Prolog IV et obtenir les solutions

`int` **p4next_solution**`(void);`

Recherche la prochaine solution du but courant. Le retour de cette fonction indique le statut de la résolution à l'aide des constantes suivantes :

P4SESSION_SOLUTION indique un arrêt sur une solution.

P4SESSION_END indique un échec (ou qu'il n'y a plus de solution).

P4SESSION_ERROR indique une erreur non-rattrappée pendant l'exécution.

On peut soit invoquer **p4next_solution()** une seule fois pour rechercher la première solution, soit l'invoquer dans une boucle pour les rechercher toutes, comme décrit ci-après :

```
int statut;
while ( (statut=p4next_solution()) == P4SESSION_SOLUTION )
    { /* C'est une solution (ni echec ni erreur), */
      /* on recupere le resultat */
    ...
    }
if ( statut == P4SESSION_ERROR )
    { /* Il y a une erreur non-recuperee pendant l'execution */
    }
```

Terminer avec le but en cours (terminer la session)

`void` **p4finish_session**`(void);`

Elle est appelée quand on en a terminé avec le but courant. C'est une opération de nettoyage destinée à remettre la machine Prolog IV dans l'état ou elle se trouvait avant l'appel à **p4new_session**().

10.1.5 Determiner la nature des termes Prolog IV

Dans cette section sont décrites des fonctions permettant de déterminer le type d'un terme prolog. La première fonction mise à la disposition du programmeur permet de savoir quel est le type du terme qui lui est passé en argument.

`int` **p4what_is**`(P4TERM T);`

Teste le terme T.
Retourne un entier représentant le type du terme passé en argument. Cet entier peut être comparé à l'une des constantes suivantes :

P4INTEGER Un entier.

P4RATIONAL Une nombre rationnel non entier.

P4FLOAT Un nombre flottant (double précision). On rappelle que les nombres flottants (de la norme prolog) ne s'unifient pas avec les autres nombres tels que les entiers et rationnels.

P4ATOM Un atome (ou identificateur).

P4DOT Une paire pointée.

P4FUNCTOR Un noeud avec un ou plusieurs fils, n'étant pas une paire pointée.

P4VAR Une variable.

P4UNEXPECTEDTERM C'est un type de terme non-documenté dans cette version.

P4NULL Le pointeur `NULL` a été passé en argument. C'est une erreur.

P4NOTATERM Ce qui est passé en argument ne peut être compris comme étant un terme. C'est une erreur.

Toutes les fonctions qui suivent sont de la forme **p4is_***ppp*`(T)` et testent leur argument qui est un terme (type `P4TERM`). Elle retournent `TRUE` si le terme passé en argument est du type requis *ppp*, `FALSE` sinon. Si l'argument passé n'est pas un terme, il est retourné également `FALSE`. La variable globale entière **p4errno** indique alors quel est le type d'erreur qui est détecté.

`int` **p4is_integer**`(P4TERM T);`

Teste le terme *T*.
Retourne un nombre non nul si *T* est un entier (quelle que soit sa taille), retourne 0 sinon.
Ce nombre retourné est un statut composé de flags ; il permet d'obtenir des informations sur la nature du terme de type "entier", notamment en ce qui concerne la possibilité de le convertir en donnée C à l'aide des types "long int" et "unsigned long int".
Le schéma suivant montre les placements respectifs des domaines de valeurs possibles pour les entiers de deux types C et du type Prolog IV. Les signes "+" désignent les nombres positifs et les signes "-" des nombres négatifs. Les entiers Prolog IV sont non-bornés (ils ne sont limités essentiellement que par la taille de la mémoire).

```
           [-----0++++]                long int de C
                [0+++++++++]           unsigned long int de C
...--------------0+++++++++++++++...   entiers Prolog IV
```

Si le statut de retour n'est pas nul, c'est une combinaison de flags. Voici les plus utiles :

P4F_LINT l'entier tient dans le type C "long int".

P4F_ULINT l'entier tient dans le type C "unsigned long int".

P4F_NEGATIVE l'entier est négatif.

`int` **p4is_float**`(P4TERM T);`

Teste le terme *T*.
Retourne `TRUE` si *T* est de type "nombre flottant de la norme prolog", retourne `FALSE` sinon.

`int` **p4is_atom**`(P4TERM T);`

Teste le terme *T*.
Retourne `TRUE` si T est un terme de type "atome" (ou "identificateur"), retourne `FALSE` sinon.

`int` **p4is_nil**`(P4TERM T);`

Teste le terme *T*.
Retourne `TRUE` si T est le terme [], retourne `FALSE` sinon. On rappelle que [] est un atome spécial, dont il est convenu qu'il permet de clore une liste.

`int` **p4is_functor**`(P4TERM T);`

Teste le terme *T*.
Retourne `TRUE` si T est un foncteur, c'est-à-dire un noeud d'où sont issus un ou plusieurs fils, retourne `FALSE` sinon. Pour permettre une meilleure discrimination des termes, la paire pointée n'est pas reconnue par cette fonction comme étant un foncteur : voir la fonction dédiée aux paires pointées **p4is_dot**().

`int` **p4is_dot**`(P4TERM T);`

Teste le terme *T*.
Retourne `TRUE` si T est une paire pointée, retourne `FALSE` sinon. On rappelle ici qu'une liste est bien plus qu'une paire pointée, puisque il faut également que son fils droit soit ou bien une liste à son tour, ou bien le terme [].

`int` **p4is_var**`(P4TERM T);`

Teste le terme *T*.
Retourne `TRUE` si T est variable, c'est à dire que son domaine de valeurs possibles n'est pas restreint à une seule valeur, retourne `FALSE` sinon.

Notes sur les testeurs de termes : Il n'existe pas de fonction **p4is_list**(). Voir à ce sujet la remarque associée à la fonction **p4is_dot**().

10.1.6 Convertir des termes Prolog IV en données C

Toutes ces fonctions sont de la forme :

cccc **p4val_** *cccc*`(P4TERM T)`

ou de la forme :

`P4ERR` **p4** *pppp*_**to_** *cccc*`(P4TERM t,` *cccc* `*val)`

Ce sont deux interfaces d'une même fonctionnalité. La première peut être utilisée là où l'on est sûr qu'il n'y a pas d'erreur, ou qu'on ne souhaite tester s'il y a erreur que plus tard (elle est contenue dans la variable globale `p4errno`), la seconde retourne la valeur de cette variable `p4errno`. *Pppp* représente un type prolog, et *cccc* un type C.
Toutes ces fonctions permettent de récupérer sous une forme utilisable en C la valeur du terme *T*. Si `p4errno` est différente de 0, c'est que *T* n'est pas un terme ou qu'il n'est pas du type approprié pour effectuer l'opération d'extraction vers C.

`long int` **p4val_lint**`(P4TERM T);`
`P4ERR` **p4integer_to_lint**`(P4TERM T, long int *val);`

Récupère et retourne la valeur du terme *T* de type entier, laquelle doit tenir dans le type C "long int".

`unsigned long int` **p4val_ulint**`(P4TERM T);`
`P4ERR` **p4integer_to_ulint**`(P4TERM T, unsigned long int *val);`

Récupère et retourne la valeur du terme *T* de type entier, laquelle doit tenir dans le type C "unsigned long int".

`double` **p4val_double**`(P4TERM T);`
`P4ERR` **p4float_to_double**`(P4TERM T, double *val);`

Récupère et retourne la valeur du terme *T* de type "nombre flottant".

`double` **p4val_as_double**`(P4TERM T);`

Soit *T* un terme numérique, cette fonction effectue une conversion en nombre flottant double, lequel est retourné.

`P4SYMBOL` **p4val_symbol**`(P4TERM T);`
`P4ERR` **p4atom_to_symbol**`(P4TERM T, P4SYMBOL *val);`

Récupère et retourne la valeur du terme *T* de type "atome", laquelle est un symbole.

`P4SYMBOL` **p4val_fsymbol**`(P4TERM T);`
`P4ERR` **p4functor_to_symbol**`(P4TERM T, P4SYMBOL *val);`

Récupère et retourne la valeur du noeud du terme *T* de type "foncteur", laquelle est un symbole.

10.1.7 Parcourir les arbres et listes Prolog IV

`P4TERM` **p4car**`(P4TERM Liste);`
`P4TERM` **p4cdr**`(P4TERM Liste);`

Ces deux primitives récupèrent respectivement le membre gauche ou droit de la paire pointée Liste.

Exemple sur les listes :

Etant donné un terme *tListe* que l'on sait être une liste, le parcourir se fait de la façon suivante :

```
while ( ! p4is_nil(tListe) )
    {
     faire quelque chose avec le terme  p4car(tListe)
    tListe = p4cdr(tListe); /* passer a la suite */
    }
```

Note : Une liste est par définition finie et se termine par [] (c.a.d. nil).

`int` **p4get_arity**`(P4TERM T);`

Récupère l'arité du terme *T*. On rappelle que l'arité d'une feuille est 0 et que l'arité d'une paire-pointée est 2. Il est retourné -1 et en cas d'erreur.

`P4TERM` **p4get_arg**`(P4TERM T, int i);`

Récupère le terme placé en *i*ème position dans la liste des fils du terme *T*.

10.2 Appeler C depuis Prolog IV

Cette section décrit les moyens mis à la disposition du programmeur Prolog IV d'appeler des programmes ou sous-routines écrits en C.
Ces trois nouvelles fonctions permettent d'appeler du code C depuis Prolog IV.

- **p4declare_function** pour associer une fonction C à un prédicat donné.
- **p4get_parameter** pour récuperer les paramètres de l'appel Prolog IV.
- **p4unify_parameter** pour unifier un paramètre et un argument de l'appel Prolog IV.

10.2.1 Enregistrer une fonction C

`P4SYMBOL` **p4declare_function**`(char *name, int arity, P4FUNCRET cfunction)`

Sert à enregistrer une fonction C auprès de Prolog IV, de sorte qu'elle soit activée chaque fois qu'un appel prolog d'indicateur de prédicat *name/arity* est lancé.
La fonction dont le nom est passé par l'argument *cfunction* doit être declaré comme suit :

```
P4FUNCRET mafonction(void);
```

et doit retourner les valeurs `P4SUCCESS` ou `P4FAIL` pour indiquer à Prolog IV un succès ou un échec.

10.2.2 Récupérer un paramètre de l'appel Prolog IV

`P4TERM` **p4get_parameter**`(int rang);`

Rang est un entier entre 1 et l'arité du prédicat (arité qui n'est pas testé à l'heure actuelle). Si une erreur est détectée, `NULL` est retourné, sinon, le terme correspondant au rang*ième* argument de l'appel courant est retourné.

10.2.3 Unifier un terme et un paramètre de l'appel Prolog IV

`int` **p4unify_parameter**`(int rang, P4TERM T);`

Il s'agit de l'unification d'un terme donné (et probablement construit par le programmeur), et l'actuel rang*ième* paramètre de l'appel courant. La plupart du temps, c'est une variable libre qui a été passée comme paramètre mais ce n'est pas requis.

- Si l'unification réussi, 0 est retourné.
- Si l'unification échoue, -1 est retourné.
- Sinon, une erreur a été détectée, l'erreur (sous la forme d'un nombre positif) est retournée.

Un exemple est donné en appendice (le fichier `example6.c`).

10.3 Appeler Prolog IV depuis TCL

La seule possibilité actuellement offerte au programmeur consiste en l'appel d'un but Prolog IV. Il faut pour cela que Prolog IVsoit en attente de lecture de but.

conn:send `Texte`

Cette procédure TCL envoie *Texte* dans l'entrée console de Prolog IV. Si le texte envoyé (et qui n'est pas affiché dans la console), est lu par le lecteur de buts, ce texte sera exécuté comme s'il avait été tapé dans la console et les éventuelles solutions s'y afficheront.
Par exemple :

```
conn:send "X = 1515 ; X = 1789. "
```

affichera dans la console de Prolog IV :

```
>>
X = 1515;

X = 1789.

>>
```

10.4 Appeler TCL depuis Prolog IV

L'appel d'une procédure TCL peut se faire depuis Prolog IV. Il y a deux façons de faire :

1. Soit on effectue un appel sans attente de résultat,
2. soit on effectue un appel dont on attend le résultat.

La différence entre les deux tient évidemment à cette attente du résultat. Prolog IV et TCL fonctionnant en général dans deux processus séparés, cette attente peut avoir son importance dans le déroulement d'une exécution.
Dans ces deux primitives, le script TCL utilisé est noté sous la forme d'une liste de termes telle que l'écriture des éléments au moyen de la primitive **write**/1 forme un script valide. Un script est limité à un appel avec arguments ; il n'est donc pas possible d'utiliser le séparateur de commande ";". Bien qu'il soit possible d'y imbriquer des appels (avec la construction [...]), cette manière de faire n'est pas conseillée. Chaque script est exécuté au niveau global.

wsend(L) Effectue un appel à TCL sans attente de résultat. Voici des exemples d'appel depuis Prolog IV de script TCL :

```
>> wsend(['toplevel .topwin']).
```

Envoie le script `{toplevel .topwin}` .

```
>> W = '.topwin', T = 'Ma fenetre',
   wsend(['toplevel ', W, ' -title "', T,'"']).
```

Envoie le script `{toplevel .topwin -title "Ma fenetre"}` .

Notes : ces deux scripts commandent la création (sous TCL/TK) d'une fenêtre. Le second script lui donne en plus un titre.

wsend2(L,R) Effectue un appel à TCL, mais avec attente de résultat. Ce résultat, envoyé depuis TCL sous la forme d'un texte, doit avoir une syntaxe de terme suivi d'un point et d'un espace, de sorte qu'il puisse être lu par les primitives de lecture de la famille **read**/*n*.
Voici des exemples :

```
>> wsend2(['set toto "success('[info tclversion]'). "'], R).
R = success('7.6').
```

La même chose, mais en passant par un appel à une procédure TCL :

```
proc toto {} {
    return "success('[info tclversion]'). "
}
```

L'appel depuis Prolog IV est alors :

```
>> wsend2(['toto'], R).
R = success('7.6').
```

En cas d'erreur pendant le script, on obtiendrait comme résultat :

```
>> wsend2(['titi'], R).
R = tcl_error('invalid command name "titi"').
```

Bien que la forme du terme retourné soit libre (sauf cas d'erreur), on posera par convention qu'elle est l'une des suivantes :

- `tcl_error('message d'erreur de TCL')` C'est une erreur qui s'est produite pendant l'exécution du script sous TCL.
- `success(Terme)` Si on souhaite retourner *Terme*.
- `fail` Si on souhaite signaler un échec.

Note : Il est très important de retourner la représentation écrite *correcte* d'un terme.

10.5 Appendice A (Erreurs de l'interface Prolog IV-C)

P4E_FAIL échec (ce n'est pas vraiment une erreur)

P4E_NO_ERROR pas d'erreur

P4E_BAD_TYPE le terme n'a pas le type attendu pour l'opération demandée

P4E_BAD_INDEX l'index est hors bornes (pendant l'accès à une chaîne ou une liste)

P4E_NO_MORE_SPACE pas assez de mémoire pour construire le terme

P4E_BAD_ARG_RANK l'index est hors bornes (pendant l'accès à un argument)

P4E_NOT_A_TERM `NULL` ou pointeur de terme invalide

P4E_UNREPRESENTABLE la valeur de ce terme ne peut être representée par le type C

P4E_NOT_TERMS le tableau de termes de la fonction **p4vmake_*xxx***() est `NULL`

P4E_NULL_TERM `NULL` passé en paramètre au lieu d'un `P4TERM`

P4E_BAD_ARITY ce nombre ne peut être une arité (doit être entre 1 et 1000)

10.6 Appendice B

Le fichier C "doubler.c" et le fichier Prolog IV "doubler.p4" :

```
/* ---------------------------------------------------------------------------
 * -- doubler.c --
 * Exemple complet d'appel a Prolog IV pour transformer un tableau C d'entiers
 * en un tableau C comportant ces entiers multiplies par deux.
 * - compilation d'un fichier Prolog IV
 * - conversions de donnees dans le sens C -> Prolog IV
 * - invocation d'un but Prolog IV
 * - conversions de donnees dans le sens Prolog IV -> C
 */

#include <stdio.h>

#include "p4term.h"

typedef long int lgint;

int nbelem = 6; /* Nombre d'elements dans la liste */
lgint matrice[] = { /* Une liste de nombres */
     1, 10, -3,
```

```
    -1,  4,  8
    };

P4SYMBOL doubler_liste; /* Le symbole (cote prolog) du */
                    /* predicat prolog a appeler   */

/* --------------------------------------------------------------------------------
 * Transforme un tableau C d´entiers C (Tab) en une liste prolog d´entiers
 * ( cree dans l´espace prolog )
 */
P4TERM matrice_de_C_vers_P4(lgint Tab[])
    {
    P4TERM tListe;
    int i;
        /* Il est plus simple de construire une liste a partir de la fin */
    tListe = p4make_nil();
    for ( i = nbelem-1 ; i >= 0 ; i-- ) {
        tListe = p4make_dot(p4make_lint(Tab[i]), tListe);
        }
    return tListe;
    }


/* --------------------------------------------------------------------------------
 * Transforme une liste prolog d´entiers en un tableau C d´entiers C (Tab)
 * ( On supposera le tableau C assez grand ! )
 */
void matrice_de_P4_vers_C(P4TERM tListe, lgint Tab[])
    {
    int i = 0;
    while ( ! p4is_nil(tListe) ) {
        Tab[i++] = p4val_lint(p4car(tListe));
        tListe = p4cdr(tListe);
        }
    }


/* --------------------------------------------------------------------------------
 * Cette procedure comporte l´appel effectif au predicat prolog
 * prenant une liste en premier argument et retournant une liste
 * dans son second argument.
 * On y trouvera egalement les conversions de donnees necessaires
 * de C vers Prolog IV et inversement.
 */
void appel_doubler_matrice(lgint matrice[], lgint resultat[])
    {
    P4TERM tBut, tRes;
    int status;

        /* On se prepare a lancer un but */
    p4new_session();

        /* On construit le terme:  doubler_liste([...], Res) (arite 2) */
    tBut = p4make_functor(2, doubler_liste,
                matrice_de_C_vers_P4(matrice),
                tRes = p4make_var()
                );
    p4make_call(tBut);  /* On l´installe comme but */
```

```
    status = p4next_solution();  /* On lance la machine Prolog */

    if ( status == P4SESSION_SOLUTION ) {
            /* Si c´est une solution (ni echec ni erreur),
             * on recupere le resultat (l´argument Res)
             */
        matrice_de_P4_vers_C(tRes, resultat);
        }
        /* on ne cherche pas les eventuelles autres solutions */
    p4finish_session();  /* on en a termine avec le but courant */
    }


/* -------------------------------------------------------------------------------
 * Le programme principal :
 * - on initialise Prolog IV
 * - on compile le fichier "doubler.p4"
 * - on appelle doubler_liste(matrice_C, resultatC)
 * - on affiche resultatC
 * - on appelle doubler_liste(resultatC, resultatC)
 * - on affiche resultatC
 */
void main(int argc, char *argv[])
    {
    int i;
    int resultat[6];

    p4init(argc, argv);                              /* Initialise Prolog IV */
    p4compile("doubler.p4");  /* On charge le programme prolog de ce fichier */
    p4cstring_to_symbol("doubler_liste", &doubler_liste);

    appel_doubler_matrice(matrice, resultat); /* L´appel (indirect) a prolog */

    for ( i = 0; i < nbelem; i++ )          /* On affiche le resultat */
        printf(" %d ", resultat[i]);
    printf("\n");

    appel_doubler_matrice(resultat, resultat);     /* Un autre appel a prolog*/

    for ( i = 0; i < nbelem; i++ )          /* On affiche le resultat */
        printf(" %d ", resultat[i]);
    printf("\n");
    }

/* -- EOF doubler.c --------------------------------------------------------- */
```

Le fichier Prolog IV "doubler.p4" :

```
/* Le programme Prolog IV */
%
% doubler_liste(L1, L2)
% retourne dans L2 la liste des elements x 2 de la liste L1.
%
doubler_liste([], []).
doubler_liste([E|L], [2*E | L1]) :-
    doubler_liste(L, L1).

% -- EOF doubler.p4 ---------------------------------------------------------
```

10.7 Appendice C

Un autre exemple de règle prédéfinie appelée depuis C : compile/1

```
/* --------------------------------------------------------------------------------
 * Un appel a la regle compile(nomDeFichier)
 * ou nomDeFichier est une chaine C
 */
void appel_compile(char *filename)
    {
    P4TERM tBut, tRes;
    P4SYMBOL symb_filename, symb_compile;
    int status;

    p4cstring_to_symbol(filename, &symb_filename);
    p4cstring_to_symbol("compile", &symb_compile);

        /* On se prepare a lancer un but */
    p4new_session();

        /* On construit le terme: compile(f) (d´arite 1) */
    tBut = p4make_functor(1, symb_compile,
                p4make_atom(symb_filename)
                );
    p4make_call(tBut); /* on l´installe comme but */
    status = p4next_solution(); /* on lance la machine Prolog */

    if ( status == P4SESSION_SOLUTION ) {
        /* si c´est une solution (ni echec ni erreur)
         */
        }
        /* on ne cherche pas les eventuelles autres solutions */
    p4finish_session(); /* on en a termine avec le but courant */
    }
```

10.8 Appendice D

Exemple montrant comment appeler une fonction C depuis un appel Prolog IV (lequel est appelé aussi depuis C).

```
/* -- example6.c --
 * Appeler C depuis Prolog IV
 */

#include <stdio.h>
#include <stdlib.h>
#include <math.h>

#include "p4term.h"

P4SYMBOL symbol_of_p4call;
P4SYMBOL symbol_of_my2degree;


/* Un nouveau predicat, pour resoudre des equations du second degre.
 * my2degree(A,B,C, X1,X2) A,B and C etant des nombres.
 */
P4FUNCRET my2degree_cproc()
{
P4TERM tA, tB, tC, tX1, tX2;
double dA, dB, dC, dX1, dX2;

p4firsterrno = 0;
tA = p4get_parameter(1);
tB = p4get_parameter(2);
tC = p4get_parameter(3);

dA = p4val_as_double(tA);
dB = p4val_as_double(tB);
dC = p4val_as_double(tC);

if ( p4firsterrno ) goto error;

if ( dA == 0 ) { /* 1er degre */
if ( dB == 0 ) { /* 0ieme degre */
if ( dC == 0 ) { /* nombre oo de solutions */
P4SUCCESS;
}
else { /* pas de solutions */
P4FAIL;
}
}
else {
dX1 = -dC/dB;
tX1 = p4make_rational_from_double(dX1);
p4unify_parameter(4, tX1);
P4SUCCESS;
}
}
else {
double delta = dB*dB - 4*dA*dC;
if ( delta >= 0 ) {
dX1 = (-dB - sqrt(delta) )/(2*dA);
```

```
dX2 = (-dB + sqrt(delta) )/(2*dA);
tX1 = p4make_rational_from_double(dX1);
if ( 0 != p4unify_parameter(4, tX1) ) P4FAIL;
tX2 = p4make_rational_from_double(dX2);
if ( 0 != p4unify_parameter(5, tX2) ) P4FAIL;
P4SUCCESS;
}
else
P4FAIL;
}

error:
/* pas de nombre ou mauvaise arite */
P4FAIL;
}

/*============================================================================
 Pour enrober un appel a prolog.
 ============================================================================*/

/*
 * Appel my2degree(A,B,C, X1,X2) avec, par exemple A=1,B=0,C=-1
 * (on doit alors avoir comme solutions 1 et -1)
 *  Obtient les solutions et les imprime.
 */
void call_0(long A, long B, long C)
{
P4TERM tBut, tX1, tX2;
int status;
/* Prepare Prolog IV a l´appel d´un but */
p4new_session();
/* construit le terme :  my2degree(A,B,C, X1,X2) */
tBut = p4make_functor(5, symbol_of_my2degree,
p4make_lint(A),
p4make_lint(B),
p4make_lint(C),
tX1 = p4make_var(),
tX2 = p4make_var()
);
p4make_call(tBut); /* Installe ce terme comme but */
status = p4next_solution(); /* demarre la machine prolog */
if ( status == P4SESSION_SOLUTION ) {
/* c´est une solution (pas d´echec ni d´erreur),
 * obtient le resultat
 */
char bX1[30], *sX1 = bX1;
char bX2[30], *sX2 = bX2;

if ( p4is_var(tX1) ) /* oo de solutions */
sX1 = "<Any>";
else
sprintf(bX1, "%le", p4val_as_double(tX1));
if ( p4is_var(tX2) ) /* oo de solutions */
sX2 = "<Any>";
else
sprintf(bX2, "%le", p4val_as_double(tX2));
printf("For my2degree(%ld,%ld,%ld, X1,X2), I got X1=%s, X2=%s\n",
A, B, C, sX1, sX2);
```

```
}
else {
printf("For my2degree(%ld,%ld,%ld, X1,X2), I got no solution\n", A,B,C);
}
/* ici, plus de solution, ou erreur... */
p4finish_session(); /* on "nettoie" la machine prolog */
}

/*
 * main procedure
 */
void main(int argc, char *argv[])
{
p4init(argc, argv); /* initialise Prolog IV */
p4cstring_to_symbol("p4call", &symbol_of_p4call);
p4cstring_to_symbol("my2degree", &symbol_of_my2degree);

p4declare_function("my2degree", 5, (P4FUNCRET)my2degree_cproc);

call_0(1,0,-1); /* appel indirect a "my2degree(1,0,-1,X1,X2)" (1,-1)*/
call_0(1,1,-6); /* appel indirect a "my2degree(1,1,-6,X1,X2)" (2,-3)*/
call_0(9,9,2); /* appel indirect a "my2degree(9,2,2,X1,X2)" (1/3,2/3)*/
call_0(0,5,2); /* appel indirect a "my2degree(0,5,2,X1,X2)" (-0.4,X)*/
call_0(1,1,1); /* appel indirect a "my2degree(1,1,1,X1,X2)" (none)*/
call_0(0,0,0); /* appel indirect a "my2degree(0,0,0,X1,X2)" (X,X)*/

printf("Au revoir !\n");
p4exit(0);
}

/* -- EOF example6.c ----------------------------------------------- */
```