Séminaires Technologiques
Prolog

Françoise Dubisy

Informatique de gestion

UE 225 : Séminaires technologiques

Hénallux – Catégorie économique – 2022-2023

Table des matières

1 Qu’est-ce que l'intelligence artificielle ? 5

1.1 L'origine de l'intelligence artificielle 5

1.2 Principes, méthodes et caractéristiques de l'intelligence artificielle 7

1.2.1 L'intelligence requiert de la connaissance 7

1.2.2 Manipulation de symboles 7

1.2.3 Usage intensif d'inférences 8

1.2.4 Notion d'apprentissage 9

1.3 Comparaison des méthodes d'intelligence artificielle par rapport aux méthodes classiques 10

2 Domaines d'application 10

3 Les systèmes experts 13

4 Connaissance procédurale >< connaissance déclarative 15

5 Introduction au langage Prolog 18

6 Base de connaissances 20

6.1 Faits 20

6.1.1 Terme 20

6.1.2 Symbole de prédicat 21

6.1.3 Fait 21

6.2 Règles 22

6.3 Procédures Prolog 25

7 Moteur d'inférence de Prolog 26

7.1 Question 26

7.2 Backtracking (Rétro-parcours) 29

7.2.1 Backtracking dans un arbre de recherche OU 29

7.2.2 Backtracking dans un arbre de recherche ET 34

8 Récursivité 40

9 Tuyaux et recommandations 46

Première partie :

Qu’est-ce que l’intelligence artificielle ?

1 Qu’est-ce que l'intelligence artificielle ?

L'intelligence artificielle est l'étude du **comportement intelligent**.

Son but :

- Créer une théorie de l'intelligence qui tient compte du comportement des êtres naturels intelligents et qui guide la création d'entités artificielles, c'est-à-dire de machines capables de comportement intelligent.
- Rendre les **ordinateurs plus intelligents**, plus utiles.

Faire faire à la machine ce que l'homme est capable de faire.

1.1 L'origine de l'intelligence artificielle

L'intelligence artificielle est une science carrefour

Elle résulte de la convergence de plusieurs disciplines ayant un intérêt pour la pensée et le fonctionnement du cerveau :

- **Neurologie**

L'étude du système nerveux chez les êtres vivants a permis de développer des modèles du cerveau en tant qu'organisation de neurones. Ces derniers modèles ont inspiré les travaux en intelligence artificielle sur les **réseaux neuronaux**.

- **Psychologie**

La psychologie dégage les mécanismes qui régissent notre comportement. La psychologie **cognitive**, basée sur la description des facultés intellectuelles en tant que **systèmes de traitement de l'information**, fait un large usage des programmes d'intelligence artificielle comme outils de modélisation des processus cognitifs. On essaie donc de prolonger l'intelligence humaine dans les ordinateurs. Mais un autre objectif est indissociable : **mieux comprendre l'intelligence humaine.**

Exemples :

- *Pour construire une machine capable de jouer aux échecs, il faut s'interroger sur la façon dont l'humain raisonne en jouant.*

- *Les progrès de l'aéronautique n'ont été possibles que grâce à l'étude du vol des oiseaux. Ceci a eu pour conséquence une meilleure compréhension des mécanismes de vol chez l'oiseau.*

• Logique

Herbrand (1930's) et Turing (1950's) ont mis au point une méthode de déduction automatique ainsi que des modèles théoriques de la machine.

Ceci a donné naissance aux deux principaux langages utilisés en intelligence artificielle : **LISP** et **PROLOG**.

Les représentations logiques inspirent la création du premier programme d'intelligence : *le Logic Theorist* (1956).

• Linguistique

Les résultats des recherches en linguistique ont été utilisés pour alimenter les travaux de **traduction automatique** et de **compréhension du langage par la machine**.

• Informatique

L'informatique fournit à l'intelligence artificielle l'**outil** indispensable à la simulation et l'implantation.

Pionniers de l'intelligence artificielle

Le **Logic Theorist** de Newell, Shaw et Simon marque la naissance de l'intelligence artificielle en **1956**. Le programme est capable de démontrer des théorèmes de la logique des propositions

1.2 Principes, méthodes et caractéristiques de l'intelligence artificielle

1.2.1 L'intelligence requiert de la connaissance

Un important concept en intelligence artificielle est le concept de **connaissance**.

Les entités intelligentes semblent **anticiper** les conséquences de leurs actions et les réactions de l'environnement. Elles agissent donc comme si elles "savaient", "connaissaient" le résultat de leurs actions.

Ceci n'est possible que si on suppose que les entités elles-mêmes possèdent de la connaissance sur leur environnement.

☺ → **Importance de la connaissance !**

N.B. Problème de la **représentation des connaissances**.

Il ne faut pas de longues tables de valeurs numériques mais des structures riches et souples avec la possibilité d'exprimer des liens variés entre les éléments.

1.2.2 Manipulation de symboles

On peut distinguer trois phases de développement en informatique :

• A l'origine, l'ordinateur traitait presque exclusivement de l'information **numérique**.

• Par la suite, les informations étaient de type **alphanumérique** c'est-à-dire des nombres et textes.

Exemple : bases de données

• Enfin, en intelligence artificielle, les connaissances (faits, énoncés, règles, méthodes...) sont représentées par des systèmes de **symboles** à manipuler.

⇨ Traitement de symboles

1.2.3 Usage intensif d'inférences

Un programme en intelligence artificielle calcule peu, mais il déduit beaucoup en tirant des conséquences (conclusions), d'abord à partir des données de départ, et ensuite à partir des résultats intermédiaires de plus en plus nombreux qu'il découvre.

Il existe différents types d'inférences. En voici deux exemples :

Déduction

si on a **P ⇒ Q**

et **P**

on peut en déduire **Q**

Exemple : S'il neige, alors il fait froid.

or, il neige.

Déduction : Il fait froid.

Induction

L'induction se base sur le principe de la généralisation à partir d'instances.

Si on a : *mouette vole*

canari vole

cormoran vole

pigeon vole

et si on sait que ***Tous sont des oiseaux*** 

↳ Conclusion par induction : **Tous les oiseaux volent**.

💣 **Induction** >< **Déduction**

↓ ↓

La conclusion n'est **pas garantie**

⇩

Peut ne pas être toujours applicable

⇩

Truth Preserving : si les prémisses sont vraies, les conclusions sont vraies

Exemples : L'autruche ne vole pas, or l'autruche est un oiseau.

1.2.4 Notion d'apprentissage

Cette notion est nouvelle par rapport à l'informatique classique.

Les systèmes intelligents pourraient être capables d'apprentissage au fur et à mesure de leur **expérience**.

Il ne s'agit pas seulement d'ajouter de nouvelles informations dans la base de connaissances. L'apprentissage demande éventuellement un remaniement, une **intégration** de ces nouvelles informations au reste de la base de connaissances.

1.3 Comparaison des méthodes d'intelligence artificielle par rapport aux méthodes classiques

Méthodes d'intelligence artificielle	Méthodes classiques
Près du fonctionnement de l'être humain	Près du fonctionnement de la machine
Traitement des symboles	Traitement de nombres ou textes
Utilisent beaucoup d'inférences	Utilisent beaucoup de calculs
Font appel à des heuristiques et raisonnements incertains	Suivent des algorithmes rigides et exhaustifs
Sont généralisables à des domaines complètement différents	Ne sont généralisables qu'à une classe de problèmes semblables

- **Perception**
    - Vision : interprétation d'images
    - Parole :
        - Reconnaissance de la parole/écriture
        - Compréhension/traduction de langues naturelles

- **Résolution de problèmes** requérant de l'expertise

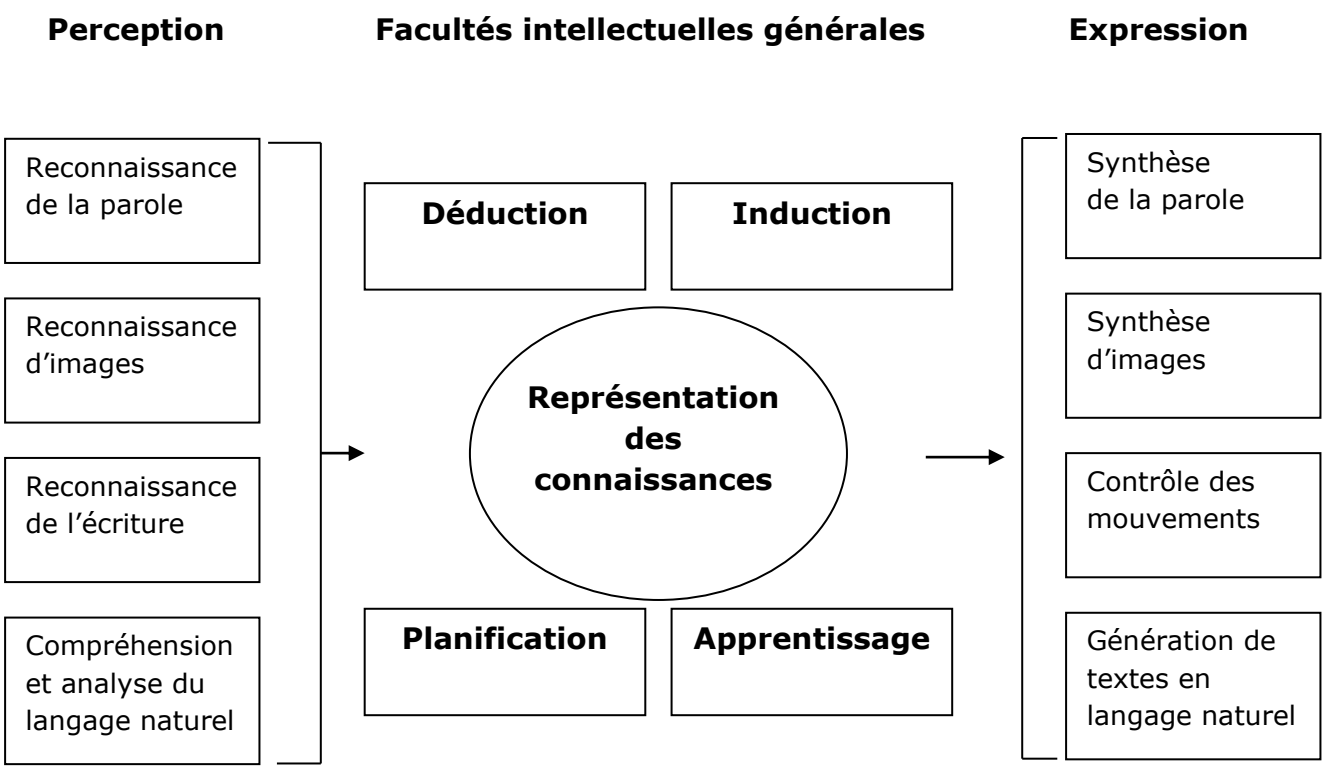
  → **Systèmes experts**

  Exemples : diagnostic médical, analyse chimique...

- **Génération de plans**, projets (planning)

  Exemple : robotique

Différents champs d'étude en intelligence artificielle



3 Les systèmes experts

Un **expert** est une personne qui possède une **connaissance** et une **expérience pointue** dans un domaine étroit. Cette expérience est **difficile à transmettre** à un novice. Ce transfert demande souvent de longues années d'efforts et d'apprentissage.

D'où l'idée des informaticiens travaillant en intelligence artificielle d'utiliser l'ordinateur pour acquérir (une fois pour toutes) et stocker les connaissances des experts. En effet, en informatique, le **stockage** des informations est peu coûteux et le transfert des connaissances est instantané par une simple **copie** des données.

Notons qu'accumuler des connaissances, c'est bien (stockage peu coûteux, facilité de copie) ☺

Mais savoir s'en servir, c'est mieux. ☺ ☺

Il faut donc pouvoir **automatiser un raisonnement** sur base des connaissances ainsi accumulées et stockées. Il faut donc trouver un formalisme adéquat à la fois pour représenter les connaissances et les utiliser, c'est-à-dire raisonner.

Les connaissances des experts semblent être souvent représentées sous forme de **règles**. En effet, les experts semblent raisonner, en partie tout au moins, sur base de règles.

Exemples de raisonnement chez l'être humain basé sur des règles :

① Application d'une règle générale à un cas particulier

C'est par exemple le cas de l'étudiant à l'examen. Il doit résoudre un exercice en appliquant les principes, théorèmes et autres techniques apprises au cours. L'étudiant est capable (ce qui est souhaitable pour lui !) de résoudre ainsi des exercices jamais encore rencontrés.

② Déduction d'une conclusion si les conditions sont satisfaites

Si la règle *"Si A et B sont vrais, alors C est vrai"* fait partie des connaissances d'un individu, celui-ci déduira automatiquement que *C est vrai* s'il se rend compte que *A et B sont vrais.*

③ Enchaînement de règles

Si les règles *"Si A est vrai, alors B est vrai"*

et *"Si B est vrai, alors C est vrai"*

font partie des connaissances d'un individu, celui-ci déduira automatiquement que *C est vrai* s'il se rend compte que *A est vrai*.

Les langages de programmation procéduraux classiques (boucles, affectation...) sont inappropriés pour ce type de tâches.

↳ Un autre type de langage est indispensable pour stocker les connaissances et les exploiter de manière efficace. Ce nouveau type de langage de programmation doit permettre le raisonnement par **déduction** à partir des connaissances emmagasinées, et ce **en réponse à une question**.

Ces nouveaux langages sont basés sur la **logique** mathématique.

4 Connaissance procédurale >< connaissance déclarative

Un comportement intelligent dépend des connaissances que l'entité "intelligente" possède sur son environnement.

Dans les machines, on peut considérer qu'il y a **2 façons de posséder de la connaissance** : de façon **implicite** ou **explicite**.

Connaissance implicite

La connaissance est stockée dans le programme ; elle est codée dans la séquence d'opérations.

Exemple : programme d'inversion d'une matrice (en mathématiques).

⇨ Difficile d'extraire cette connaissance du programme.

= Connaissance procédurale

= Le comment faire

Connaissance explicite

Les connaissances sont représentées par des phrases **déclaratives** : ce sont des déclarations à propos du monde.

Exemples : Le carburateur est un élément du moteur

Un moteur en marche fait du bruit et de la fumée

Un carnivore mange de la viande

= Connaissances déclaratives

= Le quoi

N.B. Les informations stockées dans les tables des **bases de données** sont également de type **explicite**. Elles sont indépendantes des programmes d'accès aux données.

Exemple : tables reprenant le signalétique des employés.

Illustrons ces deux types de connaissances à partir d'exemples puisés chez l'homme.

- *Tennis par un champion*

  → Connaissance procédurale.

- *Construction d'un pont*

  → Connaissance déclarative sur les matériaux, les structures.

Avantages des connaissances déclaratives

- Leur **modification** est **aisée**.

  Il n'est pas nécessaire de toucher au code du programme.

- Ces connaissances peuvent être **réutilisées**.

  Dans un but différent du but original.

- Elles sont **indépendantes de l'application**.

  ↳ Elles peuvent être réutilisées dans d'autres applications sans qu'il soit nécessaire de les dupliquer dans chaque application.

- Elles peuvent être étendues par des raisonnements qui induisent de **nouvelles connaissances** à partir de connaissances données.

Seconde partie :

Le langage Prolog

5 Introduction au langage Prolog

Le langage **Prolog** est un langage destiné à la programmation logique, d'où son nom (**PRO**gramming in **LOG**ic).

Le but de toute programmation logique est d'arriver à une solution en **déduisant logiquement une conclusion à partir de connaissances de départ** (connues).

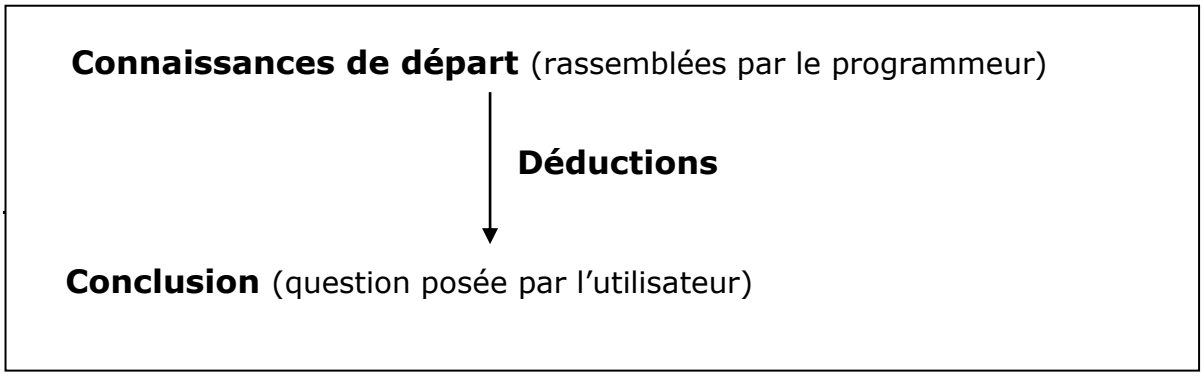
L'objectif d'un programme Prolog est de démontrer qu'une hypothèse de départ (la **question** posée au programme) est **vraie**, et ce, par **déductions successives** à partir des connaissances de départ que possède le programme.

Un programme prolog n'est pas un ensemble d'instructions/actions (affectations, boucles, alternatives…) comme dans un programme procédural classique.

Un programme Prolog est constitué d'un ensemble de **faits** et de **règles**.

Par conséquent, écrire un programme Prolog c'est déterminer l'ensemble des connaissances à partir desquelles le programme pourra démarrer sa recherche par déductions successives pour répondre à une question que l'utilisateur lui posera.

Ecrire un programme Prolog c'est donc déterminer **sa base de connaissances.**



Supposons que nos connaissances de départ contiennent :

Le fait : *Socrate est un humain*

La règle : *Tous les humains sont mortels*

Peut-on déduire de cet ensemble de connaissances, le fait que *Socrate est mortel ?*

6 Base de connaissances

Une base de connaissances est constituée d'un ensemble de clauses. Une clause est soit un fait, soit une règle.

💣 **Tout fait et toute règle Prolog se terminent par un point**. 💣

6.1 Faits

$$p(a_1,\ldots, a_n)\ .$$

Symbole de **prédicat** ⇦ ⇩

Arguments = **termes**

6.1.1 Terme

- **Constante**

  Domaines standard :

  → integer, real

  → char *:* valeur entre **' '**

  → symbol **: première lettre minuscule !** (sinon entre **' '**)

  *Exemples : **p**ierre, **r**ome, **'J**ules **C**ésar'…*

- **Variable** : la **première lettre doit être en majuscule**, ce qui permet à Prolog de distinguer une variable des autres symboles (constantes, prédicats…).

  *Exemples : **T**iti, **X**, **X**YZ*

💣 Attention en Prolog, il n'y a **pas de fonction** dans le sens mathématique d'une fonction qui renvoie un résultat 💣

6.1.2 Symbole de prédicat

Tout symbole de prédicat commence de préférence par une **minuscule**.

6.1.3 Fait

Un fait permet de représenter différents types de connaissances :

① Un fait peut exprimer une **propriété d'un objet.**

Exemple : blonde(anne).

② Un fait peut exprimer qu'une **relation existe entre des objets**. Un fait est alors une relation Prolog entre les termes.

Exemple : pere(jules,pierre).

③ Un fait peut exprimer qu'un **objet appartient à une classe**.

Exemples : etudiant(pierre).
femme(anne).

④ Un fait peut représenter **une ligne d'une table en relationnel.**

Exemple : emprunt(pierre,bible,'3 septembre 2019 ').

⑤ Un fait peut être associé à **une probabilité.**

Exemple : maladeDe(pierre,grippe,0.8).

Ce fait exprime qu'il y a 80% de chances que Pierre ait la grippe.

⑥ Un fait peut représenter **une fonction classique.**

Exemple : somme(3,4,7).

Ce fait exprime que la somme de 3 et 4 vaut 7.

Une **fonction** à **N arguments** se traduit en Prolog par un **prédicat à N+1** arguments.

L'argument supplémentaire est le résultat.

L'utilisation de variables dans les faits permet de résumer plusieurs faits. On parlera de **fait universel**.

Exemple : julie aime tout le monde

↳ *aime(julie,**X**).*

6.2 Règles

Forme générale : conséquent :- antécédent.
↳but à établir

ou encore **conclusion** **:-** **conditions.**

foncteur unique :- (signifie ⇐) conjonction de foncteurs . (∧)

c :- a_1, a_2, … , a_m .

Exemple :

espece(X,tigre) :- mammifere(X), carnivore(X), couleurFauve(X),
rayuresNoires(X).

Dans cet exemple il est spécifié que pour être un tigre, **il suffit** d'être un mammifère carnivore de couleur fauve avec des rayures noires. Quatre conditions suffisent donc pour être un tigre.

Les règles du type $\mathbf{A} \Leftarrow \mathbf{B} \vee \mathbf{C}$ ne sont représentables en Prolog que si elles sont scindées en : $\mathbf{A} \Leftarrow \mathbf{B}$
$\mathbf{A} \Leftarrow \mathbf{C}$

Exemple : $parent(X,Y) \Leftarrow pere(X,Y) \vee mere(X,Y)$

$\Rightarrow$ $parent(X,Y) \Leftarrow pere(X,Y)$

$parent(X,Y) \Leftarrow mere(X,Y)$

Ou encore, en Prolog : *parent(X,Y) :- pere(X,Y) .*
parent(X,Y) :- mere(X,Y).

Deux règles de même conséquent représentent donc en Prolog une alternative (**OU**).

Il en va de même pour les règles qui contiennent plusieurs conséquents reliés par $\wedge$ (**ET**). Il faut les scinder en plusieurs règles (cf. résumé ci-dessous).

En résumé

Logique des prédicats	Prolog
si (a(x) **et** b(x)) alors c(x)	c(**X**) **:-** a(**X**)**,** b(**X**)**.**
si (a(x) **ou** b(x)) alors c(x)	c(**X**) **:-** a(**X**)**.** c(**X**) **:-** b(**X**)**.**
si a(x) alors (c(x) **et** d(x))	c(**X**) **:-** a(**X**)**.** d(**X**) **:-** a(**X**)**.**
si (a(x) **ou** b(x)) alors (c(x) **et** d(x))	c(**X**) **:-** a(**X**)**.** c(**X**) **:-** b(**X**)**.** d(**X**) **:-** a(**X**)**.** d(**X**) **:-** b(**X**)**.**
si a(x) alors (c(x) **ou** d(x))	*impossible !*

N.B. Si le dernier cas se présente, le programmeur doit "affiner" sa règle. Il doit "puiser" dans le domaine d'application et essayer de trouver des conditions supplémentaires lui permettant d'identifier dans quel cas on peut conclure **c**(x) et dans quel autre cas on peut conclure **d**(x).

Exemple : si (a(x) ***et condition1(x)*** *) alors* ***c****(x)*

si (a(x) ***et condition2(x)*** *) alors* ***d****(x)*

Ce qui se traduit en Prolog par : *c(**X**)* ***:-*** *a(**X**),* ***condition1(x).***

*d(**X**)* ***:-*** *a(**X**),* ***condition2(x).***

6.3 Procédures Prolog

> Une procédure Prolog est un ensemble de clauses (faits ou règles) qui définissent **la même relation**.

Exemples : ***parent****(jules, pierre).*

parent*(pierre,anne).*

parent*(X, Y) :- pere(X, Y).*

parent*(X, Y) :- mere(X, Y).*

Toutes les règles de la procédure ont donc **le même prédicat dans leur conséquent** !

Prolog est (relativement) non procédural. L'ordre **entre les procédures** n'a **pas d'importance.**

Exemples :

mere( )		ancetre( )
...		...
parent( )	ou	parent( )
...		...
ancetre( )		mere( )

Mais l'ordre **à l'intérieur d'une même procédure** est **important**. Pour des raisons de performance (et éviter le bouclage),

↳ Les faits doivent apparaître en tête de procédure

7 Moteur d'inférence de Prolog

On exploite la base de connaissances en la questionnant.

7.1 Question

Toute question est une clause particulière de la forme :

$a_1, a_2, \ldots a_m.$

Ce qui signifie : Est-ce que a_1 ET a_2 ET … a_m peuvent être déduites de la base de connaissances ?

Soit une base de connaissances qui comprend les faits suivants :

mere(julie,anne).

mere(julie,laure).

mere(anne,paul).

Exemples de questions posées à Prolog et de réponses fournies par Prolog :

mere(julie,anne). *Est-ce que Julie est la mère de Anne ?*

Réponse de Prolog : **yes**

mere(anne,julie). *Est-ce que Anne est la mère de Julie ?*

Réponse de Prolog : **no**

mere(X,anne). *Qui est la mère de Anne ?*

Réponse de Prolog : **X = julie**

mere(julie,X). *De qui Julie est la mère ?*

*(on demande **toutes les possibilités**)*

Réponse de Prolog : **X = anne**

X = laure

mere(julie,X),mere(X,paul). *Est-ce que Julie est la grand-mère maternelle de Paul ? Si oui, qui est la maman de Paul (et donc la fille de Julie) ?*

Réponse de Prolog : **X = anne**

mere(julie,_). *Est-ce que Julie est mère (on ne veut pas savoir de qui)?*

Réponse de Prolog : **yes**

N.B. Le caractère **_** (underscore) peut être utilisé à la place d'une variable. Il s'agit de la **variable** dite **anonyme**. Aucune valeur ne sera proposée en retour par Prolog pour une telle variable si elle apparaît dans une question.

En conclusion, une procédure Prolog peut être "appelée" de différentes manières.

① Avec **tous les paramètres instanciés**.

Exemple : mere(julie, anne).

Ceci a pour effet de vérifier *si Julie est la mère de Anne*

② Avec **variables**

Exemple : mere(X,anne).

Ceci a pour effet de *trouver la mère de Anne.*

mere(julie, X).

Ceci a pour effet de rechercher *de qui Julie est la mère,* autrement dit, *quels sont les enfants de Julie ?*

⇨ ☺ Il n'est donc **pas nécessaire de créer une procédure enfant(X, Y) !**

Ceci illustre ce qu'on appelle la **réversibilité** des procédures Prolog.

Une question est un **déclencheur d'inférence ou de déduction**. Suite à une question, le système essaye d'obtenir des résultats "satisfaisants" à partir du contenu de sa base de connaissances et des données particulières soumises dans la question.

Pourquoi "satisfaisants" ? Parce que les données dont dispose le système peuvent être **incomplètes**.

N.B. Notons que parfois les connaissances sont incertaines. Typiquement, dans un système expert de diagnostic médical, un coefficient de certitude devrait être associé aux règles.

Pour rappel, poser la question $a_1,a_2,.....,a_n$. équivaut à demander si $a_1$et a_2 et.....et a_n peuvent être déduits de la base de connaissances.

Deux types de réponse sont possibles :

- **YES** si $a_1$et a_2 et.....et a_n peuvent être déduits de la base de connaissances

- **NO** Une réponse négative **ne signifie pas que** a_1 et a_2 et.....et a_n est **faux dans l'absolu**, mais qu'on ne dispose pas dans la base de connaissances des données permettant de conclure que c'est vrai .

Si des **variables** apparaissent dans la question, on veut en plus qu'elles **soient instanciées**.

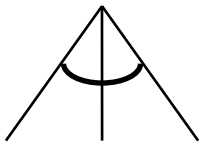
Exemple : mere (X,anne). **X= julie**

7.2 Backtracking (Rétro-parcours)

Le raisonnement effectué par le moteur d'inférence de Prolog peut être représenté sous forme d'un **arbre de résolution ET/OU.**

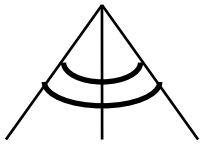
Dans ce type d'arbre, **une règle** est un nœud de type **ET** : chaque nœud fils est une formule atomique (un prédicat) apparaissant dans l'antécédent de la règle. Pour que la règle soit applicable, il faut que chacun des fils (conditions) soit applicable.

Représentation graphique d'un nœud ET :



Le choix entre plusieurs règles ayant un même conséquent est représenté par un nœud de type **OU**. En effet, il suffit qu'une seule des règles ayant le même conséquent soit applicable.

Représentation graphique d'un nœud OU :



7.2.1 Backtracking dans un arbre de recherche OU

Le principe du backtracking dans un arbre de recherche OU sera illustré à l'aide de la base de connaissances suivante :

Base de faits :

poils(jim).

mangeViande(jim).

rayuresNoires(jim).

couleurFauve(jim).

allaite(titi).

longCou(bonzo).

...

Base de règles :

espèce(X,guépard) :- mammifère(X), carnivore(X), couleurFauve(X), tachesSombres(X).

espèce(X,girafe) :- longesPattes(X), longCou(X), tachesSombres(X), rayuresNoires(X).

espèce(X,tigre) :- mammifère(X), carnivore(X), couleurFauve(X), rayuresNoires(X).

espèce(X,zèbre) :- ongulé(X), noirEtBlanc(X), rayuresNoires(X).

espèce(X,autruche) :- oiseau(X), saitPasVoler(X), longCou(X).

espèce(X,puingouin) :- oiseau(X), saitPasVoler(X), nage(X), noirEtBlanc(X).

espèce(X,mouette) :- oiseau(X), saitVoler(X), rie(X).

mammifère(X) :- poils(X).

mammifère(X) :- allaite(X).

carnivore(X) :- dentsPointues(X), griffes(X), yeuxEnAvant(X).

carnivore(X) :- mangeViande(X).

ongulé(X) :- mammifère(X), sabots(X).

ongulé(X) :- mammifère(X), rumine(X).

oiseau(X) :- plumes(X).

oiseau(X) :- vole(X), oeufs(X).

...

- Dans une base de connaissances Prolog, les **faits sont placés en tête** du programme.

  Les règles viennent ensuite.

Question : ***espèce (jim, Y).***

→ Cette question déclenche le moteur d'inférence.

Prolog essaye de trouver un **fait** ou une **règle dont le conséquent s'unifie avec la question**. La recherche est exhaustive : Prolog parcourt toute la liste des faits et des règles **de haut en bas**.

S'il est impossible de trouver un fait qui s'unifie à la question, alors la première règle dont le conséquent s'unifie avec la question est choisie.

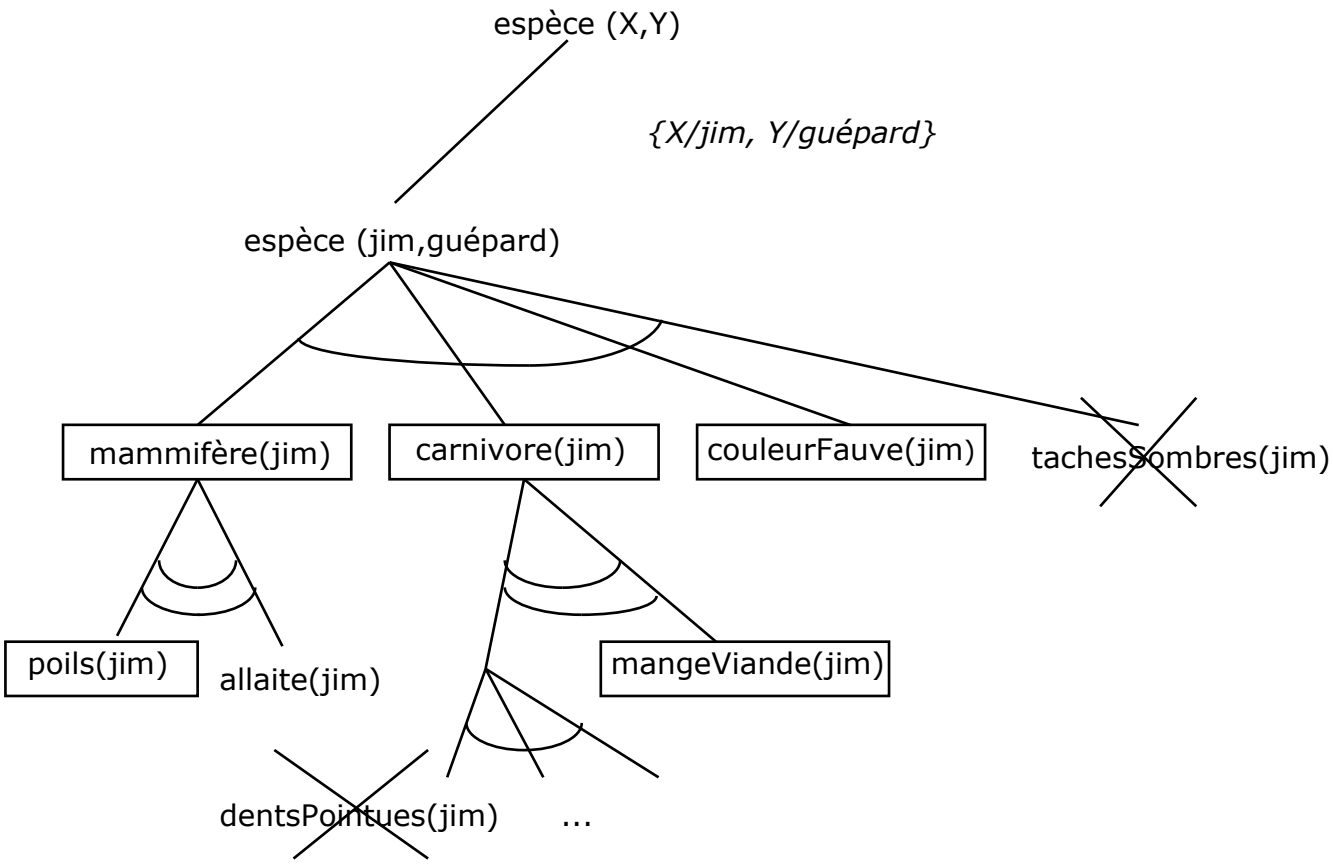
Dans notre base de connaissances, le choix se porte sur ***espèce (X, guépard)***.

Les formules atomiques (prédicats) qui constituent l'antécédent de la règle constituent la liste des objectifs à réaliser. L'ordre d'apparition des formules atomiques (prédicats) dans les conditions de la règle est respecté dans la liste des objectifs à réaliser.

Notons que les substitutions éventuelles sont mémorisées et appliquées à la liste des objectifs, soit ***{X/jim, Y/guépard}.***

La liste des objectifs à réaliser est **donc *{mammifère(jim), carnivore(jim), couleurFauve(jim), tachesSombres(jim)}.***

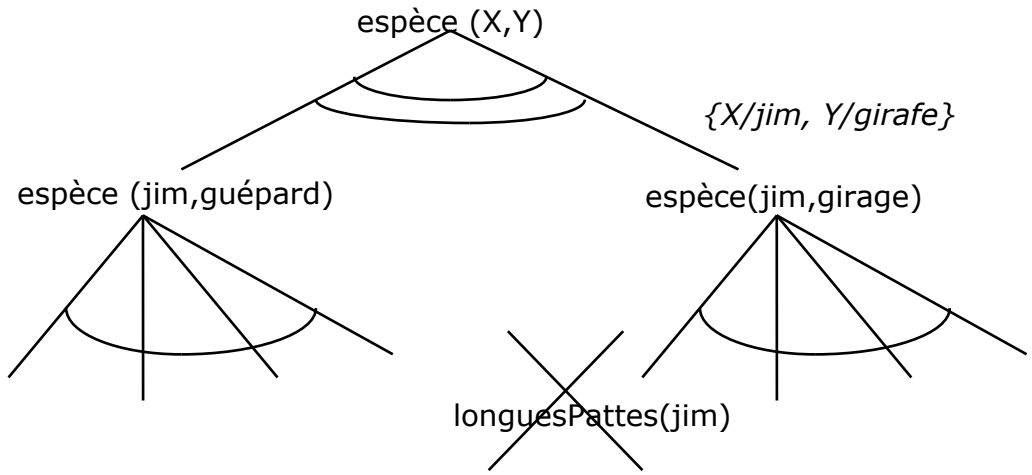
L'application de la règle *espèce(jim, guépard)* échoue car un des quatre sous-objectifs (ET) échoue*: tachesSombres (jim)* n'est pas un fait et aucune règle ayant ce conséquent n'existe.



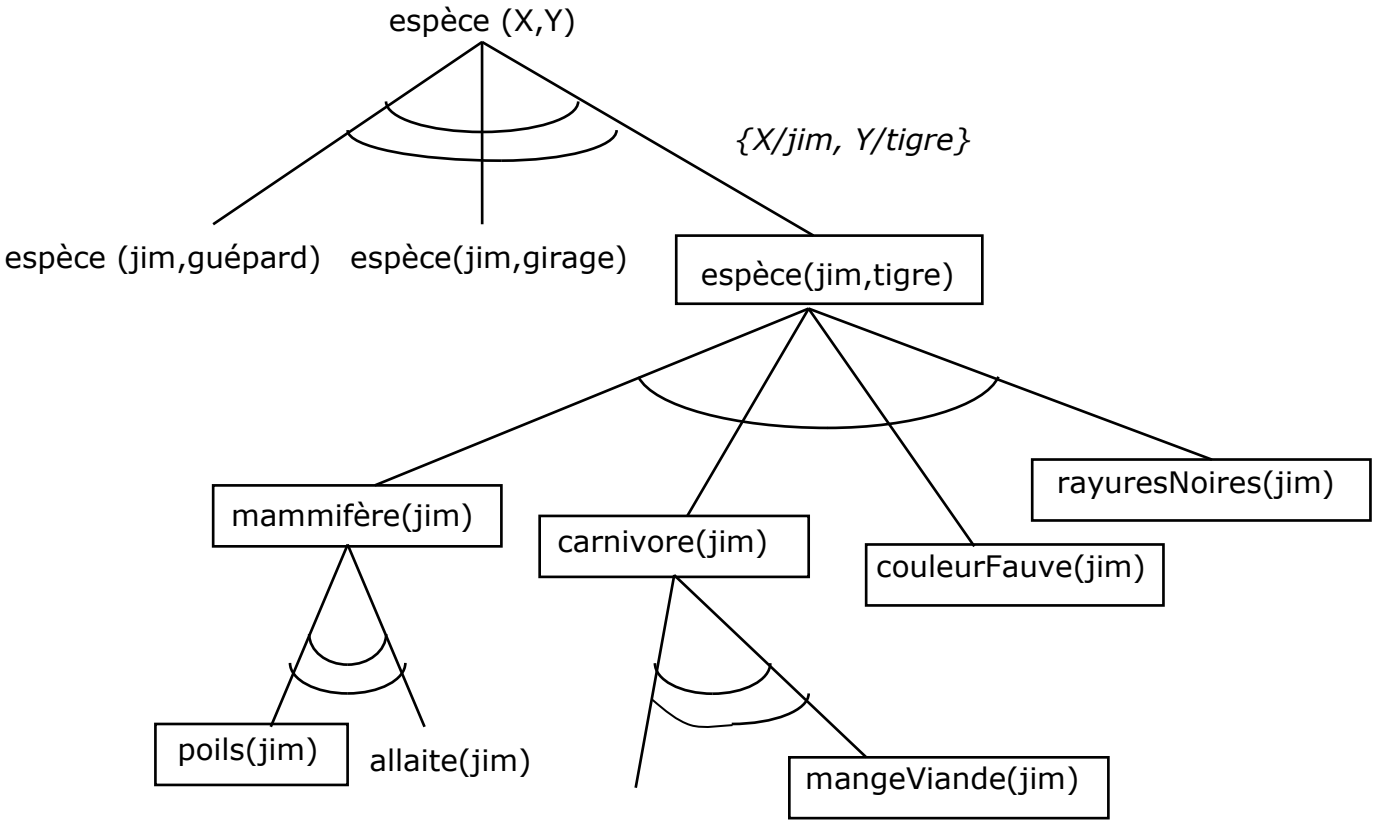
Le backtracking dans un arbre OU consiste à passer à la règle suivante dont le conséquent est espèce(X, Y), c'est-à-dire qu'il y a rétro-parcours dans le sous-arbre OU avec modification / **mise à jour du jeu de substitutions**.

Les liaisons établies pour tenter de prouver que *jim est un guépard* sont défaites et **uniquement ces liaisons-là**. La liaison *{Y/guépard}* est donc défaite, mais pas la liaison *{X/jim}.*

La deuxième règle essayée est la règle ***espèce(jim,girafe).*** Celle-ci échoue également car la première condition de la règle échoue : *longuesPattes(jim).* Il y a de nouveau backtracking. L'instanciation *{Y/girafe}* est défaite et uniquement celle-la.



La troisième règle essayée est la bonne, à savoir ***espèce(jim,tigre).*** En effet, chacune des conditions de cette règle réussit.



La réponse à la question *espèce(jim, X)* est donc **affirmative**, et l'instanciation **Y = tigre** est proposée.

☺ Comme toutes les solutions sont demandées à Prolog, un backtracking artificiel est provoqué. Le moteur d'inférence de Prolog tentera alors de satisfaire l'objectif *espèce(jim,Y)* avec un autre jeu de substitution pour Y (il n'y a pas d'autre solution).

7.2.2 Backtracking dans un arbre de recherche ET

Le principe du backtracking dans un arbre de recherche ET sera illustré à l'aide de la base de connaissances suivante :

Base de faits :

grandMere(anne,marie).

grandMere(claire,marie).

grandMere(anne,jean).

grandMere(jeanne,jean).

grandMere(france,anne).

grandMere(charlotte,pauline).

grandPere(pierre,marie).

grandPere(jules,marie).

grandPere(gerard,jean)

grandPere(jules,jean).

grandPere(pierre,christophe).

...

Base de règles :

cousin(X,Y):-grandsParents(M,P,X),grandsParents(M,P,Y).

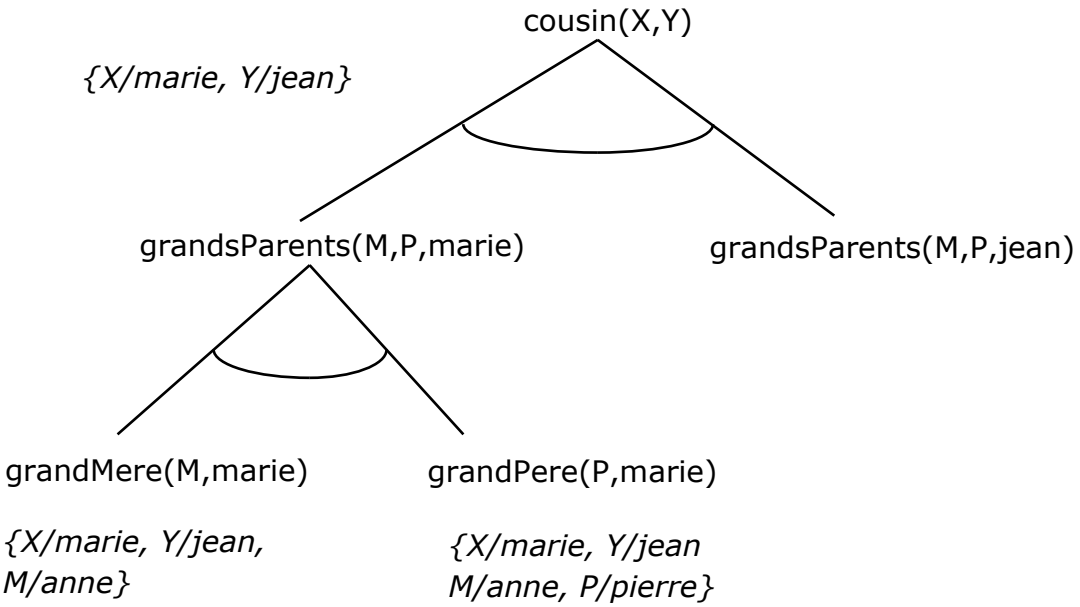
grandsParents(M,P,X):-grandMere(M,X),grandPere(P,X).

Question : ?- ***cousin(marie, jean).***

Quand un objectif échoue dans un sous-arbre de résolution ET, il y a **rétro-parcours jusqu'à l'objectif (Obj¡) ayant nécessité l'instanciation de la variable la plus récente**. **Ce lien-là est défait et uniquement celui-là**. Le moteur d'inférence de Prolog réessaye l'objectif Obj¡ avec un nouveau jeu de substitution concernant la variable défaite.

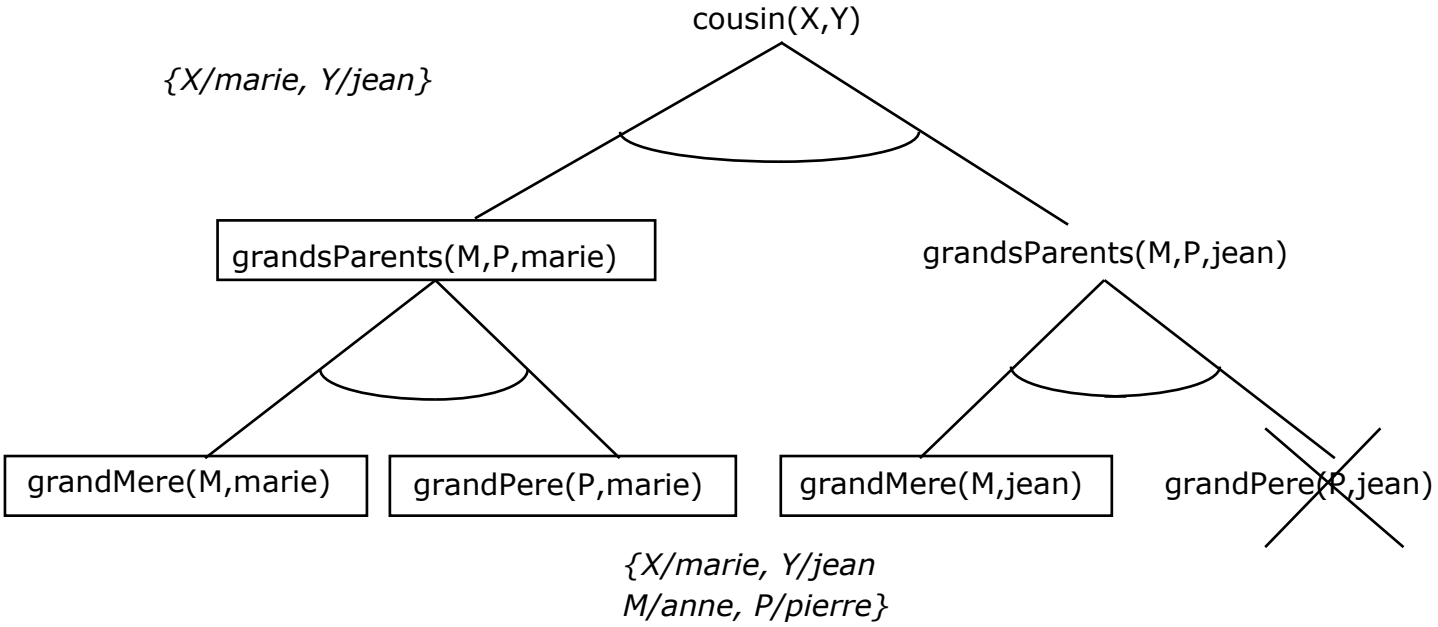
Dans notre base de connaissances, deux objectifs doivent être réalisés en vue de répondre à la question *cousin(marie, jean),* à savoir ***grandsParents(M,P,marie)* et *grandsParents(M,P,jean).*** Le jeu de substitutions est jusqu'à présent constitué de *{X/marie, Y/jean}.* Pour réaliser *grandsParents(M,P,marie),* deux sous-objectifs doivent à leur tour être réalisés: *grandMere(M,marie)* et *grandPere(P,marie).* La liste des objectifs est donc *{grandMere(M,marie), grandPere(P,marie), grandsParents(M,P,jean)}*

Le premier des faits de la base des faits satisfaisants *grandMere(M,marie)* est ***grandMere(anne, marie)***. Le jeu de substitutions devient *{X/marie, Y/jean, M/anne}.* Le premier des faits satisfaisant *grandPere(P,marie)* est ***grandPere(pierre, marie).*** Le jeu de substitutions devient *{X/marie, Y/jean, M/anne, P/pierre}.*



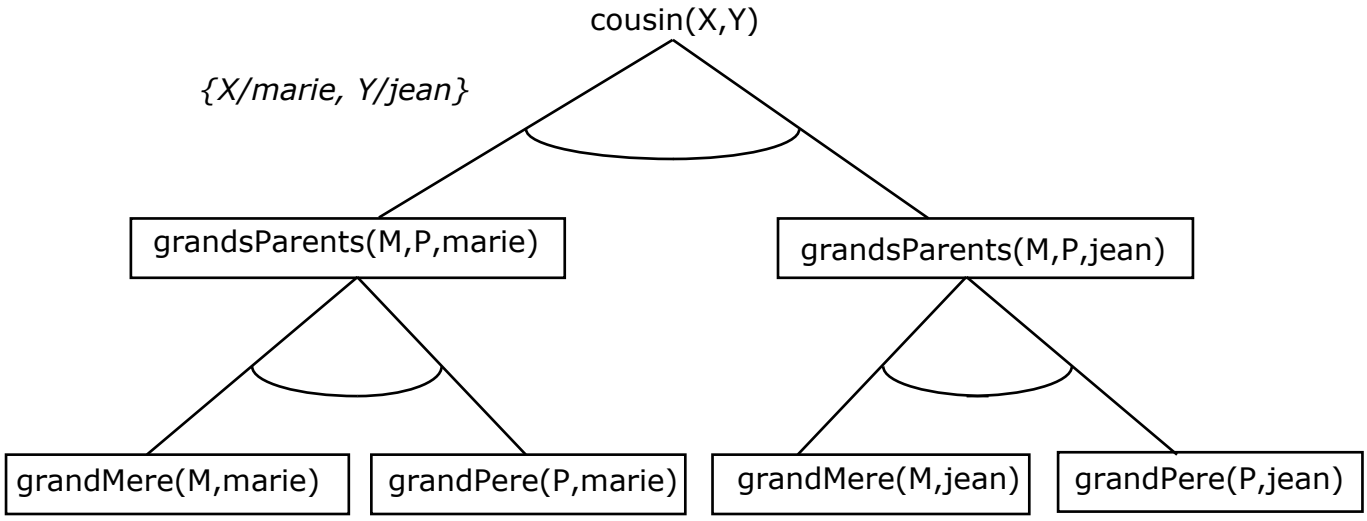
L'unique objectif de la liste à réaliser est maintenant ***{grandsParents(anne,pierre,jean)}.***

Pour réaliser *grandsParents(anne,pierre,jean),* deux sous-objectifs doivent à leur tour être réalisés: *grandMere(anne,jean)* et *grandPere(pierre,jean).* Le sous-objectif ***grandMere(anne,jean)*** est un fait. Quant au sous-objectif ***grandPere(pierre,jean)****,* il est impossible à réaliser.



⇨ Il y a donc **backtracking** jusqu'à l'objectif ayant nécessité l'instanciation de variable la plus récente. Il s'agit en l'occurrence de l'objectif ***grandPere(pierre,marie)*** qui avait nécessité l'instanciation *{P/pierre}.* Ce lien-là est défait et uniquement celui-là. L'objectif ***grandPere(P,marie)*** **est réessayé** avec une autre substitution pour la variable P. Le fait qui suit *grandPere(pierre,marie)* dans la base de connaissances est ***grandPere(jules,marie).***

La liste des objectifs à réaliser est maintenant constituée de *{grandsParents(anne,jules,jean)}.* Pour réaliser l'objectif *grandsParents(anne,jules,jean),* deux sous-objectifs doivent être réalisés: ***grandMere(anne,jean)*** qui est un fait et ***grandPere(jules,jean).*** Le sous-objectif *grandPere(jules,jean)* est cette fois-ci également un fait. La liste des objectifs est vide.



{X/marie, Y/jean, M/anne, ~~P/pierre~~} ⇨ *{X/marie, Y/jean, M/anne,**P/jules**}*

⇨ ***cousin(marie,jean)*** **est donc vrai**. ☺

Un objectif réussit (est satisfait)

- soit s'il **s'unifie à un fait** ;
- soit s'il **s'unifie à un conséquent** (une conclusion) de règle **dont tous les antécédents** (toutes les conditions) **réussissent**.

Règles d'unification de deux prédicats

Deux prédicats s'unifient si :

- ils ont tous deux le **même symbole de prédicat** ;
- ils ont tous deux le **même nombre d'arguments** ;
- leurs **termes s'unifient deux à deux**.

Règles d'unification de deux termes

Une **constante** peut s'unifier à

- la **même constante**,
- **n'importe quelle variable**.

Une **variable** peut s'unifier à

- **n'importe quelle constante**,
- **n'importe quelle** autre **variable**.

En conclusion

① Le moteur d'inférence essaye d'établir un objectif (la question) en **exécutant une chaîne de déduction**. L'output du moteur d'inférence sera donc **YES** ou **NO**.

Si, en outre, apparaissent dans la question des **variables**, un **jeu de valeurs sera proposé pour** ces variables.

② L'ordre de considération des règles (/faits) est statique, c'est-à-dire prédéterminé :

- **Du haut vers le bas pour les faits et règles.**
- **De gauche à droite** pour les conditions d'une règle.

③ En cas **d'échec** dans l'établissement d'un objectif (X), deux traitements sont possibles selon le type d'objectif qui échoue :

- Si X est un **objectif OU** (c'est-à-dire le fils d'un nœud OU)
  - O**n passe à** l'objectif (frère) suivant, c'est-à-dire **la règle suivante qui a le même conséquent (même conclusion)** ;
  - Les instanciations de variables établies en vue de démontrer l'objectif qui a échoué sont annulées, et uniquement celles-là.

- Si X est un **objectif ET** (fils d'un nœud ET)
  - On "rétro-parcourt" jusqu'à l'objectif (Y) qui avait **nécessité la plus récente instanciation de variable ;**
  - On défait cette instanciation-là, et uniquement celle-là ;
  - On repart de l'objectif Y qu'on réessaye avec une autre valeur d'instanciation pour la variable défaite.

8 Récursivité

Une procédure est récursive quand elle "**se rappelle elle-même**".

Le principe de la récursivité est de **réduire un problème général en problèmes de plus en plus simples jusqu'à arriver à un cas trivial** (cas évident).

Soit la notion de supérieur hiérarchique dans une entreprise. Toute personne, mis à part le directeur général, est sous la responsabilité directe d'un autre employé.

Soit la structure d'entreprise suivante :

Leroy (directeur général)
↓
Leduc
↓
Baron
↓
Manuel

Que l'on traduit en Prolog par les 3 faits :

responsableDirect(leroy,leduc).

signifiant que Leroy est le responsable direct de Leduc

responsableDirect(leduc,baron).

signifiant que Leduc est le responsable direct de Baron

responsableDirect(baron,manuel).

signifiant que Baron est le responsable direct de Manuel

On aimerait pouvoir poser les questions suivantes :

superieurHierarchique(leroy,manuel).

Leroy est-il le supérieur hiérarchique de Manuel ?

ou encore

superieurHierarchique(X,manuel).

Qui sont les supérieurs hiérarchiques de Manuel ?

Il faut donc écrire la procédure *superieurHierarchique*.

Toute procédure récursive en Prolog comprend **au minimum deux clauses** :

- au moins une clause décrivant **le cas trivial** et
- au moins une **règle récursive**.

Une règle est récursive si le prédicat formant la conclusion de la règle est "rappelé" avec d'autres arguments dans une des conditions de la règle :

p(X,Y) :- … , p(R,S), … .

Dans l'exemple, tout responsable direct est également un supérieur hiérarchique. Il suffit donc d'être responsable direct de quelqu'un pour être aussi son supérieur hiérarchique. Cette constatation constitue le cas trivial (évident) et donc la première clause de la procédure :

superieurHierarchique(X,Y) :- responsableDirect(X,Y).

Si la question posée est *superieurHierarchique(X,manuel),* cette première clause de la procédure permet de trouver qu'un supérieur hiérarchique de *Manuel* est (entre autres) son supérieur direct, à savoir *Baron*.

Reste à trouver la règle récursive générale. Pour rappel, travailler par récursivité, c'est réduire un problème général en problèmes de plus en plus réduits.

Trouver les supérieurs hiérarchiques d'un employé particulier, revient à trouver son responsable direct puis à rechercher les supérieurs hiérarchiques de ce dernier. On pourrait ensuite continuer de remonter la hiérarchie.

superieurHierarchique(X,Y) :- responsableDirect(Z,Y),
superieurHierarchique(X,Z) .

Si la question posée est ***superieurHierarchique(leroy,manuel),*** cette clause permet de décomposer le problème en sous-problèmes.

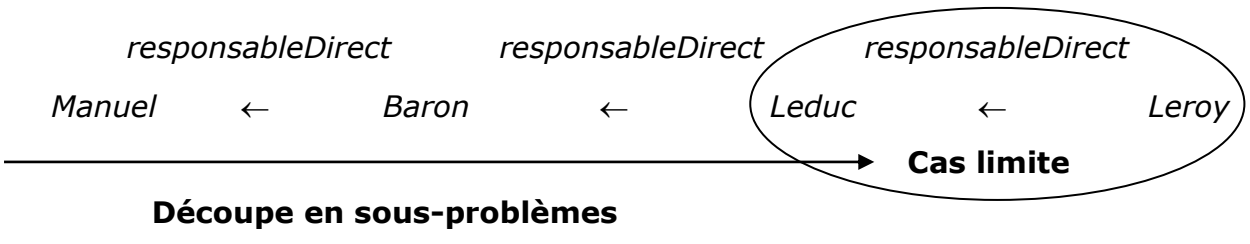
D'après la règle, il suffit de trouver un responsable direct de *Manuel* et de vérifier que *Leroy* est bien un supérieur hiérarchique de ce dernier. Le responsable direct de *Manuel* est *Baron*. Reste à démontrer que *Leroy* est bien un supérieur hiérarchique de *Baron*. On a donc réduit le problème de départ.

	responsableDirect		superieurHierarchique?	
Manuel	←	*Baron*	← … ←	*Leroy*

Pour démontrer que *Leroy* est bien un supérieur hiérarchique de *Baron*, il suffit de trouver un responsable direct de *Baron* et de vérifier que *Leroy* est bien un supérieur hiérarchique de ce dernier. Le responsable direct de *Baron* est *Leduc*. On a encore réduit le problème de départ. Reste à démontrer que *Leroy* est bien un supérieur hiérarchique de *Leduc*. Et ainsi de suite.

	responsableDirect		*responsableDirect*		*superieurHierarchique?*	
Manuel	←	*Baron*	←	*Leduc*	← … ←	*Leroy*

On arrête de réduire le problème en sous-problèmes quand le cas limite, c'est-à-dire le cas trivial, est applicable. Or, ce cas limite est applicable si l'on cherche à démontrer que *Leroy* est bien un supérieur hiérarchique de *Leduc*. En effet, ce cas limite nous indique que pour être un supérieur hiérarchique, il suffit d'être un responsable direct, ce qui est le cas pour *Leroy* par rapport à *Leduc*. CQFD. La réponse à la question *superieurHierarchique(leroy,manuel)* est ***yes.***



Si la question posée est ***superieurHierarchique(X,manuel),*** la règle récursive indique qu'il faut commencer par rechercher les supérieurs hiérarchiques du responsable direct de *Manuel*, à savoir *Baron*.

Pour trouver les supérieurs hiérarchiques de *Baron*, il faut rappeler récursivement la règle sur *Baron*, et donc rechercher les supérieurs hiérarchiques du responsable direct de *Baron*, à savoir *Leduc*.

Pour trouver les supérieurs hiérarchiques de *Leduc*, il faut rappeler la règle récursivement sur *Leduc*. Et ainsi de suite.

Cette décomposition en sous-problèmes s'arrête quand le cas limite, c'est-à-dire le cas trivial est rencontré. Ce qui est le cas lorsque l'on cherche les supérieurs hiérarchiques de *Leduc*. Le cas limite stipule que pour être supérieur hiérarchique de quelqu'un, il suffit d'être son responsable direct, ce qui est le cas pour *Leroy* par rapport à *Leduc*. CQFD.

La réponse à la question *superieurHierarchique(X,manuel)* est ***X = baron***

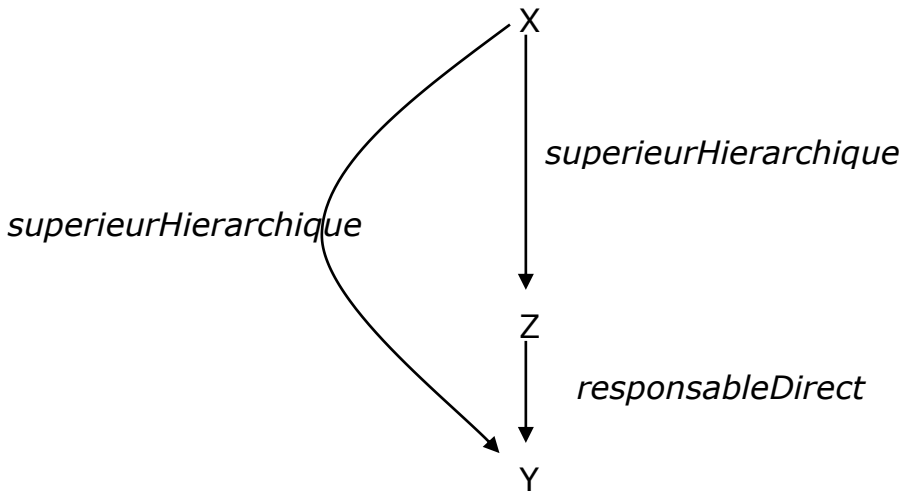
X = leduc

X = leroy

En conclusion, la procédure complète est :

superieurHierarchique(X,Y) :- responsableDirect(X,Y).

superieurHierarchique(X,Y) :- responsableDirect(Z,Y),
superieurHierarchique(X,Z) .

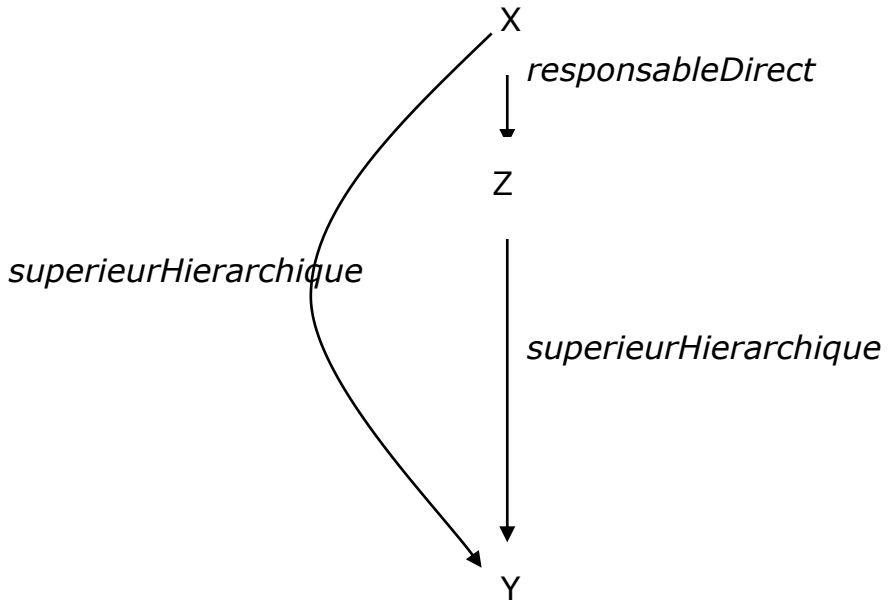


Il est à noter qu'une autre écriture tout aussi valable de la procédure est :

superieurHierarchique(X,Y) :- responsableDirect(X,Y).

(même cas limite)

superieurHierarchique(X,Y) :- responsableDirect(X,Z),
superieurHierarchique(Z,Y) .



Attention : des procédures mal écrites peuvent finir par boucler !

Pour éviter tout bouclage infini il faut respecter deux consignes :

① Placer toujours **en tête de procédure le (ou les) cas limite(s)**.

② Faire attention à **l'ordre des conditions dans les règles récursives**. Il faut éviter de placer en première condition d'une règle l'appel au même prédicat que celui de la conclusion de la règle. Placer les conditions dans un mauvais ordre peut conduire au bouclage de la procédure.

Exemple de procédure qui boucle :

superieurHierarchique(X,Y) :- responsableDirect(X,Y).

superieurHierarchique(X,Y) :- superieurHierarchique(X,Z) *,*
responsableDirect(Z,Y).

Cette procédure finit par boucler quand on l'exploite en posant la question :

superieurHierarchique(X,Y) .

L'affichage obtenu est le suivant :

X= leroy, Y = leduc
X= leduc, Y = baron
X= baron, Y = manuel
X= leroy, Y = baron
X= leduc, Y = manuel
X= leroy, Y = manuel

Ensuite, le message **"Stack overflow"** apparaît. Ce qui signifie que la recherche continue et que la procédure boucle.

9 Tuyaux et recommandations

Pour rappel, toute instruction (fait, règle ou question) Prolog se termine par un **point**.

Variables

💣 Une **variable** commence par une **majuscule**.

*Exemple 1 : reduction(**x**) :- chomeur(**x**).*

> *Ne signifie pas que tout chômeur bénéficiera d'une réduction mais que si la constante x est un chômeur, alors la constante x obtiendra une réduction.*

*Exemple 2 : humain(**A**nne) est équivalent à humain(**X**).*

> *C'est-à-dire que tout objet de l'univers est un être humain !*

Commentaires

Tout commentaire dans le code d'un programme Prolog est précédé de **%**.

Prédicats prédéfinis

Un certain nombre de prédicats binaires (deux arguments) **prédéfinis** existent : **=, <, =<, >= …**

Attention à la syntaxe : ils s'utilisent en notation infixée et non en notation préfixée comme les autres prédicats.

Notation **préfixée** : d'abord le symbole du prédicat suivi des termes entre parenthèses

Exemple : parent(jules, marie)

Notation **infixée** (si prédicat binaire) : d'abord le premier terme suivi du symbole de prédicat suivi du second terme

Exemple : X **>=** *10*

Ces prédicats prédéfinis peuvent être utilisés comme *condition dans des règles*.

Exemple : enfant(X) :- humain(X), age(X,Y), Y<12.

Utilisation erronée du prédicat +

☹ **X = Y+1.** **avec Y non instancié** !

Si la variable Y n'est pas encore unifiée à une constante, l'expression Y+1 n'est pas évaluable. Il y a donc erreur à la compilation.

☹ **X = X+1.**

Deux cas peuvent se présenter :

- Soit **X est inconnu**, c'est-à-dire pas encore unifié à une constante.

  L'expression X+1 n'est donc pas évaluable. Il y a également erreur à la compilation.

- Soit **X est connu** : Exemple : …, X=5, X = X+1, …

  L'expression X+1 est donc bien une expression mathématique évaluable (soit 5+1=6). La nouvelle valeur (6) devrait être ensuite unifiée à X. Cependant, X est déjà unifié à une autre constante (5). Comme **deux constantes ne peuvent s'unifier que si elles sont égales, l'objectif X = X+1 échouera.** La réponse de Prolog sera : **no**.

Variables différentes dans la même règle

Deux variables de noms **différents** utilisées dans les conditions d'une règle **peuvent éventuellement s'unifier à la même constante** !

Par conséquent, si l'on désire que deux variables ne s'unifient jamais à la même valeur, il faut explicitement exprimer cette contrainte en rajoutant une condition à la règle.

Syntaxe :

not (A=B) signifie que les variables A et B doivent être différentes, c'est-à-dire qu'elles ne peuvent être unifiées à la même constante.

Exemple :

*Soit la règle : "Une personne est demi-frère ou demi-sœur (demi-sibling) avec une autre personne si toutes deux n'ont **qu'un seul** parent en commun."*

En Prolog :

demiSibling(X,Y):- parent(P1,X), parent(P1,Y), parent(P2,X), parent(P3,Y),
not (X=Y), not (P1=P2), not (P1=P3), not (P2=P3).

Justifications :

not (X=Y)

obligatoire pour éviter qu'une personne soit considérée comme son propre (demi)sibling.

not (P2=P3)

pour forcer ces deux variables à être différentes
(P2 et P3 concernent les parents qui ne sont pas communs)

not (P1=P2)

Car P1 est le parent commun ; il faut donc forcer P2 et P3 à être différents de P1.

et **not (P1=P3)**

Pas de fonction en Prolog

Il n'y a pas de fonction en Prolog. Il est donc faux d'écrire des expressions du genre :

Y = factorielle(X) ☹

Si le résultat de la factorielle d'un nombre doit être calculé et récupéré dans la variable Y, il faut prévoir un prédicat binaire : le premier argument étant le nombre dont on calcule la factorielle et le second argument étant le résultat :

factorielle(X,Y) ☺