UNIVERSITÉ FRANÇOIS - RABELAIS
CAMPUS BLOIS

UFR Sciences et Techniques

Licence S&T – 1°année

Logique pour l'informatique

TD MACHINE

Enseignant **Jean-Yves ANTOINE**
(Jean-Yves.Antoine AT univ-tours.fr)

Programmation logique sous l'environnement SWI-PROLOG

Au cours de ce TP, vous allez travailler avec l'environnement SWI-Prolog sous Windows (SWI-Prolog existe également sur les plates-formes Linux ou MacOS). Il s'agit d'un environnement de programmation non intégré. Vos programmes seront donc écrits et enregistrés à l'aide d'un éditeur de texte séparé (par exemple, bloc-notes sous Windows), puis vous appellerez l'interpréteur SWI-Prolog pour exécuter ces derniers. SWI-Prolog se réduit à une fenêtre d'interrogation de programmes Prolog.

Nous allons décrire les différentes opérations qui peuvent résumer une session type en se basant sur l'exemple de programme que nous allons étudier au cours de cette première séance de découverte. Ce programme décrit un arbre généalogique particulier concernant les rois bretons du Haut Moyen-Age.

1. Princes de Bretagne : découverte de l'environnement SWI-Prolog

Fils de Nominoe, qui se libéra de la suzeraineté franque en battant la carolingien Charles le Chauve à Redon en 845, Erispoë fut, à partir de 851, le premier roi de Bretagne. L'existence de ce royaume se poursuivit jusqu'en 952, date de la mort d'Alain IV. Le Royaume de Bretagne fut alors transformé en un duché très largement indépendant du royaume de France jusqu'à la fin du XV° siècle (mariage d'Anne de Bretagne avec le roi de France). Les bretons associent souvent, à tort, la gratuité de leurs autoroutes à l'ancienne indépendance du duché de Bretagne, qui ne devait aucun droit de péage au roi de France. Par contre, le témoignage de cette autonomie se retrouve dans les ruines des forteresses que le souverain français et son suzerain breton édifièrent tout au long de la frontière séparant leurs domaines respectifs. Celle-ci passait entre Angers et Nantes, la majeure partie de la Loire-Atlantique actuelle faisant partie du duché de Bretagne. Cette séparation perdura après la révolution, et il fallut attendre l'arrivée du Général de Gaulle pour qu'un remodelage régional séparât Nantes de la région Bretagne. Il créait ainsi une improbable région des Pays de Loire … séparée du Val de Loire à proprement parler. Suite aux dernières élections locales en 2004, le conseil régional de Bretagne et le conseil général de Loire-Atlantique ont émis le vœux de voir ce département rattaché à la région Bretagne. Ce rattachement entraînerait une refonte des régions Centre et Pays de la Loire : dans le futur, Blois fera donc peut-être partie d'une région ligérienne à l'identité plus marquée que la région Centre, dont le seul dénominateur commun semble être de ne pas avoir pu être casé quelque part… Mais bon, foin de digressions historiques, entrons dans le vif du sujet…

1.1. Ecriture du programme

Utilisez un simple éditeur de texte ASCII pour rédiger votre programme (Bloc Note Windows par exemple). On trouvera page suivante le programme Prolog correspondant à notre exemple généalogique. Comme tout programme Prolog, il se compose de deux parties :

- les faits, qui décrivent dans notre cas l'arbre généalogique des rois Bretons. Cet arbre se limite ici aux relations *mère* et *père*, introduites par les prédicat d'arité 2 du même nom, ainsi qu'à la spécification du sexe de chaque personne (prédicats *homme* et *femme* d'arité 1).
- les règles, qui vont permettre de définir des relations de parenté un peu plus riches : par exemple, *fils* et *grand-père*.

Pour des raisons de lisibilité, on ne définit qu'une seule clause par ligne.

Ce programme a déjà été partiellement écrit dans le fichier `breizh.pl` que vous pouvez récupérer sur le forum. Copiez ce fichier sur votre compte : à partir de maintenant, vous ne travaillerez plus que sur cette copie

1.2. Exécution — interpréteur PROLOG

De manière générale, une session d'exécution correspond à la succession des étapes suivantes :

1. appel de l'interpréteur ;
2. chargement du programme désiré en mémoire de l'interpréteur SWI-Prolog;
3. exécution : succession de questions posées à l'interpréteur ;
4. sortie de l'interpréteur .

```
/*************************************************************************/
/* FICHIER : breizh.pl                                                   */
/* AUTEUR :  G. Koscielny et J.-Y. Antoine                               */
/* VERSION : 1.3.                                                        */
/* DATE :    15/12/2004                                                  */
/* OBJET :   genealogie royaume de Bretagne                              */
/*************************************************************************/

/******************************** FAITS **********************************/

/*------ Faits : definition des personnages royaux ----------------------*/

homme(conanIVlepetit).
homme(alainlenoir).
homme(henriIer).
homme(hoelIV).
homme(alainIVfergent).
homme(conanIIIlegros).
homme(mathiasII).
homme(hoelIII).

femme(mathilde).
femme(berthe).
femme(havoise).
femme(constance).

/*-------- Faits : definition des relations de parente directes ----------*/

/* pere_de(X,Y) vrai si le pere de X est Y */

pere_de(conanIVlepetit, alainlenoir).
pere_de(constance, alainlenoir).
pere_de(berthe, conanIIIlegros).
pere_de(hoelIV, conanIIIlegros).
pere_de(conanIIIlegros, alainIVfergent).
pere_de(alainIVfergent, hoelIII).
pere_de(mathiasII, hoelIII).

/* mere_de(X,Y) vrai si la mere de X est Y */

mere_de(conanIVlepetit, berthe).
mere_de(constance, berthe).
mere_de(hoelIV, mathilde).
mere_de(berthe, mathilde).
mere_de(alainIVfergent, havoise).
mere_de(mathiasII, havoise).

/***************** REGLES : relations de parente indirectes **************/

/* parent_de(X,Y) vrai si Y est un des parents de X */

parent_de(Fils, P) :- pere_de(Fils, P).
parent_de(Fils, P) :- mere_de(Fils, P).

/* fils_de(P,F) vrai si F est un fils de P */

fils_de(P, Fils) :- parent_de(Fils, P), homme(Fils).
```

Lors du chargement du programme (étape 2), SWI-Prolog vérifie la syntaxe de votre programme. Attention, toutes les erreurs de syntaxe ne sont pas détectées. Par exemple, si une faute de frappe intervient dans le nom d'un prédicat, il y a de fortes chances pour que SWI-Prolog croie qu'il s'agit d'un nouveau prédicat. Si la sémantique de votre programme est erronée, c'est à dire si vous constatez des problèmes lors de l'étape 3, il vous faudra bien entendu revenir sur votre programme et reprendre l'écriture de programme. Cette opération peut-être réalisée sans sortir de l'interpréteur.

1.2.1. Appel de l'interpréteur — L'appel de l'interpréteur se traduit par l'ouverture d'une fenêtre SWI-Prolog et l'affichage dans celle-ci d'un message d'accueil suivi de la ligne de commande de l'interpréteur :

```
1 ?-
```

On reconnaît le début d'une clause but dans la syntaxe Edimburgh. De fait, toute commande est interprétée comme un but ou une question que l'interpréteur doit résoudre. En début de session, le premier but qui nous intéresse est le chargement du programme Prolog concerné.

1.2.2. Chargement d'un programme — Le chargement d'un programme se traduit par la consultation du fichier correspondant. Utilisez pour cela le prédicat prédéfini `consult/1` (d'arité 1) prenant en argument le nom de fichier, donné entre guillemets (terme constant de type *message* vu en cours) :

```
?- consult('breizh.pl').
```

Si votre fichier comporte l'extension *.pl*, il n'est pas nécessaire de la préciser.

En règle générale, vous devez préciser le chemin absolu d'accès à ce fichier, depuis le répertoire racine (H: par exemple) jusqu'à celui de votre fichier. Dans notre exemple :

```
?- consult('H:\prolog\tp0\breizh').          fichier avec une extension .pl
?- consult('H\prolog\tp0\breizh.pro').       fichier avec une autre extension
```

Afin d'éviter une écriture aussi fastidieuse, vous pouvez préciser une fois pour toute le nom du répertoire de travail à l'aide du prédicat prédéfini `chdir/1`. Une fois cette opération effectuée, vous n'avez plus à retaper à chaque fois le chemin d'accès à votre fichier :

```
?- chdir('H:\prolog\tp0').                   positionnement répertoire de travail
?- consult('breizh').                        chargement du fichier sous ce répertoire
```

Si le chargement s'effectue correctement, la réponse est affirmative et l'interpréteur rend la main. Il s'affiche à l'écran :

```
?- consult('breizh').
breizh compiled, 0.00 sec, 24 bytes.
Yes
```

Deux cas d'échec peuvent survenir. Soit le fichier n'existe pas et l'interpréteur répond par la négative (en règle générale, vous avez mal spécifié le nom du fichier ou de son chemin d'accès) :

```
?- consult('breizh').
[WARNING: breizh: No such file]
No
```

Soit le fichier est chargé correctement mais le programme ne respecte pas la syntaxe de Prolog (syntaxe "Edimburgh ") . Il s'affiche alors un message qui précise la localisation (numéro de ligne) de la première erreur rencontrée. Puis l'interpréteur rend la main en répondant par l'affirmative **en ne conservant que les clauses correctes** :

```
?- consult('breizh').
[WARNING: (/a:/prolog/tp0/parente.pl:21)
         Syntax error: Unexpected end of clause]
breizh compiled, 0.00 sec, 0 bytes.
Yes
```

Attention — Du fait de l'indépendance des clauses, les clauses correctes du programme restent compréhensibles par l'interpréteur quelque soient les erreurs par ailleurs. Il est donc possible de travailler avec un programme présentant des erreurs de syntaxe. Attention cependant de ne pas se leurrer : l'absence de certaines clauses incorrectes change totalement la sémantique de votre programme. Il est donc important de corriger ces erreurs avant de poursuivre.

Notez que si vous modifiez votre programme après l'avoir déjà chargé, par exemple lorsque vous avez détecté une erreur d'exécution, vous devez le recharger après sauvegarde : l'environnement n'étant pas intégré, SWI-prolog ne tient en effet compte de vos modifications que si vous faites de nouveau appel au prédicat `consult/1`. Il s'agit là d'une caractéristique propre à la plupart des interpréteurs Prolog.

Une fois le chargement effectué, l'interpréteur a en mémoire l'ensemble des connaissances sur lesquelles il va raisonner. Désormais, on va l'interroger sur ces dernières.

1.2.3. Exécution — A partir d'un programme donné, on peut lancer un nombre infini d'exécutions correspondant à autant de questions. L'interpréteur va y répondre par résolution, en se basant sur les connaissances décrites dans le programme. La forme de la réponse dépendra de la nature de la question (fermée ou ouverte).

Question fermée — Ici, l'interpréteur doit uniquement répondre par l'affirmative ou la négative (question satisfaisable ou contradictoire). Dans le cas d'une réponse négative, on se retrouve directement sous la ligne de commande :

```
?- mere_de(constance, havoise).
No
```

Dans le cas d'une réponse positive, l'interpréteur s'arrête à la première solution obtenue. Dans cet exemple, l'interpréteur sait qu'il n'existe pas d'autre solution (fait) et rend la main.

```
?- mere_de(constance, berthe).
Yes
```

Question ouverte — Dans le cas d'une réponse négative, on retrouve un affichage analogue à celui d'une question fermée contradictoire. Mais dans le cas d'une réponse affirmative, l'interpréteur doit donner l'interprétation des variables qui permet de satisfaire la question (affichage de la solution). L'interpréteur propose ainsi une première solution :

```
?- fils_de(hoelIII, X).

X = alainIVfergent
```

Si vous désirez obtenir les solution suivantes, il vous suffit de taper indifféremment les caractères `r` (pour Redo), `n` (pour New) ou tout simplement un point-virgule, ceci après chaque nouvelle solution. Au contraire, pour arrêter la résolution après une solution, il vous suffit de taper le retour chariot :

solutions affichées en totalité	***demande d'arrêt de l'utilisateur***
`?- fils_de(hoelII, X).`	`?- fils_de(hoelIII, X).`

`X = alainIVfergent ;`
`X = mathiasII ;`
`No`
	`X = alainIVfergent`
	`Yes`

Le cycle d'attente se poursuit ainsi, jusqu'à ce qu'on désire sortir où qu'il n'y aie plus de nouvelle solution : Prolog répond alors par la négative.

1.2.4. Sortie de l'interpréteur

On utilise le prédicat prédéfini `halt/0` qui permet de quitter l'interpréteur et ferme alors la fenêtre SWI-Prolog :

```
?- halt.
```

1.3. A vous de jouer

Vérifiez la bonne exécution de votre programme en posant des questions à Prolog. On cherchera ainsi à vérifier la bonne définition, d'une part des faits du programme, puis des règles. A chaque fois, on emploiera des questions fermées ou ouvertes, mais aussi des questions à réponse positive ou négative pour s'assurer du bon fonctionnement du programme dans toutes les situations.

Nous allons maintenant nous concentrer sur la programmation Prolog proprement dite, en complétant quelque peu le programme utilisé précédemment.

2. Princes, princesses, ducs, duchesses de Bretagne ...

2.1. Nouvelles relations de parentés : extension du programme Prolog

On désire compléter notre programme en définissant de nouvelles relations de parenté au sein de notre arbre généalogique. Définissez pour cela les nouveaux prédicats suivants :

- `fille(X,Y)` qui réussit si X est la fille de Y
- `frere(X,Y)` qui réussit si X est le frère de Y
- `soeur(X,Y)` qui réussit si X est la soeur de Y
- `gdparent(X,Y)` qui réussit si X est un grand parent de Y (grand-père ou grand-mère)

Faites les jeux d'essais nécessaires pour vérifier le bon fonctionnement de votre programme.

2.2. Entrées / sorties formatées

SWI-Prolog fournit des prédicats prédéfinis d'écriture formatée de termes. Ceux-ci sont très utiles pour afficher des messages, les valeurs de variables et lire des valeurs entrées au clavier par l'utilisateur que l'on unifiera ensuite à des variables.

Ecriture — Pour afficher un message ou la valeur d'une variable, vous utiliserez le prédicat `write/1`. Le prédicat write/1 permet d'afficher au choix la valeur d'une constante (un nombre, un atome ou encore une chaîne de caractères qui sera alors placée entre guillemets), ou encore celle d'une variable instanciée :

`write(3).`	affiche la constante " 1 ".
`write(hoel).`	affiche l'atome constant " hoel ".
`write('le roi hoel 3').`	affiche la chaîne constante de caractères " le roi hoel 3 ".
`write(X).`	affiche la valeur de la variable X (si celle-ci a été instanciée par unification).

L'ensemble de ces possibilités d'affichage peut être combiné (prédicat d'arité variable) :

```
write('la reponse est', X)
```

Il existe également un prédicat `nl/0` qui permet de passer à la ligne suivant (nl = new line) pour la suite de l'affichage.

Travail demandé — Modifiez votre programme pour obtenir des réponses formatées en langage naturel à vos questions concernant la relation de parenté `fils_de/2`. Par exemple on désire obtenir un affichage de ce type :

```
?- fils_de(HoelIII, X).
alainIVfergent est le fils de HoelIII
```

3. Programmation récursive en Prolog : première approche

La puissance de Prolog réside avant tout dans son moteur d'inférence basé sur la résolution de Robinson. En particulier, celle-ci permet de déduire de nouvelles connaissances à l'aide de prédicats récursifs. Par exemple, la clause ci-dessous définit les amis d'une personne comme les amis de ses amis :

```
ami(X,Y) :- ami(X,Z), ami(Y,Z).
```

Bien entendu, cette récursivité doit être mise en oeuvre avec soin sous peine de voir la stratégie de résolution Prolog boucler indéfiniment (ce qui serait le cas dans l'exemple ci-dessus, d'ailleurs…). Nous n'avons pas le temps d'étudier en détail les situations de récursivité bloquantes, qui correspondent toutes à une récursivité à gauche. Nous allons nous contenter d'étudier ce problème expérimentalement sur quelques exemples.

1 — On demande de réaliser un prédicat d'arité 2 `ancetre(Kozh,Yaouank)`qui réussit si *Kozh* est un ancêtre quelconque (parent, grand - parent, arrière grand parent, etc...) de *Yaouank.*

2 — Essayez d'écrire un prédicat d'arité 2 `famille(H1,H2)` qui réussit si *H1* et *H2* font partie d'une même famille (ou lignee), c'est à dire qu'ils ont au moins un ancêtre commun

Programmation logique et récursivité

1 L'algorithme de Syracuse ... sans multiplication ni soustraction.

Quittons maintenant le monde merveilleux des familles royales pour s'intéresser à quelques notions mathématiques non moins merveilleuses. Si le célèbre théorème de Fermat, après avoir donné du fil à retordre aux mathématiciens pendant plusieurs siècles, est maintenant démontré, il reste de nombreuses conjectures dont l'énoncé est d'une simplicité déroutante mais dont la preuve reste encore à trouver. L'algorithme de Syracuse (du nom de l'université américaine où il a été étudié) est de ceux-là. En voici le principe :

(1) on part d'un entier initial N,
(2) si N est pair, on le divise par 2. Sinon, on le multiplie par 3 et on ajoute 1 au résultat. On obtient ainsi un nouvel entier N (nouvel élément de la suite de Syracuse).
(3) Poursuivre avec le nombre N obtenu en reprenant à l'étape (2), et ainsi de suite.

Cet algorithme très simple a une particularité : on retombe toujours au bout d'un moment sur le nombre 1. C'est cette convergence vers 1 qui déroute les mathématiciens. Enoncé en 1950 par le mathématicien allemand Collatz, le problème résiste toujours aux efforts des chercheurs, très nombreux, qui s'y sont attaqués. A tel point qu'on a pu dire, à l'époque de la guerre froide, que c'était le KGB qui avait lancé le problème pour ralentir la recherche mathématique américaine !

Dans cet exercice, nous n'allons bien sûr pas nous attaquer à la démonstration de la conjoncture, mais réaliser un programme Prolog qui vérifiera expérimentale sa validité. Pour cela, on désire réaliser un programme Prolog qui, une fois fournit un entier quelconque de départ, affiche tous les nombres de la suite obtenus par l'algorithme de Syracuse, jusqu'à arriver à un 1 : vous vérifiez ainsi que l'on retombe bien toujours sur ce nombre.

Pour réaliser ce programme, on pourra utiliser les deux prédicats arithmétiques prédéfinis suivants :

> `plus(X,Y,Z)` `est` vrai ssi X, Y, Z **entiers relatifs** vérifient Z = X+Y
> `between(X,Y,Z)` vrai si et seulement si l' **entier relatif** Z est compris (au sens large) entre les **entiers relatifs** X et Y, **Y étant supérieur à X.**

En SWI-Prolog, il n'existe pas de prédicat logique prédéfini réalisant l'opération de multiplication ou de division[1]. Dans le cadre de cet exercice, l'utilisation récursive d'additions et de soustractions devrait nous sortir d'affaire...

1 — Réaliser un prédicat d'arité 1 `impair(N)` qui réussit si et seulement si N est un nombre impair. On cherchera pour cela un algorithme récursif permettant de se ramener, en utilisant uniquement les prédicats arithmétiques prédéfinis, aux cas particuliers N=1 (réponse : vrai) et N=0 (réponse : faux).

2 — Réaliser un prédicat d'arité 2 `div2(NN,N)` qui réussit si et seulement si N est un nombre entier égal au nombre entier NN divisé par 2. On cherchera pour cela un algorithme récursif permettant de réaliser la division par 2 uniquement à l'aide des prédicats arithmétiques prédéfinis. On remarquera au passage que le prédicat réalisé ne réussit que pour des nombres pairs.

3 — Réaliser un prédicat d'arité 2 `suivant(Courant,Suivant)` qui réussit si et seulement si `Suivant` est le nombre obtenu en appliquant une fois l'étape (2) de l'algorithme de Syracuse au nombre `Courant`.

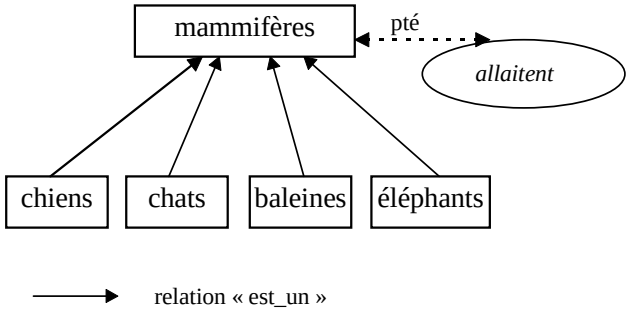
4 — En utilisant les prédicats définis précédemment, réaliser le prédicat d'arité 1 `syracuse(Init)` qui, partant du nombre entier initial `Init`, affiche tous les éléments de la suite obtenue par l'algorithme de Syracuse jusqu'à arrêt sur un " 1 ".

2. Petite histoire de l'art : réseaux sémantiques

Les réseaux sémantiques correspondent à un type de représentation des connaissances inventés au début des années 1970 et qui permet de limiter la déclaration des propriétés associées à un objet représenté en faisant remonter celle-ci à la plus haute catégorie englobante (on parle d'*hyperonyme*) à laquelle ont peut rattacher cet objet. Par exemple, au lieu de préciser que les chats, les chiens, les vaches, les baleines et les éléphants allaitent tous leurs bébés, il est plus simple

[1] Il est en fait possible de surmonter ce problème en Prolog à l'aide de prédicats **extralogiques**, implantés dans un langage de progr ammation impératif (langage C pour SWI-Prolog). Pour le moment, il est clair que la programmation de cet exercice est assez lourde. Cela fait toutefois tout l'intérêt de ce problème du point de vue de la récursivité et vous montre qu'il reste possible de réaliser un tel programme en se limitant à une **programmation purement logique**.

de définir une catégorie hyperonyme " mammifères " et de définir à ce niveau cette propriété. Tous les éléments appartenant à cette catégorie (on parle d'*hyponymes*) posséderons alors cette propriété (on parle d'*héritage de propriété*) :



On crée ainsi une représentation sous forme d'un arbre taxinomique correspondant à un réseau sémantique simplifié[2].

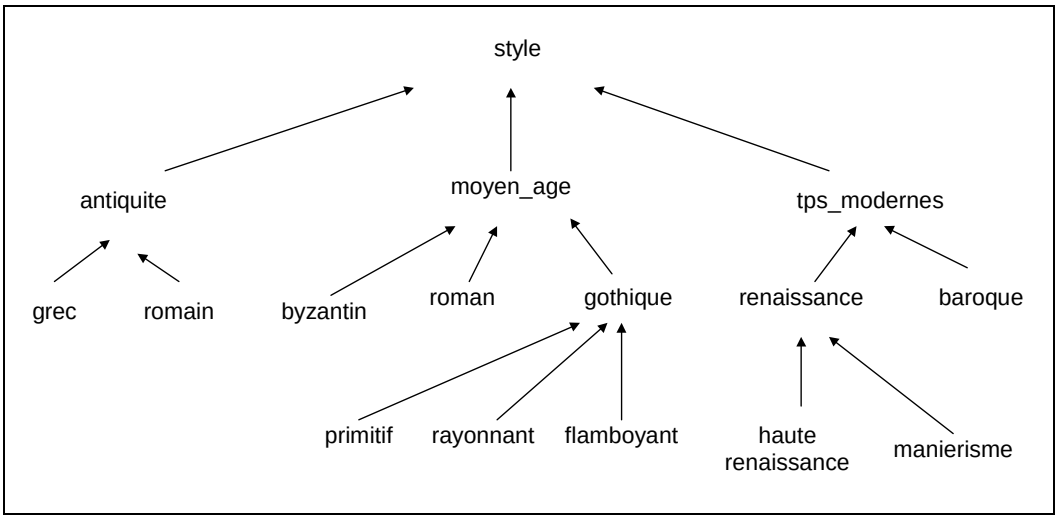
La représentation d'un tel réseau sémantique en Prolog est simple : les " branches " de l'arbre sont décrites une à une par un prédicat `est_un(X,Y)` qui rend compte des relations d'hyponymie / hyperonymie. Par exemple :

```
est_un(chats,mammiferes).
```

On définit alors chaque propriété au niveau de plus élevé possible de l'arbre. Par exemple, on n'a plus à définir la propriété `allaitement` qu'au niveau des mammifères. Cette propriété sera ensuite **héritée** par l'ensemble des sous-espèces de la classe des mammifères, grâce à la relation `est_un`. L'objectif de ce TP sera précisément de réaliser ce mécanisme d'héritage ... hautement récursif ... suivant les relations taxinomiques `est_un`.

2.1. Réseau sémantique architectural

Dans ce TP, nous allons travailler sur un réseau sémantique consacré à l'histoire de l'architecture. Nous étudierons ici une taxinomie des styles architecturaux européens depuis l'antiquité jusqu'aux temps modernes (fin XVIII° siècle) :



Jusqu'ici, nous n'avons fait que décrire la hiérarchie des styles architecturaux. Nous allons maintenant nous intéresser à leurs caractéristiques. Nous allons nous intéresser à deux caractéristiques architectoniques principales : le type d'arc utilisé pour les ouvertures et la présence ou non d'ordres, c'est à dire l'utilisation ou non d'un des types de colonnes (dorique, ionique, corinthien, toscan, composite) inventés au cours de l'antiquité et maintes fois utilisées depuis.

Au regard de ces deux notions, on peut énoncer les propriétés définies dans le tableau suivant :

STYLE	ARCS	ORDRES
Antique	non	oui

[2] On remarque en effet que les relations entre les éléments concernent ici uniquement des relations d'hyponymie - hyperonymie formant un arbre hiérarchisé. Dans un réseau sémantique général (i.e. non simplifié), ces relations peuvent être plus complexes et former un graphe non orienté.

Roman	en plein cintre	non
Byzantin	outrepassé	non
Gothique	ogive	non
Temps modernes	en plein cintre	oui

Cette taxinomie de styles architecturaux est décrite dans le fichier `archi.pl` placé sur le réseau. On utilise pour cela le prédicat d'arité 2 `est_un` qui décrit l'ensemble des relations *directes* entre styles et sous-styles. Par exemple :

```
?- est_un(gothique,moyen_age).
YES

?- est_un(roman,antiquite).
NO

?- est_un(manierisme, tps_modernes).
NO
```

Par ailleurs, les caractéristiques de chaque style sont définies, au niveau de la catégorie taxinomique la plus englobante, à l'aide de deux prédicats `ordre` (présence d'ordre) et `arc` (type d'arc si utilisé) d'arités respectives 1 et 2.

1 — Récupérez le fichier `archi.pl` et copiez-le sur votre disquette. Vous travaillerez à présent avec ce fichier

2 — Les faits écrits ci-dessus ne décrivent que les relations directes entre 2 nœuds de l'arbre. Définir un prédicat récursif `is_a` d'arité 2 qui permette d'étendre la portée du prédicat `est_un` à des relations indirectes. Par exemple :

```
?- is_a(manierisme, tps_modernes).
YES

?- is_a(manierisme, antiquite).
NO
```

Effectuez plusieurs jeux d'essais pour bien vérifier le bon comportement du prédicat `is_a`.

2.2 Héritage de propriétés dans un réseau sémantique

Il nous faut maintenant nous préoccuper de l'héritage des propriétés suivant les relations taxinomiques décrites par le prédicat `is_a`.

Complétez la définition des prédicats `ordre` et `arc` de manière à assurer un héritage de propriété dans toutes les sous-classes d'une classe possédant une certaine propriété... Testez le bon comportement de votre programme. Par exemple, vérifiez que l'on a bien :

```
?- ordre(romain).
YES

?- arc(grec,X).

NO

?- arc(X,ogive).
YES
X = gothique
X = gothique_primitif
X = gothique_rayonnant
X = gothique_flamboyant
```

3. Listes : un problème stroumpfement stroumpf

Remarque — Cet exercice fait appel à l'utilisation d'une structure de données avancée : les *listes Prolog*. Si vous avez le temps d'aller jusqu'ici, demandez une petite explication sur les listes à votre enseignant avant de vous lancer dans la programmation de votre problème…

Craignant de voir sa côte de popularité chuter comme celle de son premier ministre, Jacques Chirac cherche depuis quelques mois à s'impliquer au minimum dans les affaires de politique intérieure afin d'apparaître comme un Président à la stature internationale au-dessus des problèmes franco-français. Ne sachant comment tuer le temps jusqu'à la prochaine élection présidentielle à laquelle pense également son nouvel ami de trente ans, Nicolas Sarkozy, notre cher président accueille chaque voyage officiel comme autant de bouffées d'oxygène. Aussi saute-t-il de joie lorsqu'il apprend qu'il est

invité en visite d'Etat au pays des stroumpfs ! Malheureusement, le chef du protocole est incapable de trouver un traducteur parlant le langage stroumpf. Les services de la présidence font alors appel à vos talents de programmeur pour réaliser un petit traducteur automatique français/stroumpf.

Représentation des connaissances — On cherche à réaliser un programme Prolog qui traduise une phrase écrite en français en une phrase stroumpf. Pour cela, on représentera tout énoncé par une liste de mots. Par exemple :

Vive le peuple stroumpf [vive, le, peuple, stroumpf]

On suppose qu'on dispose par ailleurs des prédicats d'arité 1 `verbe(X,TPS)`, `adverbe(X)` et `autre(X)` qui réussissent si X est un mot correspondant respectivement à un verbe (conjugué ou non), un adverbe, ou à toute autre catégorie grammaticale. Ces prédicats fonctionnent à la fois en mode (+) et en mode (-).

1 — On considère à titre d'exemple le vocabulaire restreint suivant : {je,vous, la, au, de, nom, bienvenue, souhaite, France}. Représenter chacun de ces mots à l'aide des prédicats définis précédemment.

En première approximation, le langage stroumpf est très proche du français. Pour effectuer la traduction, il suffit en effet de remplacer tous les verbes français par le mot *stroumpfe* et tous les adverbes par *stroumpfement*. Par exemple :

Reste ici, je reviens tout de suite est traduit par *Stroumpfe ici, je stroumfe tout de suite*

2 — Ecrire un prédicat d'arité 2 `trad_mot(FR,S)` qui réussit si `S` correspond à la traduction en stroumpf du mot français `FR`.

à la traduction en stroumpf d'un énoncé français représenté par la liste `L_FR`.