Prolog en quelques lignes

1 Introduction

Un programme Prolog est constitué d'une suite d'affirmations : chaque affirmation est sous forme d'une clause possèdant *au plus* un littéral positif et quon appelle clause de Horn. On distingue :

- les *faits* qui sont des clauses de Horn positives avec un seul littéral qui est positif.
- les *règles* ou clauses de Horn strict $p \vee \neg q_1 \dots q_n$ s'écrit $p : -q_1, \dots q_n$
- les *buts* ou questions qui sont des clauses de Horn négatives (sans littéral positif)

Dans un premier temps,nous considérons des programme Prolog sans but, purement descriptif. Voici un exemple de programme Prolog.

```
% fichier essai
% exemples de faits
bonjour.      % il faut un point pour terminer l'assertion
toto.

humain(alice).  % on pose que le prédicat humain est vrai pour alice.
                % les noms de constante doivent commencer par une minuscule
humain(marius).
humain(marc).
humain(olive).

memechose(fouiller, chercher).
memechose(fouiller, farfouiller).
memechose('bon chat', 'bon rat').
memechose(X,X).  % quelque soit la valeur de X (qui est une variable), memechose
                 % est vrai pour (X,X)
objet(_).        % _ = variable anonyme

parentde(marc, olive).
absent(marc).
absent(olive).
absent(hector).

% exemples de règles
presentation :- bonjour.  % :- se lit "si"; l'assertion se termine par un point

enfantde(X,Y) :- parentde(Y,X).

present(X) :- humain(X),       % la virgule se lit "et"
              not(absent(X)).   % négation par l'échec.
```

Un tel programme doit être saisi à l'aide d'un éditeur de texte standard (une bonne idée sur `turing` est d'utiliser `kate` qui permet de faire facilement des aller-retours entre l'éditeur et l'interprète). Plutôt qu'un programme dans le sens où on l'entend habituellement, il s'agit d'une description logique d'une situation. Cette description doit ensuite être chargée dans un interprète Prolog. L'interprète Prolog installé sur `turing` s'appelle SWI-prolog et on le lance en tapant dans un shell la commande `pl`. Vous pourrez trouver d'autres interprètes Prolog opensource pour diverses plateformes, les fonctionnalités et la syntaxe de base est plus ou moins normalisée. La norme de fait est le Prolog d'Edimbourg. Une fois l'interprète lancé, un message d'invite est affiché, habituellement, il s'agit de `?-`, indiquant que l'utilisateur peut poser une question, c'est-à-dire demander à l'interprète de démonter ou d'essayer de satisfaire un but. Par exemple à la question :

```
?- halt.
```

l'interprète s'arrête. À la question :

```
?- help(abolish).
```

l'interprète donne toute l'aide possible sur le prédicat prédéfini abolish.

Notre premier but consiste ici à charger le fichier précédent :

```
?- consult(essai).
```

On peut ensuite proposer des buts propres au programme `essai` comme, par exemple :

```
?- bonjour.
yes

?- salut.
no

?- humain(alice).
yes

?- humain(paul).
no

?- humain(X).
 X = alice ;
 X = marius ;
 X = marc ;
 X = olive ;
no

?- halt.
```

Remarque : l'interprète prolog de SWI fournit des facilités d'édition comparables au shell `bash` : autocomplétion (avec `tab`, rappel des commandes et édition avec les touches directionnelles).

Lorsque plusieurs résultats sont possibles, on force l'interprète Prolog à chercher les autres solutions en tapant un point virgule. On arrête l'énumération des réponses en tapant un retour chariot simple (`Enter` ou `Return`). On sort de l'interprète en tapant le but `halt`, sans oublier le point pour terminer la requête.

2 Prolog manipule des termes et des clauses

2.1 Termes

Contrairement à ce que nous avons vu en cours à propos de la logique du premier ordre, Prolog ne fait pas de distinction entre les termes fonctionnels et les termes prédicatifs. Un terme peut être :

- une variable, son nom commence alors par une majuscule ou underscore ;
- une constante, en Prolog, on dit un *atome* à ne pas confondre avec la notion d'atome vue en logique. Un atome peut comporter des caractères spéciaux, il doit alors être délimité avec des cotes (`'`). Exemples : `truc`, `'ceci est un atome'`, `'=fonc='` ;
- un terme composé construit à l'aide d'un symbole fonctionnel ou prédicatif, qu'on appelle dans la terminologie Prolog un *foncteur*, et d'arguments qui sont eux-mêmes des termes. Exemples : `humain(alice)`, `g(f(a), X, h(c,alice))`. On appelle foncteur principal le symbole fonctionnel en tête du terme en cas de notation préfixée. Note : on peut aussi utiliser des notations infixées à condition de déclarer les foncteurs en question à l'aide de la directive `op` (voir plus bas).

Il convient de noter qu'un symbole fonctionnel n'a pas de signification intrinsèque, et on peut surcharger les opérateurs en changeant les types ou le nombre des arguments. On prend l'habitude de désigner un foncteur Prolog en donnant son nom et son arité (par exemple, `op/3` désigne le foncteur `op` avec trois arguments).

2.2 Clauses

Une clause Prolog est une clause de Horn qui peut être réduite à un littéral, positif ou négatif, être une conjonction de buts ou être stricte. Dans ce dernier cas, elle s'écrit :

`tête :- littéral`$_1$`, littéral`$_2$`, ..., littéral`$_n$`.`

Les buts sont habituellement utilisés lors de l'interrogation d'un programme Prolog. On peut aussi les employer pour indiquer des directives d'interprétation ou pour lancer des requêtes au chargement d'un fichier. Lorsqu'on est sous l'interprète Prolog, on pose une conjonction de buts au prompt (`?-`) en séparant les différents littéraux par une virgule. Dans un programme Prolog, un but est naturellement défini comme étant une clause sans tête qui commence donc par l'opérateur `:-`. Dans certains interprètes, c'est l'opérateur `?-` dans le source du programme qui est utilisé. Exemples :

```
:- op(500, xfx, est_le_pere_de).  % définit un opérateur infixe de précédence 500

:- [source1, source2, principal].  % charge les fichiers dont le nom est indiqué

:- main.  % lance la requête main, qui déroulera ensuite un pgm
```

3 égalité

L'égalité entre deux termes Prolog peut prendre différentes significations. Il s'agit soit de comparer syntaxiquement deux termes, soit de les rendre semblables (unification et/ou interprétation). Cela correspond aux opérateurs Prolog suivants :

`=`	:	unification des termes à droite et à gauche ;
`==`	:	identité syntaxique ;
`=:=`	:	égalité arithmétique (évaluation des deux termes et comparaison) ;
`is`	:	instanciation arithmétique ;
`not(_ = _)`	:	non unifiablilité ;
`\==`	:	différence syntaxique ;
`=\=`	:	différence arithmétique.

Ces comparaisons sont très différentes, comme on peut le voir dans la session suivante :

```
?- X = 3, X = Y.
X = 3, Y = 3

?- X = 3, X == Y.
no

?- X = 3, Y = 2+1, X = Y.
no

?- X = 3, Y = 2+1, X =:= Y.
X=3, Y=2+1
```

4 Arithmétique

Les expressions arithmétiques Prolog sont des termes comme les autres. Leur évaluation est provoquée par l'opérateur `is` comme nous l'avons vu plus haut. Exemples :

```
?- X = 3 + 4.
X = 3+4

?- X is 3+4.
X=7
```

Les opérateurs arithmétiques usuels sont les suivants :

`+` : addition ;
`-` : soustraction ;
`*` : produit ;
`/` : division réelle ;
`//` : division entière ;
`mod` : reste dans la division euclidienne ;

Les opérateurs de comparaison suivants provoquent également l'évaluation des termes arithmétiques :

`> , < , >= , =< , =:=` et `=\=`

5 Contrôle du retour arrière

On peut inhiber le *backtracking* (ou retour arrière) en Prolog en utilisant un opérateur de contrôle (et donc extra logique) nommé *coupure* ou *cut*. Attention, il ne faut pas confondre cet opérateur avec la règle d'inférence vue en cours de logique. Cet opérateur peut être utilisé pour améliorer l'efficacité de l'exploration de l'arbre des buts, on parle alors de *cut vert* ou pour changer la signification déclarative d'un programme, on parle alors de *cut rouge*. Exemples :

% cut vert
```
% définition d'une fonction en escalier
f(X,0) :- X < 3, !.
f(X,2) :- 3=<X, X<6, !.
f(X,4) :- 6 =< X.
```

% cut rouge
```
% définition d'une fonction en escalier
f(X,0) :- X < 3, !.
f(X,2) :- X<6, !.
f(X,4).
```

Pour bien voir la différence entre ces deux programmes, tracer les buts :

```
?- f(1,Y), 2<Y.
```

avec les deux programmes tels quels, puis avec les deux programmes modifiés en enlevant les coupures. Que se passe-t-il si on change l'ordre des clauses ?

Autre exemple :

% appartenance
```
appartient(X,[X|_]) :- !.
appartient(X,[_|L]) :- appartient(X,L).
```

Remarque : l'utilisation de coupures réduit fortement les propriétés de non déterminisme, de réversibilité et de déclarativité qu'on pourrait attendre de prédicats écrits en Prolog.

6 échec, négation par l'échec

Le prédicat prédéfini `fail` échoue toujours. Il peut être vu comme l'équivalent du but `1=2` par exemple.
Coupure et échec sont utilisés conjointement pour définir un succédané de négation logique : la négation par l'échec. En Prolog, cette négation est définie par l'opérateur `not` qui se comporte de la manière suivante : `not(But)` réussit si `But` échoue. Les variables non instanciées de `But` au moment de l'appel à `not` ne sont donc pas instanciées en cas de réussite. `not` peut être défini en Prolog même :

```
not(B) :- call(B), !, fail.
not(_).
```

Une grande différence avec la négation en logique classique réside dans ce qu'on appelle l'hypothèse du monde clos : tout ce qui n'est pas prouvable est réputé faux. C'est ainsi qu'avec notre exemple plus haut :

```
?- not humain(pascal).
 yes
```

On ne peut pas non plus exhiber de contre-exemples :

```
?- not (not humain(X)).
 X = _342 ... ou yes ... ça dépend des interprètes Prolog !
```

Le but réussit mais sans donner de valeurs à `X`.

Ceci peut produire des écarts profonds avec la logique, comme avec le programme :

```
r(a).
q(b).
p(X) :- not r(X).
```

pour lequel on obtient le comportement suivant :

```
?- q(X), p(X).
 X = b

?- p(X), q(X).
 no
```

7 Quelques prédicats prédéfinis standards

7.1 Entrées/sorties

consult/1 : charger un programme. On peut aussi utiliser une liste pour charger un ou plusieurs programmes dans l'interprète (`[nom_prog1,nom_prog2, ..., nom_progn]`). Attention : si on charge plusieurs fois le même programme, on duplique les clauses.

reconsult/1 : re-charger un programme : effacer les anciennes clauses et charger les nouvelles (on peut aussi utiliser une liste en plaçant un `-` devant le nom du fichier `[-nom_prog]`).
Attention ces conventions ont évoluées et les interprètes prolog font habituellement un reconsult avec la commande `[nom_prog1,nom_prog2, ..., nom_progn]` .

read/1 : lire un terme syntaxiquement correct, terminé par un point, sur le flux d'entrée standard.

write/1 : écrire un terme sur le flux de sortie standard.

put/1 : écrire un caractère (c'est le code ASCII qui doit être donné en argument).

get/1 : lire un caractère imprimable (c'est le code ASCII qui est lu) sur le flux d'entrée.

get0/1 : lire n'importe quel caractère.

nl/0 : passage à la ligne.

tab/1 : affiche le nombre d'espaces donné en argument.

see/1 : change le flux de sortie standard

seen/0 : ferme le flux de sortie ouvert avec `see`, et utilise à nouveau le flux de sortie standard.

seing/1 : indique quel est le flux de sortie standard.

tell/1 : même chose pour le flux d'entrée : redirection,

told/0 : fermeture,

telling : et interrogation.

help/1 : fournit de l'aide sur le prédicat prédéfini donné en argument.

7.2 Nature des termes

Il n'y a pas à proprement parler de *types* en Prolog. Le type d'un terme est déterminé d'après sa forme syntaxique. Les interprètes Prolog proposent habituellement un certains nombre de prédicats pour tester la nature des termes. Voici les plus usuels (certains dépendent de l'interpréteur) :

var/1 : `var(X)` réussit si `X` est une variable non instanciée.

nonvar/1 : équivalent à `not var`.

atom/1 : réussit si l'argument est un atome Prolog

integer/1 : réussit si l'argument est un entier

real/1 ou float/1 : réussit si l'argument est un nombre à virgule (et pas un entier).

numeric/1 ou number/1 : réussit si l'argument est un nombre

atomic/1 : réussit si l'argument est un atome ou un nombre

7.3 Manipulation et décomposition des termes

=.. : opérateur infixe appelé *univ*. Si `Term =.. Liste` réussit `Term` est un terme bien formé et `Liste` une liste dont le premier élément est le foncteur principal du terme et les autres éléments les arguments (des termes) de `Term`.

functor/3 : `functor(Term, Fonc, Arité)` réussit en instanciant (dans un sens ou dans un autre) `Fonc` avec le foncteur principal de `Term` et `Arité` avec le nombre d'arguments de `Term`. (On peut aussi s'en servir pour composer des termes où les arguments sont des variables).

arg/3 : `arg(N, Term, ArgN)` extrait le nème argument de `Term`.

name/2 : `name(X, Listcar)` transforme une chaîne (liste) de caractères en un atome ou réciproquement.

7.4 Manipulation de clauses

Les prédicats donnés ci-dessus ne sont pas fournis dans tous les interprètes : vérifier leur existence avant de les utiliser dans un programme. Rappelons que, en principe, l'appel à un prédicat non défini ne provoque pas une erreur mais un échec ...
Note : dans l'interprète de SWI, l'appel à un prédicat non défini provoque une erreur. Il semble que la plupart des interprètes aient évolués dans ce sens.

assert/1, asserta/1, assertz/1 : ajout d'une clause en début (a ou rien) ou en fin (z) de l'espace de travail.

retract/1 : retrait d'une clause.

abolish/1 ou clear/1 ou ... : retrait de toutes les clauses concernant un prédicat. Non standardisé semble-t-il.

call/1 : `call(X)` ou tout simplement `X` lorsque `X` est instancié à un terme syntaxiquement correct, lance le but en question.

clause/2 : `clause(Tete, Corps)` réussit en instanciant `Tete` à une tête de clause du programme, et `Corps` au corps de la clause (la virgule est alors vue comme un opérateur dont la précédence est définie par l'interprète).

listing/0 et/ou listing/1 : liste toutes les clauses (0) ou les clauses se référant au prédicat donné en argument (1).

save/1, load/1 : sauve, charge l'environnement de travail sous une forme semi-compilée rangée dans un fichier dont le nom est donné en argument.

7.5 Manipulation ensemblistes

Il y a trois prédicats classiques de recherche d'ensembles de solutions. Suivant les interprètes, tous ne sont pas présents, mais on peut toujours les reprogrammer en Prolog en utilisant les prédicats de manipulation de clause vus plus haut.

findall/3 : `findall(f(X1,..., Xn), but(..., X1,...,Xn), Liste_f(X1, ..., Xn))` essaie de satisfaire les buts mentionnés en deuxième argument, les instanciations servent à construire des termes de la forme donnée en premier argument et qui sont rangés dans la liste donnée en troisième argument. S'il n'y a pas de solutions (les buts échouent), c'est la liste vide qui instancie le troisième argument.

bagof/3 : presque comme `findall`, mais en cas d'échec des buts en deuxième argument, ce prédicat échoue lui-aussi.

setof/3 : presque comme `bagof`, mais les termes trouvés sont rangés suivant un ordre lexicographique sur les termes et les doublons sont éliminés.