

Programmation Logique – TP 4

Un système expert en Prolog

Narendra Jussien

2001 – 2002

1 Introduction

Selon E. Feigenbaum, on peut définir l'*Intelligence Artificielle* comme étant un domaine de l'informatique qui vise à conférer à l'ordinateur un comportement reconnu comme *intelligent* par l'homme. Cette *intelligence* sera reconnue principalement par la manipulation d'information de nature symbolique (par opposition à numérique par exemple).

Les domaines d'application de l'intelligence artificielle sont divers et variés : reconnaissance des formes, de la parole, compréhension du langage naturel, jeux, résolution de problèmes, systèmes experts, démonstration automatique, vision artificielle, génération de plans, traduction automatique, ...

Prolog est un langage particulièrement adapté à tout ce qui touche à l'Intelligence Artificielle. Nous allons le voir aujourd'hui avec le cas particulier des *systèmes experts*.

Un système expert, toujours selon une définition de E. Feigenbaum, est un programme informatique intelligent qui utilise des connaissances et des procédures d'inférences dans le but de résoudre des problèmes d'une difficulté telle qu'ils requièrent une expertise humaine conséquente. Les connaissances nécessaires pour y arriver ainsi que les procédures d'inférences utilisées peuvent être assimilées à une modélisation de l'expertise des meilleurs spécialistes du domaine considéré.

MYCIN est un des tout premiers systèmes experts jamais développés. Il date de 1972. Le but de ce système expert interactif est d'aider les médecins à proposer la meilleure thérapie antimicrobienne pour leur patient atteints d'infections bactériologiques. À partir des symptômes, de l'histoire du patient et de résultats de tests de laboratoire, le système diagnostique la cause de l'infection et propose un traitement selon l'expertise de praticiens des maladies infectieuses (vous trouverez plus d'informations sur MYCIN à partir de <http://www.eas.asu.edu/~drapkin/556/mycin.html>).

La structure générale d'un système expert peut se décomposer en trois grandes parties :

- Une *base de connaissances* qui contient l'ensemble des règles de production du système ;
- Une *base de faits* qui contient des informations sur le problème particulier à résoudre ;
- Un *moteur d'inférences* qui réalise les démonstrations. Ce moteur d'inférences exploite bien évidemment activement base de faits et base de connaissances. De plus, il contient un module d'explications sur le raisonnement mené par le système.

La partie la plus importante dans un système expert, c'est la connaissance. L'extraction de cette connaissance est un travail particulièrement difficile que l'on laisse à des spécialistes : les cognitivistes.

Nous nous intéresserons dans ce TP plus particulièrement au moteur d'inférences. Pour cela, nous considérerons des règles de productions du type : **si** a **et** b **et** c sont vérifiées **alors** d. D'autre part, nous ne considérerons que de la connaissance *d'ordre 0* i.e. où n'intervient aucune variable¹.

Pour le raisonnement, nous allons écrire ce qu'on appelle un moteur d'inférences en *chaînage arrière*. Le principe est simple : l'utilisateur fait une hypothèse sur le résultat qu'il cherche à obtenir et le système essaie de valider cette hypothèse. Pour cela, il regarde si l'hypothèse est conclusion d'une règle et pour cette règle il essaie de valider chacune des prémisses (a, b, et c dans l'exemple précédent). S'il y parvient l'hypothèse est vérifiée. Dans certains cas, l'ordinateur peut être amené à poser des questions à l'utilisateur pour vérifier certaines prémisses (celles qui ne sont ni des faits, ni des conclusions de règles, et donc pour lesquelles il n'a aucune information ...).

Comme exemple de base de connaissances, vous trouverez dans le fichier `~jussien/Fi3/regles.pl` un ensemble de règles inspirées d'un système expert existant : le système NEREIS de reconnaissance d'annélides polychètes (appelés plus communément vers marins).

Le système expert NEREIS permet au vu d'un ver en possession de l'utilisateur de déterminer sa place dans la nomenclature i.e. sa famille, son genre et son espèce².

Voici le contenu du fichier fourni :

```
% La base de connaissances
regle(1,cirres_ventraux,[deux_antennes,pas_de_paragnathes]).
regle(2,cirres_lateraux,[trois_antennes,pas_de_paragnathes]).
regle(3,machoire,[pas_de_paragnathes,pieds_birames,segment_bucal_apode]).
regle(4,pas_de_machoire,[paragnathes,pieds_birames,segment_bucal_avec_pieds]).
regle(5,nereis,[cirres_ventraux,machoire]).
regle(6,micronereis,[cirres_lateraux,machoire]).
regle(7,pas_de_paragnathes,[cirres_tentaculaires,branchies_spiralees,proboscis_complexe]).
regle(8,segment_bucal_apode,[parapodes_posterieurs]).
regle(9,machoire,[pieds_unirames,segment_bucal_invisible]).

% La base de faits
fait(cirres_tentaculaires).
fait(branchies_spiralees).
fait(parapodes_posterieurs).
```

Cet ensemble de règles permet de reconnaître des vers de la famille des *Nereidae* dont le genre est *nereis* ou *micronereis* (conclusions des règles 5 et 6). La règle 5 peut se lire de la façon suivante : si le ver observé possède des cirres ventraux (excroissance sensorielles³) et une mâchoire alors son genre est *nereis*.

Souvent, on ne peut pas observer précisément le segment bucal (où se trouve la mâchoire) d'un ver (c'est trop petit), c'est pourquoi il y a d'autres règles qui permettent de déduire cette information. Par exemple, la règle 3 précise qu'un ver marin possède une mâchoire s'il n'a pas de paragnathes (sorte de griffes pour attraper des proies) et s'il a des pieds biramés (avec deux branches) et un segment bucal apode (sans pieds).

1. *Remarque* : le calcul propositionnel est d'ordre 0, le calcul des prédicats lui est d'ordre 1. Il existe un ordre 0+, il prend en compte ce qu'on appelle les *variables à attributs*.

2. Pour avoir plus d'information sur ce sujet, vous pouvez consulter la page <http://njussien.nexen.net/nereis.html>. Le système expert NEREIS est lui en ligne à <http://www.ima.uco.fr/etudiants/projets/nereis/>

3. On trouve un mini lexique sur le vocabulaire des spécialistes des vers marins à <http://www.ima.uco.fr/etudiants/projets/nereis/lexique.htm>

Ainsi, si l'utilisateur veut vérifier que le ver qu'il observe est bien du genre *neréis*, le système va vérifier chacune des prémisses en demandant éventuellement à l'utilisateur quelques informations lorsqu'il en a besoin.

2 Un moteur d'inférences en chaînage arrière

2.1 Modélisation de la connaissance et outils

Les faits sont modélisés à l'aide du prédicat `fait/1`. D'autre part, les règles sont modélisées à l'aide du prédicat `regle(Numero,Conclusion,ListePremisses)`.

Avant de commencer la réalisation du moteur d'inférences à proprement parler, nous allons commencer par écrire quelques outils de visualisation de la connaissance.

1. Écrire le prédicat `affiche(N)` qui affiche la règle numéro `N`. On pourra utiliser le prédicat prédéfini `format/2`⁴.

```
| ?- affiche(3).
Règle : 3 ->
    si pas_de_paragnathes et pieds_birames et segment_bucal_apode
    alors machoire
```

2. Écrire le prédicat `lf/0` qui affiche tous les faits de la base de connaissances.

```
| ?- lf.
fait : cirres_tentaculaires
fait : branchies_spiralees
fait : parapodes_posterieurs
```

yes

3. Écrire le prédicat `lc(C)` qui affiche toutes les règles de la base de connaissances ayant pour conclusion `C`.

```
| ?- lc(machoire).
Règle : 3 ->
    si pas_de_paragnathes et pieds_birames et segment_bucal_apode
    alors machoire
Règle : 9 ->
    si pieds_unirames et segment_bucal_invisible
    alors machoire
```

yes

4. Écrire prédicat `lp(P)` qui affiche toutes les règles ayant la prémisses `P` dans leur liste de prémisses.

```
| ?- lp(machoire).
Règle : 5 ->
    si cirres_ventraux et machoire
    alors nereis
Règle : 6 ->
    si cirres_lateraux et machoire
    alors micronereis
```

yes

⁴ *Rappel* : vous avez à votre disposition une documentation complète de SICStus Prolog à l'adresse suivante: <http://www.cling.gu.se/datorinfo/sicstus/sicstus3/sicstus3.html>

2.2 Une première version

La résolution de problèmes par chaînage arrière consiste à faire une hypothèse que l'on cherche à valider. Pour ce faire, deux cas sont possibles : l'hypothèse faite est un **fait** connu dans la base de connaissances, ou l'hypothèse est conclusion d'une **règle** de la base de connaissances. Dans le premier cas, la validation est terminée, dans le second il faut alors valider la liste des prémisses de la règle considérée.

Écrire le prédicat `valider(H)` qui valide l'hypothèse selon le principe présenté ci-dessus.

```
| ?- valider(segment_bucal_apode).
yes
| ?- valider(nereis).
no
```

Ce premier moteur d'inférences est vraiment minimal, on peut l'améliorer simplement en proposant de demander à l'utilisateur la valeur de vérité d'un atome lorsque celui-ci n'est ni un fait ni la conclusion d'une règle de la base. On utilisera pour cela le prédicat prédéfini `read/1`. Les réponses acceptées seront : `o`, `oui`, `y`, `yes`, `ok`, `evidemment`, ...

Compléter la définition du prédicat `valider/1` pour prendre en compte ces demandes possibles.

```
| ?- valider(machoire).
La proposition proboscis_complexe est-elle vraie ?
|: oui.
La proposition pieds_birames est-elle vraie ?
|: oui.
yes
```

2.3 Une version un peu plus efficace

Essayons de valider la proposition `nereis`. Voici ce qui se passe :

```
| ?- valider(nereis).
La proposition deux_antennes est-elle vraie ?
|: oui.
La proposition proboscis_complexe est-elle vraie ?
|: oui.
La proposition proboscis_complexe est-elle vraie ?
|: oui.
La proposition pieds_birames est-elle vraie ?
|: oui.
yes
```

Le système pose plusieurs fois la même question. En effet, on peut être amené à redémontrer plusieurs fois la même chose. C'est ce qui se produit ici pour la preuve de `pas_de_parnathes` qui sert à prouver à la fois `cirres_ventraux` et `machoire` dans la vérification de l'hypothèse `nereis`.

Pour remédier à ce problème, nous allons utiliser une possibilité intéressante du langage Prolog, l'ajout d'informations dans la base de prédicats par l'intermédiaire du prédicat `assert/1`. `assert(P)` ajoute le fait `P` dans la base de prédicats.

Compléter la définition de `valider` pour prendre en compte cette possibilité.

3 Les modules d'explications

L'intérêt principal d'un système expert réside dans le fait qu'il est capable d'expliquer précisément le raisonnement utilisé. Il y a deux types d'explications qu'un système expert peut fournir : le *comment* (comment ai-je réussi à démontrer tel résultat?) et le *pourquoi* (pourquoi suis-je en train de demander ceci à l'utilisateur?). Nous allons maintenant nous intéresser à ces deux types d'explications.

3.1 Le *comment*

On veut obtenir le fonctionnement suivant :

```
| ?- valider(nereis).
La proposition deux_antennes est-elle vraie ?
|: oui.
La proposition proboscis_complexe est-elle vraie ?
|: ok.
La proposition pieds_birames est-elle vraie ?
|: evidemment.
yes
| ?- comment(nereis).
* nereis prouvé par la règle 5 car
  * cirres_ventraux prouvé par la règle 1 car
    + deux_antennes a été fourni par l'utilisateur
  * pas_de_paragnathes prouvé par la règle 7 car
    - cirres_tentaculaires est un fait
    - branchies_spiralees est un fait
    + proboscis_complexe a été fourni par l'utilisateur
  * machoire prouvé par la règle 3 car
    * pas_de_paragnathes prouvé par la règle 7 car
      - cirres_tentaculaires est un fait
      - branchies_spiralees est un fait
      + proboscis_complexe a été fourni par l'utilisateur
    + pieds_birames a été fourni par l'utilisateur
  * segment_bucal_apode prouvé par la règle 8 car
    - parapodes_posterieurs est un fait
yes
```

Pour cela, on introduira de nouvelles informations dans la base de connaissances : les faits *deduit(P,N)* qui seront insérés dans la base lorsque P aura été prouvé par la règle N. On distinguera de plus les faits de base (-) et les faits demandés à l'utilisateur (+).

Modifier en conséquence le moteur écrit jusqu'à présent. On commencera par écrire une version du module d'explication *comment* sans indentation avant de passer à la version avec indentation.

3.2 Le *pourquoi*

On cherche maintenant à obtenir le comportement suivant :

```
| ?- valider(nereis).
La proposition deux_antennes est-elle vraie ?
|: oui.
La proposition proboscis_complexe est-elle vraie ?
|: oui.
La proposition pieds_birames est-elle vraie ?
|: pourquoi.
  j'ai déjà prouvé : [pas_de_paragnathes]
```

```

    si pieds_birames est vrai et que je peux démontrer [segment_bucal_apode]
    j'en déduirai machoire par la règle 3
La proposition pieds_birames est-elle vraie ?
|: oui.
yes

```

On ne s'intéresse, dans l'explication de type pourquoi, qu'à la règle en cours d'utilisation.

Pour réaliser le comportement demandé on propose d'ajouter un deuxième paramètre à (entre autres) **valider**. Ce paramètre est un terme fonctionnel qui mémorise l'état courant du calcul que l'on veut transmettre et dont on aura besoin lorsque l'utilisateur demandera pourquoi une question lui est posée :

```
etat(Deja_prouves,A_faire,Conclusion,NumeroRegle).
```

Compléter en conséquence les définitions du moteur d'inférences pour obtenir le fonctionnement désiré.

4 Pour aller plus loin : faits négatifs

La modélisation et la résolution qui ont été proposées jusqu'à présent ont deux défauts. Considérons l'exemple suivant :

```

| ?- valider(nereis).
La proposition deux_antennes est-elle vraie ?
|: oui.
La proposition proboscis_complexe est-elle vraie ?
|: non.
La proposition pas_de_parnathes est-elle vraie ?
|: non.
La proposition cirres_ventraux est-elle vraie ?
|: oui.
La proposition proboscis_complexe est-elle vraie ?
|: non.
La proposition pas_de_parnathes est-elle vraie ?
|: non.
La proposition pieds_unirames est-elle vraie ?
|: oui.
La proposition segment_bucal_invisible est-elle vraie ?
|: oui.
yes

```

On retrouve ici un problème (la redémonstration d'un même résultat) que l'on avait résolu lorsque les réponses apportées étaient positives. On voit ici que le problème reste entier en cas de réponse négative.

D'autre part, si on regarde la base de connaissance de plus près on constate que certaines propositions (ici **parnathes** et **pas_de_parnathes** et aussi **machoire** et **pas_de_machoire**) bien que non liées entre elles du point de vue Prolog, le sont d'un point de vue logique: **pas_de_parnathes = non(parnathes)**.

Proposer une représentation de ces faits *négatifs* et intégrer leur traitement dans le moteur écrit jusqu'à présent.

On obtiendra alors le fonctionnement suivant :

```

| ?- valider(nereis).
La proposition deux_antennes est-elle vraie ?
|: oui.
La proposition proboscis_complexe est-elle vraie ?
|: non.
La proposition non(parnathes) est-elle vraie ?

```

|: non.
La proposition cirres_ventraux est-elle vraie ?
|: oui.
La proposition pieds_unirames est-elle vraie ?
|: oui.
La proposition segment_bucal_invisible est-elle vraie ?
|: oui.

yes

| ?- comment(nereis).

* nereis prouvé par la règle 5 car

+ cirres_ventraux a été fourni par l'utilisateur

* machoire prouvé par la règle 9 car

+ pieds_unirames a été fourni par l'utilisateur

+ segment_bucal_invisible a été fourni par l'utilisateur

yes