Notes sur le Prolog

Disclaimer : Les quelques notes dispensées ici sont inspirées du cours de Dominique Pastre que vous pourrez retrouver ici (allez clique !). Celui-ci est plus complet et plus détaillé. Vous trouverez aussi en fin de documents des références vers des liens utiles.

1 Rappels et principes

- clause de Horn : clause disjonctive avec au plus un littéral positif. **Exemple** : $(q \vee \neg p)$, $(\neg p \vee \neg q)$ sont des clauses de Horn, $(p \vee q \vee \neg r)$ n'en est pas une. **Rappel** : $p \vee \neg q$ est équivalent à $q \rightarrow p$.
- unification : étant donné deux formules F_1, F_2, on cherche à trouver une assignation de leur variables pour qu'elles deviennent identiques. **Exemple** : on veut unifier $f(x, g(3)) = f(2, y)$. Pour ce faire, il faut définir $x = 2, y = g(3)$.
- principe de résolution : $\{p \vee q, \neg p \vee q\} \vdash q$.
- modus ponens : $\{p \rightarrow q, p\} \vdash q$.

`Prolog` est un langage notamment adapté à la manipulation des prédicats et des relations. On y retrouve ni boucle, ni instruction, ni affectation. En fait, il s'articule autour de faits, de règles de déduction et de questions. Il permet donc de représenter des connaissances. Il utilise entre autre le modus ponens (cas particulier de résolution) et l'unification pour l'inférence. **Pour les TP nous utiliserons** `SWI-Prolog`.

2 Définitions et objets de `Prolog`

Concept	Description	Exemples
constante	nombre, ou chaîne commençant par une minuscule	`196, a, pouet`
variable	chaîne commençant par une majuscule ou _	`X, Arbre, _, _a`
terme	constante, variable, ou symbole fonctionnel appliqué à une liste de termes	`X, a, a + 3`
littéral	symbole de prédicat appliqué à une liste de termes	`aime(X, Y), couple(X, Y), true`
clause	implications ou faits sur des littéraux : - $(p \wedge r) \rightarrow q$ s'écrit `q :- p, r.` (« *q est impliqué par p et r.* »), - le fait « *$p(X)$ est vrai* » s'écrit `p(X).`. *Attention : point . à la fin !*	`c(X, Y) :- a(X, Y), a(Y, X).`, `aime(nathan, rose).`,

Table 1 – Concepts de `Prolog` avec quelques exemples.

3 Programme Prolog, interpréteur SWI−Prolog : un exemple

Un programme Prolog est un ensemble de faits et de règles. En fait les règles à l'instar des faits sont des clauses. Les fichiers portent l'extension .pl et sont écrits via n'importe quel *éditeur* de texte (et non pas un *traitement* de texte). Par exemple, observons les effets de la Saint-Valentin sur un petit groupe de gens. On tient pour acquis que certaines personnes en aiment d'autres. Autrement dit, on aura des faits aime(X, Y). Pour y voir plus clair dans les couples qui peuvent se dessiner, il nous faut décrire logiquement ce à quoi ils correspondent, i.e *« X et Y sont en couple s'ils s'aiment. »* :

$$\forall X,\ Y \quad (\text{aime}(X,Y) \wedge \text{aime}(Y,X) \rightarrow \text{couple}(X,Y))$$

cette implication fera donc l'objet d'une règle couple dans notre programme. Nous pouvons donc écrire le fichier valentin.pl contenant notre code :

```
/* Fichier : valentin.pl */

/* Exemples de faits */
aime(rose, laurent).  % rose aime laurent.
aime(laurent, lise).
aime(nathan, neige).
aime(neige, nathan).
aime(neige, jezabel).

/* Exemple de regles */
couple(X, Y) :- aime(X, Y), aime(Y, X).
```

Pour poser des questions par contre, il faut utiliser l'interpréteur SWI−Prolog. Sous Linux, la commande swipl ouvre ce dernier. Une fois le prompt ?− proposé, il faut se déplacer dans le répertoire où se trouve le fichier, et le charger :

```
?- working_directory(X, 'chemin/du/dossier').
?- consult(nom_du_fichier).  % par exemple : le fichier valentin
```

Maintenant que le fichier est chargé, on peut poser des questions :

```
?- aime(rose, laurent).
true.
?- aime(lise, nathan).
false.
?- couple(X, neige).
X = nathan.
?-  aime(X, jezabel), couple(nathan, X).  % ',' signifie ET
X = neige.
```

On remarque que l'on peut poser une question sur un fait, par exemple aime(rose, laurent). (vrai/faux) ou une résolution couple(X, neige). où Prolog va essayer d'unifier l'expression, i.e. trouver une valeur de X qui rend couple(X, neige) vraie. On quitte l'interpréteur avec

```
?- halt.
```

4 Subtilités du langage

Conjonction et Disjonction Bien qu'on l'ait utilisé, nous ne nous sommes pas vraiment penchés sur l'utilisation du $\wedge$ ou du $\vee$ pour modéliser des prédicats. En effet, une implication peut avoir plus d'une prémisse, c'est le cas de couple(X, Y) :- aime(X, Y), aime(Y, X) par exemple. Pour modéliser ce $\wedge$ donc, on utilise la virgule , :

$$\forall X\ (p(X) \wedge q(X) \wedge r(X)) \rightarrow s(X)$$

peut se modéliser par :

```
s(X) :- p(X), q(X), r(X).
```

On peut également modéliser des disjonctions avec le point-virgule `;` ou en séparant les clauses. Autrement dit :

$$\forall X \ (p(X) \vee (q(X) \wedge r(X))) \rightarrow s(X)$$

peut se modéliser de deux manières :

```
/* 1ere possibilite, separer les clauses */
s(X) :- p(X).
s(X) :- q(X), r(X).

/* 2eme, utiliser le point-virgule */
s(X) :- p(X) ; q(X), r(X).
```

Le `;` peut également servir à énumérer les différentes solutions lorsqu'on pose des questions à l'interpréteur. Par exemple, reprenons notre base de connaissance sur la saint-Valentin. La question `?- aime(X, Y).` a plusieurs solutions possibles

```
?- aime(X, Y).
X = rose,
Y = laurent ;
X = laurent,
Y = lise.
```

Déclaratif et procédural On peut voir un programme `Prolog` sous l'aspect déclaratif ou procédural. Dans le premier cas, plus proche de la logique avec laquelle on décrit le programme, l'ordre des clauses n'a pas d'importance, autrement dit :

```
a(X, Y) :- b(X, Y).
a(X, Y) :- b(X, Z), a(Y, Z).
```

et

```
a(X, Y) :- a(Y, Z), b(X, Z).
a(X, Y) :- b(X, Y).
```

sont identiques puisqu'ils définissent la même chose. Cependant ! D'un point de vue procédural, plus proche de la manière de fonctionner de `Prolog`, ces deux programmes ne sont pas équivalents. En effet, si dans le deuxième cas on essaye de résoudre `a(X, Y)`, puisque `Prolog` lit les clauses dans l'ordre et essaye de résoudre les prémisses une par une dans l'ordre également, on bouclera infiniment (la première prémisse de `a` est `a`, qui appelle `a` etc). Ca ne sera pas le cas dans le premier programme. Aussi, il est intéressant de penser ses programmes d'abord en déclaratif puis les adapter en procédural afin d'éviter les récursions infinies par exemple.

Variable _ Cette variable est anonyme et est utilisée quand la valeur qu'elle peut prendre ne nous intéresse pas ou est quelconque.

Coupe (*« cut »*) La coupe `!` est un littéral permettant de limiter les solutions explorées par `Prolog` lors d'une déduction. Prenons un exemple, `p(X) :- q(X), r(Y), !, s(Z).` et disons que l'on veut résoudre `p(X)`. `Prolog` va lire l'implication est définir à partir de `p(x)` 4 nouveaux buts (conditions à satisfaires/unifier) : `q, r, !, s`. S'il arrive à unifier `q(X)` et `r(Y)` alors la coupe (qui est vraie par défaut) va limiter l'ensemble des solutions possibles pour `p(X)` aux valeurs trouvées pour `q(X)` et `r(Y)` plus les solutions possibles pour `s(Y)`. Dans cet exemple où une seule règle est en jeu, cela revient à fixer `q(X)` et `r(Y)` aux valeurs valides trouvées, et énumérer les valeurs possibles

de `s(Y)` pour rendre `p(X)` vraie. Lorsque plusieurs implications ayant `p(X)` pour tête sont présentes, on s'interdit également toutes utilisation de ces autres règles dès lors que la coupe est atteinte. Plus d'exemples sont donnés au chapitre *« cut »* de [1].

Listes Au delà des atomes, variables, constantes, etc, on peut utiliser des listes notées `[a, b, c, d]` ou `[a | [b, c, d]]`. La deuxième version utilise la distinction tête/queue (ou head/tail). Etant donné le langage, les fonctions sur les listes doivent être définies de manière récursive. On donne quelques exemples :

```
/* dernier/2 : vrai si X est le dernier element de L */
dernier(X, [X]).
dernier(X, [_ | L]) :- dernier(X, L).

/* membre/2 : vrai si X est un membre de L */
membre(X, [X | _]).
membre(X, [_ | L]) :- membre(X, L).

/* taillepair/1 : vrai si L a un nombre pair d'elements */
taillepair([]).
taillepair([_, _ | L]) :- taillepair(L).
```

5 Aparte sur les outils

Un éditeur de texte disponible ici est `Emacs` qui a l'avantage de fournir une coloration syntaxique pour `Prolog`. Voici les instructions pour créer un fichier et activer la coloration :

- ouvrir un terminal, se déplacer dans le répertoire de votre choix,
- lancer `emacs` depuis le terminal,
- une fois `Emacs` ouvert, effectuer `ctrl + x` puis `ctrl + f` et écrire le nom du fichier qui vous intéresse. **Remarque :** si le nom de fichier existe déjà, `Emacs` va charger le fichier existant, sinon il va en créer un nouveau portant le nom donné.
- activer la coloration : `alt + x` puis écrire `prolog-mode`.
- vous avez gagné.

Si toutefois, vous aviez des problèmes avec les éditeurs installés sur les machines, il existe un éditeur/compilateur `Prolog` en ligne `SWISH` disponible ici.

6 Liens utiles

- Learn Prolog Now, `http://learnprolognow.org`
- Cours de Dominique Pastre, `http://www.normalesup.org/~pastre/prolog/cours.pdf`
- Wikilivre Prolog, `https://en.wikibooks.org/wiki/Prolog`
- Documentation SWI-Prolog, `http://www.swi-prolog.org/pldoc/index.html`