

Module INFO2 : Prolog-Lisp (30h)

partie A (16h)

**« Initiation à la programmation logique
avec gProlog »
dialecte Prolog (syntaxe Edimbourg)**

Support de cours aux planches et vidéos

Jean-Christophe PETTIER
Daniel ROCACHER

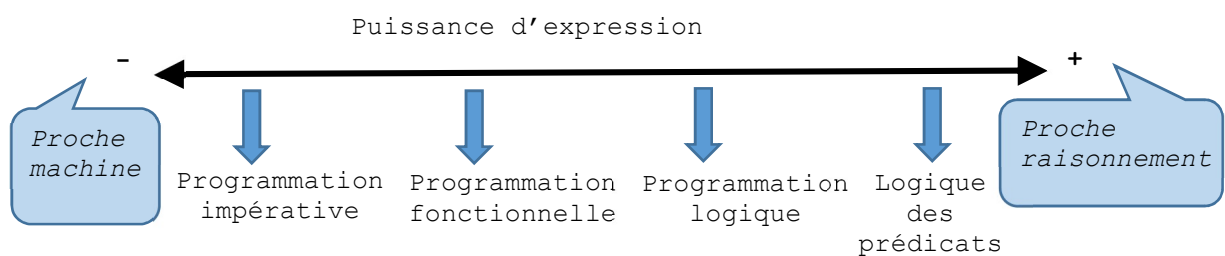
version du 13 janvier 2024

Prolégomènes

Il existe un très grand nombre de langages de programmation et il en apparaît régulièrement de nouveaux. Heureusement les paradigmes de programmation sont moins nombreux. Ainsi, hors programmation concurrente, trois grandes familles de langages sont distinguées : impérative, fonctionnelle, logique. Les concepts issus de ces paradigmes sont parfois associés pour former des langages multi-paradigmes. La programmation orientée-objet est essentiellement impérative mais peut aussi être vue comme orthogonale à ce qui précède. Des langages comme Pharo (ex-Smalltalk) ou plus récemment Python sont à la fois impératifs et fonctionnels dans le sens où ils sont fondés sur la notion d'état et de type abstrait de données et qu'ils portent également dans leur ADN de nombreux gènes de programmation fonctionnelle. Par exemple, ils intègrent la prise en compte de fonctions entités de première classe, c'est-à-dire pouvant être manipulées comme des données. Ce dernier concept (souvent appelé lambda fonction en référence au lambda-calcul, base théorique de la programmation fonctionnelle) est si important qu'il a été relativement récemment introduit dans des langages comme Java (8 et +). Rust est un autre exemple de langage récent offrant quant à lui programmation concurrente et fonctionnelle.

Les langages que nous aborderons pour illustrer ces concepts seront pour la programmation logique : Prolog et pour la programmation fonctionnelle : Racket qui est un dialecte du langage historique Lisp.

☞ Un axe intéressant d'approche des paradigmes de programmation est dans la puissance d'expression ou « *le nombre de lignes de code permettant d'arriver à un résultat* ». La hiérarchie ci-dessous est communément admise car $L1 > L2$ s'il est justifié d'écrire un interpréteur de $L1$ en $L2$.



Connaître les concepts liés à ces trois principales familles de programmation, ainsi que quelques références, permet de se les approprier rapidement et d'être mieux armé pour comprendre les différences entre langages. Puisque dans la hiérarchie des interpréteurs on a $\text{Prolog} > \text{Lisp} > \text{C}$, « à tout seigneur tout honneur, commençons par la plus grande puissance d'expression ! »

1. Introduction

Tout d'abord, vous aurez compris que l'acronyme *Prolog* provient de *Programmation logique*.

1.1. Un peu d'histoire

(non abordé en VIDEO-PPT)

1971 : Mise en œuvre d'un formalisme d'analyse de la langue naturelle et de démonstration automatique. Université de Luminy à Marseille : équipe d'Alain Colmerauer et Philippe Roussel.

Idée ! Ce système peut être élargi => prémisse de Prolog (le premier langage implémenté avant d'avoir été inventé) et utilisation de la logique du 1^{er} ordre comme base formelle de programmation.

1973 : 1^{er} interprète *Prolog*. Université de Marseille.

1977 : 1^{er} compilateur *Prolog*. Université d'Edimburgh, David Warren.

... un langage utilisé dans les laboratoires...

1981 : choix de Prolog comme langage de base d'un vaste projet de recherche au Japon (dit de 5^{ème} génération) => coup de projecteur, une référence dans le monde de l'IA symbolique¹.

1982 : *Prolog II* (Marseille, A Colmerauer) => Popularisation et commercialisation.

1989 : *Prolog III* : introduction de la programmation logique avec contraintes.

1996 : *Prolog IV* : extension *Prolog III* en intégrant des contraintes sur des espaces plus complexes.

¹ L'IA symbolique se différencie de l'IA connexionniste parce que la première part de connaissances générales transmises à la machine par des humains pour résoudre des problèmes et développer des raisonnements, alors que la seconde part d'exemples de solutions qu'elle essaie d'extrapoler par des méthodes statistiques.

1.2. De nombreux dialectes

(non abordé en PPT)

De nombreux dialectes et interprètes de Prolog existent. Ils peuvent être catégorisés selon leur école/syntaxe :

- la syntaxe d'Edimburgh : C-Prolog, Delphia Prolog, Quintus, **Gprolog** (*celui utilisé*), SWI-Prolog
- la syntaxe de Marseille : Prolog II , Prolog III, Prolog IV

Tous les systèmes Prolog sont fondés sur les mêmes concepts de base : une base de connaissances et un moteur utilisant deux mécanismes fondamentaux : l'*unification* et la *résolution*.

Les Prolog ont évolué pour intégrer des notions de contraintes. Par exemple : prise en compte de l'évaluation retardée (freeze) en Prolog II, introduction des contraintes (numériques, booléennes et arbres) en Prolog III et de contraintes complexes (réels, contraintes non linéaires) en Prolog IV.

En cours et TP nous nous intéressons aux concepts de base (*unification, résolution*, plus un mécanisme de *contrôle de la résolution*) pour terminer sur un très rapide regard sur l'intérêt de la programmation logique avec contraintes.

1.3. Installation

(abordé en PPT)

Le langage utilisé est **GNU Prolog** (*Gprolog*) : c'est un langage de programmation logique avec contraintes développé par *Daniel Diaz* à l'INRIA. Il est libre et téléchargeable à partir de : <http://www.gprolog.org/>

Cette note de cours ne détaille pas les spécificités syntaxiques et les différents prédicats prédéfinis de Gprolog. Pour cela, se référer au manuel :

<http://www.gprolog.org/manual/gprolog.pdf>

ou à des polys plus détaillés comme celui de Romain Janvier [*R. Janvier*] :

https://perso.liris.cnrs.fr/alain.mille/enseignements/Master_PRO/BIA/coursProlog.pdf

1.4. Quelles applications ?

(non abordé en PPT)

Le calcul formel / symbolique :

- analyse de la "langue naturelle"
- grammaires, compilation, automates
- intelligence artificielle et modélisation des raisonnements (systèmes experts, diagnostic, . . .) => IA symbolique
- bases de données (du relationnel vers le déductif)
- vérification de programmes, preuve automatique
- conception assistée par ordinateur
- ...

mais pas le calcul numérique !

2. Caractérisation de Prolog

(VIDEO-PPT 1.1)

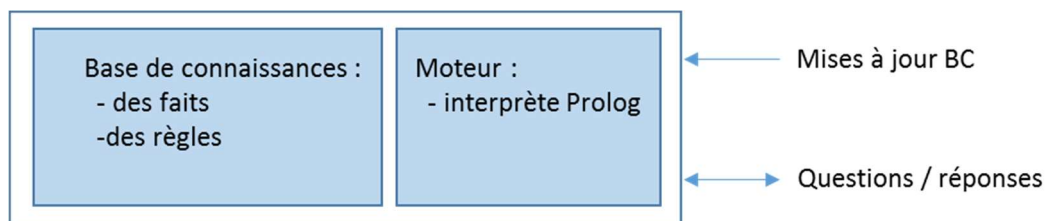
C'est un langage sous-ensemble de la logique (des affirmations vraies, pas de négation), une exécution est une preuve de concept.

⇒ Une rencontre du « 3^{ème} type » :

- pas de : variable désignant un(des) mot(s) mémoire, déclaration, affectation, structures de contrôle (if, while, goto)... rien d'habituel ;
- mais : des relations entre données (prédicats) et des solutions pour un problème à résoudre => non déterminisme.

2.1. Une première caractéristique : pas de contrôle de l'exécution !

Prolog est déclaratif : on exprime le *quoi* (la connaissance, ce qui est vrai) et pas *comment* la machine doit résoudre un problème. Les informations de la base de connaissances (données) sont décrites logiquement sans se préoccuper de la manière dont elles seront utilisées ultérieurement. Le niveau machine (mémoire/variable, séquence de commandes) est masqué. Une exécution est déclenchée par une interrogation. Le moteur (contrôle) applique le principe de raisonnement logique sous-jacent pour trouver toutes les solutions à une question.



En Prolog on exprime 3 choses :

- des faits ;
- des règles ;
- des questions.

Programmer en Prolog c'est :

- mettre à jour la base de connaissances ;
- interroger la base de connaissances.

👉 *Le contrôle de l'exécution ne relève plus du programmeur comme en programmation impérative mais du moteur Prolog.*

Un premier programme :

faits	<code>mot('hello').</code>
	<code>mot('world').</code>
une règle	<code>doubleMot(X,Y) :- mot(X), mot(Y), X\=Y.</code>
une question	<code>doubleMot(X, Y), write(' '), write(X), write(' '), write(Y).</code>

Vous pouvez y remarquer que

- les faits et règles se construisent sur des prédicats en minuscules
- les constantes littérales chaînes de caractères se construisent sur des quotes
- les majuscules (en 1^{er} caractère) sont des variables
- Les items lexicaux de prolog sont le point (termine toute ligne), la virgule (séparatrice de buts et dans les règles le :- séparant à gauche un but (conséquence) vrai si la suite des sous-buts à droite peut être satisfaite

Note : $X \neq Y$ correspond à une contrainte imposant que les variables X et Y doivent avoir des valeurs différentes

En *Gprolog* la base de connaissances est saisie dans un fichier texte avec l'extension *.pl* (*helloWorld.pl*). Celui-ci est ensuite chargé sous l'interpréteur en exécutant la commande :

```
consult('helloWorld.pl').
```

puis peut être rappelé avec la flèche <↑>. La commande *listing.* permet de s'assurer du bon chargement dans la base de connaissances chargée.

Une première interrogation via une pseudo-règle permettant de mémoriser la question :


```
question :- doubleMot(X, Y), write(X), write(' '), write(Y).
```

donnera :

```
|?- question.  
hello world  
  
true ? ;  
world hello  
  
true ? <RC>  
yes
```

Prolog essaie d'effacer le but *question* et propose un premier enchaînement *hello world*. En répondant ; à la poursuite de l'interrogation, nous demandons à l'interpréteur de chercher un second enchaînement *world hello* puis la saisie du *Retour Charriot* arrête la recherche.

Note : en répondant a, l'interpréteur fournit tout de suite toutes les possibilités d'enchaînement.

 *L'exécution d'un programme Prolog à consister à effacer la succession de buts dans la question, ici :*

- 1) *doubleMot(X, Y)*
- 2) *write(' ')*
- 3) *write(X)*
- 4) *write(' ')*
- 5) *write(Y)*

En remplaçant la tête de clause *doubleMot(X, Y)* par sa queue *mot(X) mot(Y)* avec la contrainte X et Y différents, puis en unifiant *mot(X)* puis *mot(Y)* avec les faits, on obtient les deux couples (X,Y) affichés.

☞ *Prolog gère les possibilités multiples d'effacement de but (l'indéterminisme) par l'énumération de toutes les possibilités.*

Il est évidemment possible d'échouer, l'interpréteur n'arrivant pas à établir la « vérité » de la question par effacement de buts :

```
| ?- doubleMot('hello','world').  
yes  
| ?- doubleMot('hello','hello').  
no
```

mini TP : En utilisant le prédicat `avant(X,Y)` fourni, filtrer la question pour n'afficher que le seul succès 'hello world'

2.2. Des prédicats primaires définis par des faits

(VIDEO-PPT 1.2)

Un *fait* est une relation entre des objets qui est toujours vraie.

Exemple : « Joseph est le père de John ».

- ➔ Une relation : « père »
- ➔ Des objets : « Joseph », « John »

Ce qui se représente par le fait `pere(joseph, john)`.

Vocabulaire :

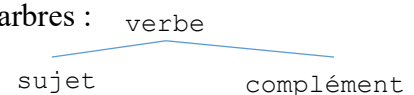
- `pere` est un prédicat (ici binaire) primaire (construit sur des faits)
- `joseph` et `john` sont des arguments (ici des constantes symboliques)

☞ *La syntaxe requiert que prédicats et constantes symboliques commencent par une minuscule.*

Autres exemples de Langue Naturelle traduits en Prolog :

Expression en LN	Notation Gprolog
Toto parle à Luc	<code>parleA(tom, luc)</code>
L'or est précieux	<code>estPrecieux(or)</code>
Max donne Milou à Jean	<code>donne(max, milou, jean)</code>

L'ordre d'écriture des arguments de la relation que décrit un prédicat est arbitraire mais il est choisi une fois pour toutes par le programmeur. L'habitude d'écriture est : `verbe(sujet, complément)`. En représentation interne, Prolog manipule des arbres :



Prolog n'attache pas de sens aux mots, il s'appuie sur des traitements syntaxiques. Ainsi, si pour une expression, le programmeur décide que le premier argument de la relation correspond au sujet, il faut garder ce positionnement pour toute la base de connaissances.

Les faits :

```
parleA(toto, luc).  
parleA(luc, toto).
```

expriment deux choses différentes. Prolog voit deux arbres dont les arguments sont différents et ne gère pas la commutativité.

Créons maintenant une véritable base de connaissances, nous nous servirons pour l'exemple de la généalogie de John Kennedy, extrait de `familleKennedy.pl` :

```
pere(joseph, john).  
pere(joseph, robert).  
pere(joseph, edward).  
pere(john, john_john).  
pere(john, caroline).  
pere(robert, kathleen).  
  
mere(rose, edward).  
mere(rose, john).  
mere(rose, robert).  
mere(rose, rose_marie).
```

☞ *Tous les faits définissant un même prédicat sont regroupés en séquence dans l'ordre énoncé par l'utilisateur. Ils forment un paquet de clauses. Cet ordre n'a pas de sens logique (et donc pas d'importance) mais, d'un point de vue pratique, cela facilite les recherches de l'interpréteur.*

2.3. Des prédicats secondaires construits par des règles

Les règles expriment des relations conditionnelles entre des prédicats. Ainsi :

« X est grand-mère de Y s'il existe Z tel que X est mère de Z
et (Z est mère de Y ou Z est père de Y) »

soit :

$$(\forall x \forall y \forall z (mère(x,z) \wedge mère(z,y)) \Rightarrow grand_mère(x,y))$$

∧

$$(\forall x \forall y \forall z (mère(x,z) \wedge père(z,y)) \Rightarrow grand_mère(x,y))$$

ce qui se développe avec deux implications à « but » unique avec des conjonctions implicites, soit des expressions de la forme :

$sous-but_1 \text{ sous-but}_2 \dots sous-but_n \Rightarrow but$

ce qui s'écrit en Prolog avec deux règles d'effacement de but (noter l'inversion entre prémisses et conclusion) :

```
grand_mere(X,Y) :- mere(X,Z), mere(Z,Y).  
grand_mere(X,Y) :- mere(X,Z), pere(Z,Y).
```

et se lit pour la première règle « *pour prouver* `grand_mere(X,Y)`, *il faut prouver* `mere(X,Z)` *puis* `mere(Z,Y)`. »

miniTP : établissez que

- 1) Rose est la grand-mère de Caroline mais pas de Robert
- 2) Robert est le frère de John
- 3) John est le frère de Robert !

☞ Une question fermée (pas de variable en entrée) peut être vérifiée de plusieurs façons. Puisque la réponse est juste binaire (vrai/faux, sans variable en sortie), seul le premier succès importe et il suffit de stopper la recherche d'autres possibilités en répondant `<RC>` au ? de l'interpréteur.

2.4. Approche de l'effacement de buts

(VIDEO-PPT 1.3)

La base de connaissances `familleKennedy2.pl` étant chargée sous l'interpréteur Gprolog, on a vu qu'on pouvait l'interroger.

⇒ Interrogation avec des questions fermées (sans variable) :

➤ John est-il le père de John-john ?

```
|?- pere(john, john_john).  
true ? <RC>  
yes
```

Prolog a trouvé un premier succès, c'est-à-dire une solution au problème. Si on avait demandé à prouver quelque chose de faux, on a vu que la réponse est négative.

```
|?- pere(john, jacqueline).  
no
```

L'interpréteur n'a pas réussi à unifier la question avec un fait, ce qui veut dire « pour autant que je sache, d'après ma base de connaissances, je ne peux pas conclure que `john` est le père de `jacqueline` ».

☞ *En Prolog, tout ce qui ne peut être prouvé dans la base de connaissances est considéré comme faux, c'est l'hypothèse du monde clos ².*

⇒ Interrogation avec des questions à inconnues :

➤ De qui John est-il le père ?

```
|?- pere(john, X).  
X=john_john ? ;  
X=caroline  
yes
```

Dans la question il y a une inconnue *x* (la première lettre de l'identificateur est en majuscule), c'est une variable Prolog dont l'existence d'une valeur possible est questionnée. Pour résoudre le problème, Prolog explore le paquet de faits (clauses) *pere* de la base de connaissances et cherche à mettre en correspondance la question avec un fait en instanciant (attribuant une valeur à) *x*.

☞ *Les variables Prolog ont un sens logique, elles sont libres (non instanciées) ou liées (associées définitivement à une valeur pour la question ou la règle dans laquelle elles apparaissent). Dans un langage impératif, une variable correspond à un mot mémoire (état) dans lequel des valeurs différentes sont stockées à des instants différents (affectations). Cette notion de valuation différente au cours du temps n'existe pas en logique, une variable est libre ou liée définitivement. On verra cependant qu'elle peut être liée à un terme composé comprenant d'autres variables, libres lors l'instanciation de la variable les englobant et qui seront liées ultérieurement.*

⇒ Interrogation avec des questions composées :

➤ Quelles ont été les épouses de John ?

On conviendra que deux individus ont été époux, s'ils ont eu un enfant en commun. La question peut alors se traduire par : « si John est le père de *x*, existe-t-il *y*, tel que *y* soit la mère de *x* ? »

```
|?- pere(john, X), mere(Y,X).  
X=john_john  
Y=jacqueline ? ;  
  
X=caroline  
Y=jacqueline
```

Dans cette question ouverte, comme dans la question fermée « grand-mère » vue précédemment, il y a 2 buts séparés par une « , » qui correspond à une **conjonction**.

² Il n'y a pas de négation. Cependant, en section V.2.2 il est montré qu'un prédicat *not* peut-être mis en œuvre en agissant sur le mécanisme d'évaluation de Prolog, (cf. négation par échec).

Prolog évalue les buts de **gauche à droite**³ :

1. recherche d'une solution (instanciation) X au but `pere(john, X)`
2. **puis**, avec la solution X , recherche des solutions Y satisfaisant `mere(Y, X)`
3. **retour** sur `pere(john, X)` pour la recherche d'une autre solution X
4. **pour chaque** solution X , on recherche des solutions Y de `mere(Y, X)`

Dans une conjonction de buts, X représente le même objet pour tous les buts. Pour une solution X trouvée (X est liée / instanciée), Prolog cherche toutes les solutions Y de `mere(Y, X)` puis Prolog revient sur le but `pere(john, X)` pour trouver éventuellement une autre solution X ... Le retour sur le premier but (backtracking) délie (rend libre) X .

☞ *Prolog gère l'indéterminisme, ie le fait qu'il y ait deux façons d'effacer le but `pere(john, X)` puis une pour `mere(Y,X)` par l'énumération de toutes les possibilités d'instanciation des couples (X,Y)*

miniTP : En utilisant « familleKennedy2.pl », trouvez tous les frères de John, observez la succession des succès :

- 1) Pourquoi les succès sont-ils doubles pour `robert` et `edward` et pas pour `rose_marie` ?
- 2) Assurez-vous que les questions `frere(john,X)` et `frere(X,john)` fournissent les mêmes réponses.
- 3) Vérifiez la correspondance de l'énumération avec la succession des faits et règles dans le fichier
- 4) La variable X apparaissant à la question 2) existe aussi dans les règles définissant le prédicat `frere/2`, pourrait-on la remplacer par une autre lettre dans un des deux contextes (question ou règles) ?

☞ Rappels cours précédent

(VIDEO-PPT 2.1)

- Base de connaissances = faits + règles = paquets de clauses
- Règle \equiv `tete :- queue.`
 - `tete` = 1 but (conclusion de la règle)
 - `queue` = conjonction de sous-buts (hypothèse de la règle)
- Hypothèse du monde fermé : si une information n'est pas présente dans la base de connaissances, elle est considérée comme fausse.
- Variable *Prolog* : une variable *Prolog* est une inconnue qui est soit libre, soit liée, elle ne peut être instanciée (liée) qu'une seule fois.

³ C'est un choix d'implémentation pour l'interpréteur car, d'un point de vue logique, il n'y a pas d'ordre entre les buts.

- Portée d'une variable : toute la règle, rien que la règle

2.5. Catégories syntaxiques

Cette section ne présente que quelques éléments de syntaxe. Pour plus de précisions, cf. le manuel utilisateur de Gprolog.

i) Clauses

La base de connaissances définit des faits (sans variable) et des règles (avec variables a priori) :

- `pere(john, john_john).`
- `grandPere(X, Y) :- pere(X, Z) , pere(Z, Y).`

Sous l'interpréteur, les questions ont la forme :

- `mere(X, caroline).`

ii) Termes

Les objets manipulés par Prolog sont des termes, on distingue : les atomes, les variables, les nombres, les termes composés.

- **Atomes :**

- **Constante symbolique** : suite de caractères, nombres, tirets bas, commençant par une *lettre minuscule* :

```
daniel
x007
```

- **Chaine de caractères** :

```
`toto'
```

- **Variables**

- Suite de caractères, nombres, tirets bas, commençant par une *lettre majuscule*

```
Daniel
X007
X
```

- **Nombres**

- Des **entiers** : 1
- Des **flottants** : 3.14

- **Termes composés**

Un terme composé est une donnée représentant un arbre décrit sous une forme : `foncteur(arg1, ..., argn)`. La racine (appelée foncteur) est une constante symbolique. Les arguments sont des termes.

➤ **Fait :** `possede(jean, voiture(electrique))`.

Ce fait caractérise (partiellement) une relation `possede` portant sur deux termes `jean` et `voiture(electrique)`. Attention, ce dernier terme **n'est pas un prédicat**⁴ (qu'il faudrait évaluer) mais une **donnée structurée** représentant un arbre à deux arguments (qui n'est pas évalué).

➤ **Questions :**

```
|?- possede(jean, X).
X = voiture(electrique)

|?- possede(jean, voiture(X)).
X = electrique
```

Les arguments d'un terme composé peuvent à leur tour être des termes composés comme ici :

➤ **Fait :**

```
possede(jean, livre(lesRobots, auteur(isaac, asimov))).
```

➤ **Question :**

```
|?- possede(jean, livre(X, Y)).
X= lesRobots, Y = auteur(isaac, asimov)
```

Si un terme composé n'est pas un prédicat, il peut cependant le devenir. Ainsi, il est possible de compléter la base de connaissances dans « familleKennedy3.pl » avec la règle :

```
complete_pere_par_mere :-      epoux(X,Y), mere(Y,Z),
                                non(pere(X,Z)),
                                assertz(pere(X,Z)).
```

qui utilise l'arbre `pere(X,Z)` avec les liaisons de variable `{X=joseph, Z=rose_marie}` pour ajouter un fait supplémentaire augmentant le prédicat `pere` avec l'usage du méta-prédicat `assertz`.

Remarques :

⁴ *Prolog* est fondé sur la logique du 1^{er} ordre : un prédicat ne peut pas avoir un prédicat en paramètre

- `non` est un méta-prédicat défini (`not` absent de *Gprolog*), nous reviendrons ultérieurement sur sa mise en œuvre
- pour permettre l'ajout ou le retrait de faits en *Gprolog* avec, il faut que le prédicat ait été rendu « dynamique »

👉 le préfixe « méta » est utilisé ici dans son acception informatique classique pour laquelle un « méta-*X* » est « un.e *X* qui traite des *X* ». par exemple, en programmation objet, une « méta-classe » est « une classe sur une classe ».

• Listes (en extension)

Une liste s'écrit sur des termes. Elle peut s'énumérer en extension :

<code>[]</code>	liste vide
<code>[pierre]</code>	liste à 1 élément (constante)
<code>[X, pierre]</code>	liste à 2 éléments (une variable, une constante)
<code>[pierre, X, paul]</code>	liste à 3 éléments (constante, variable, constante)
<code>[X1, ..., Xn]</code>	liste à <i>n</i> éléments

ou par emboîtement développé à droite d'arbres binaires (présenté plus loin en récursivité).

miniTP : En utilisant « `arbresComposes.pl` », cherchez si Jean (ou tout autre personne) :

- 1) Aime un « type » d'animal *x* ?
- 2) Aime les livres écrits par un auteur (désigné par son seul nom de famille) ?

3. Sémantique

(VIDEO-PPT 2.2)

3.1. Chainages déductifs

Problème : trouver une méthode automatique de résolution donnant une réponse à une question.

Différentes stratégies peuvent être développées pour répondre à une question.

Chainage avant : on part des faits et on applique les règles pour déduire des conclusions (mécanisme de déduction) :

faits \Rightarrow \Rightarrow conclusions.

- Un médecin constate : fièvre + boutons \Rightarrow ... \Rightarrow rougeole ou roséole ou scarlatine

Chainage arrière : on part d'une hypothèse (conclusion) et on « remonte » les règles pour trouver des faits qui confirment ou infirment l'hypothèse (mécanisme d'abduction) :

hypothèse \leq \leq faits.

- Un médecin émet l'hypothèse : supposons roséole \leq ... \leq quel âge a le patient ?
- En pratique, le médecin procède à la fois en chaînage avant (partant des symptômes, il en conclut les maladies possibles) puis en chaînage arrière (partant d'une maladie possible, il interroge le patient ou fait des examens pour obtenir des données complémentaires (faits) infirmant ou confirmant la maladie).

Le système *Prolog* se base sur un mécanisme de chaînage arrière (on part d'une question on remonte vers les faits).

3.2. Sémantique opérationnelle (SLD-résolution)

La SLD-résolution (SLD signifie Sélection Linéaire Définie) est un algorithme évaluant les buts à effacer en chaînage arrière.

➤ Reprenons l'exemple de la section précédente réduit à un sous-ensemble de clauses :

1. `grand_mere(X,Y) :- mere(X,Z) , pere(Z,Y) .`
2. `mere(rose, john) .`
3. `mere(rose, robert) .`
4. `pere(john, caroline) .`

Poser la question `grand_mere(rose, caroline)` avec une interprétation procédurale revient à effacer le but `grand_mere(rose, caroline)`.

Pour cela, on cherche dans la base de connaissances une règle qui peut être appelée en fonction du but à résoudre :

- 1 et `{X = rose, Y = caroline}` exprime que résoudre `grand_mere(rose, caroline)` revient à résoudre la conjonction de buts `mere(rose,Z)`, `pere(Z, caroline)`. Cette conjonction de buts est appelée **résolvante**.

L'algorithme SLD-résolution évalue la conjonction de buts `mere(rose,Z)`, `pere(Z, caroline)` en séquence de la gauche vers la droite.

NB : cette évaluation gauche-droite est une spécificité par rapport à la logique mathématique où cette notion de séquençement n'a pas de sens (les buts pourraient être traités dans un ordre différent).

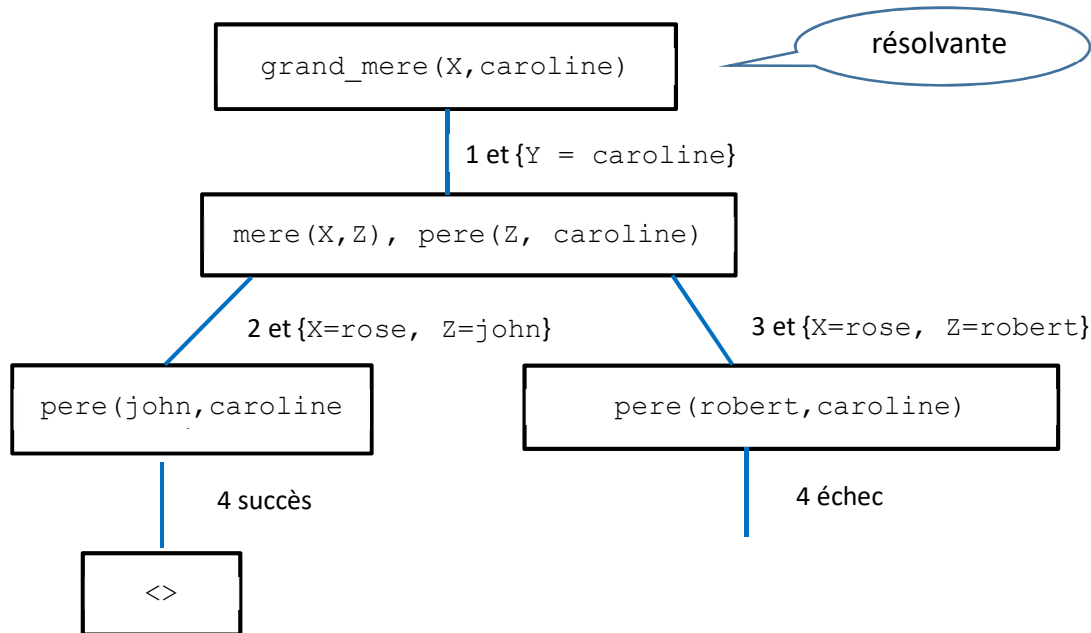
Prolog cherche donc d'abord à effacer : `mere(rose, Z)` :

- La clause 2 + `{Z = john}` efface ce but. La résolvante se réduit à `pere(john, caroline)` qui se résout par 2 puis 3 :
 - 2 ne s'unifie pas à ce but et donc ne l'efface pas : c'est un **échec**.
 - 3 ne s'unifie pas à ce but et donc ne l'efface pas : c'est un **échec**.
 - 4 efface ce but. La résolvante est vide : c'est un **succès**.

- La clause 3 + {Z = robert} efface ce but. La résolvente se réduit à `pere(robert, caroline)` qui ne s'unifie sur aucune clause, c'est un **échec**
- La clause 4 ne s'unifie pas à `mere(rose, Z)`, c'est un **échec**.

Arbre de résolution (de recherche)

Posons la question ouverte `grand_mere(X, caroline)`.



Pour résoudre une question, Prolog construit l'arbre de recherche issu de la résolvente formée de la question. L'arbre de recherche est parcouru en profondeur d'abord et de gauche à droite. Les nœuds terminaux sont des nœuds de succès ou d'échec :

- nœud de succès : la résolvente est vide. Prolog est arrivé à une solution, il l'affiche et cherche d'autres solutions en remontant dans l'arbre jusqu'à un point de choix possédant des branches non explorées (retour arrière) ;
- nœud d'échec (notée \square) : remontée dans l'arbre jusqu'à un point de choix possédant des branches non explorées.

➤ Autre exemple (synthétique), base de connaissances :

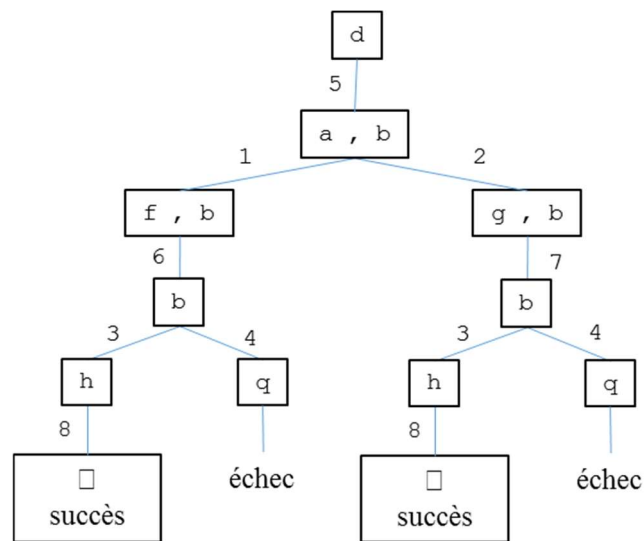
```

1. a :- f.
2. a :- g.
3. b :- h.
4. b :- q.
5. d :- a, b.
6. f.
7. g.
8. h.

```

Question : `?- d.`

Arbre de recherche :



➤ *miniTP* : Soit le prédicat récursif ci-dessous qui définit le concept d'ancêtre :

1. `ancetre(X, Y) :- mere(X, Y) .`
2. `ancetre(X, Y) :- pere(X, Y) .`
3. `ancetre(X, Y) :- mere(X, Z) , ancentre(Z, Y) .`
4. `ancetre(X, Y) :- pere(X, Z) , ancentre(Z, Y) .`

- a) Décrivez l'arbre de résolution correspondant à la question `ancetre(X, caroline)` .
- b) Pourquoi l'ordre des littéraux dans la queue de clause 4 est ici important ?

On rappelle les faits disponibles :

5. `mere(rose, john) .`
6. `mere(rose, robert) .`
7. `pere(john, caroline) .`

3.3. Sémantique déclarative

(VIDEO-PPT 2.3)

Prolog a pour fondement théorique la logique du 1^{er} ordre. Il en est proche mais il est plus restreint (notamment : perte de la négation, preuves limitées à des littéraux positifs).

La logique propositionnelle et la logique du 1^{er} ordre ont été étudiées en INFO 1. Dans cette section, il s'agit de montrer le lien entre Prolog et la logique des prédicats (*hors récursivité*).

La sémantique déclarative de Prolog consiste à donner une signification logique aux clauses Prolog.

Forme clausale

Problème : une expression logique peut prendre différentes formes équivalentes :

$$\begin{aligned} \triangleright A \Rightarrow B &\equiv \neg A \vee B \\ A \wedge B \Rightarrow C &\equiv \neg A \vee \neg B \vee C \end{aligned}$$

☞ *La manipulation automatique de telles formules conduit à les représenter d'une seule manière appelée forme clausale, c'est une formule logique composée d'une disjonction de termes positifs et de termes négatifs. Les variables sont quantifiées universellement.*

$$\begin{aligned} &L_1 \vee L_2 \dots \vee \neg L_p \vee \neg L_{p+1} \vee \dots \\ &\forall x \text{ homme}(x) \vee \text{femme}(x) \vee \neg \text{humain}(x) \end{aligned}$$

La clause vide, notée \square , ne contient aucun littéral : elle est toujours fausse.

Clause de Horn

☞ *Une clause de Horn est une forme clausale avec au plus un littéral positif*

$$L_1 \vee \neg L_2 \dots \vee \neg L_n$$

- La clause de Horn : $\forall X, Y, Z \text{ grandPere}(X, Y) \vee \neg \text{pere}(X, Z) \vee \neg \text{pere}(Z, Y)$
peut se réécrire : $\forall X, Y, Z \text{ grandPere}(X, Y) \Leftarrow \text{pere}(X, Z) \wedge \text{pere}(Z, Y)$
ou : $\forall X, Y \text{ grandPere}(X, Y) \Leftarrow \exists Z (\text{pere}(X, Z) \wedge \text{pere}(Z, Y))$

Formulation logique des règles

Une clause *Prolog* est une clause de Horn. Notation *Prolog* : $L_1 :- L_2, \dots, L_n$.

$$\begin{array}{ccc} \triangleright & \underbrace{\text{grandPere}(X, Y)}_{\text{conclusion}} & :- \underbrace{\text{pere}(X, Z), \text{pere}(Z, Y)}_{\text{hypothèses}} \\ & & \Leftarrow \end{array}$$

Si dans une règle *Prolog* une variable apparaît avant ou (avant et après) l'opérateur $:-$, le quantificateur associé est \forall . Quand une variable n'apparaît qu'après l'opérateur $:-$ le quantificateur associé est \exists .

➤ $\forall X, Y$ si $\exists Z$ tel que $(\text{pere}(X, Z) \text{ et } \text{pere}(Z, Y))$ est vrai alors $\text{grandPere}(X, Y)$ est vrai.

Un fait correspond à une clause de *Horn* sans littéral négatif mais avec seulement un littéral positif (clauses de Horn positive). En Prolog un fait correspond à une clause sans queue.

Résolution logique

En logique, pour prouver que la formule f est une conséquence logique des formules $\{f_1, f_2, \dots, f_n\}$, on démontre que l'ensemble $\{f_1, f_2, \dots, f_n, \neg f\}$ est inconsistant. Autrement dit, on démontre que le système $\{f_1, f_2, \dots, f_n, \neg f\}$ conduit à une contradiction (n'est pas satisfiable). C'est une démonstration par l'absurde ou **réfutation**, on va chercher à établir que système permet d'inférer $\{f, \neg f\}$ qui est un sous-ensemble inconsistant (on ne peut avoir quelque chose et son contraire).

Pour montrer qu'un système d'équations logiques $\{f_1, f_2, \dots, f_n, \neg f\}$ est inconsistant, on utilise la règle de résolution :

$$\frac{P \vee Q, \neg P \vee R}{Q \vee R}$$

dans laquelle les formules P et R peuvent être absentes et donc dans le cas où elles le sont toutes les deux l'obtention de la réfutation :

$$\frac{P, \neg P}{\square}$$

Rappel de logique : Si, en introduisant la négation de f dans $\{f_1, f_2, \dots, f_n\}$, on peut déduire une contradiction $\{f_i, \neg f_i\}$ (qui se réduit en la clause vide notée \square), alors on peut conclure que f se déduit de $\{f_1, f_2, \dots, f_n\}$, f est donc vrai.

Ainsi, poser en Prolog une question « est-ce que Q est vrai ? » revient à montrer : Q soit $\neg \square \Rightarrow Q$ ($\neg \square$ notant « vrai ») qui correspond à l'emploi de la méthode de résolution logique dans sa contraposée $\neg Q \Rightarrow \square$ (on insère l'hypothèse $\neg Q$ et on établit l'inconsistance \square).

L'implication recherchée $\neg \square \Rightarrow Q$ s'écrit en Prolog $Q :- \neg \square$. En Prolog on omet l'écriture de $:- \neg \square$, une question Q est une clause sans queue (puisque celle-ci est vraie).

➤ Application sur l'exemple déjà vu en sémantique opérationnelle :

1. $\neg \text{mere}(X, Z) \vee \neg \text{pere}(Z, Y) \vee \text{grand_mere}(X, Y)$.
2. $\text{mere}(\text{rose}, \text{john})$
3. $\text{mere}(\text{rose}, \text{robert})$

4. `pere(john, caroline).`

Pour montrer `grand_mere(rose, caroline)`, on introduit dans le système la négation

5. `¬grand_mere(rose, caroline).`

De 1 et 5 avec $\{X = \text{rose}, Y = \text{caroline}\}$, on déduit :


6. `¬mere(rose, Z) ∨ ¬pere(Z, caroline)`

De 6 et 4 avec $\{Z = \text{john}\}$, on déduit :

7. `¬mere(rose, john)`

De 7 et 2 on déduit la clause vide \square (faux/contradiction). Donc `grand_mere(rose, caroline)` se déduit des équations 1., 2., 3. et 4. C'est une proposition vraie.

Résumé

 Poser une question, c'est introduire dans le système, la négation de la question et montrer que l'ensemble des clauses est inconsistent (démonstration par l'absurde ou réfutation). Pour cela, il faut réussir à engendrer la clause vide en appliquant la règle de résolution autant de fois que nécessaire.

Il n'y a pas de mini-TP sur cette section établissant la liaison entre Prolog et logique mathématique.

4. La récursivité sous différentes facettes

(VIDEO-PPT 2.4)

Dans ce chapitre quelques principes de bases de la programmation en Prolog sont analysés. Les techniques mises en œuvre s'appuient sur les fondements du langage, l'unification, la réversibilité et la récursivité.

4.1. La récursion dans les données

- **Listes**

Rappel : les listes ont déjà été introduites dans leur formulation en extension

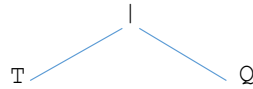
➤ *Exemples* :

- `[aa, 2, 'abc']` est une liste (hétérogène, Prolog n'est pas typé) à 3 atomes
- `[1, [aa, 2, 'abc'], 'de']` est une liste à 3 éléments dont le 2eme élément est lui-même une liste

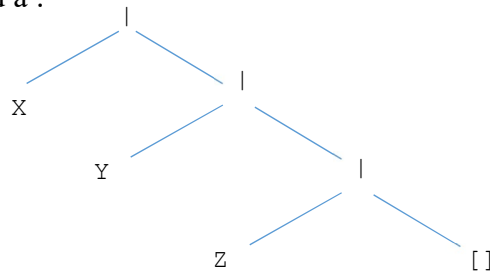
Passons à leur définition en compréhension par récursion :

Une liste L en Prolog s'écrit comme :

- la liste vide $[]$
- une liste ayant au moins un élément dans un arbre binaire $[T|Q]$, $T(ete)$ étant l'élément de tête et Q étant la liste initiale L privée de son 1^{er} élément (c'est donc aussi une liste).



Une telle liste est un terme composé binaire dont le foncteur est la barre verticale $|$ et qui se développera dans sa définition récursive sur le fils droit $Q(ueue)$. Ainsi $[X | [Y | [Z | []]]]$ correspond à :



Les listes sont des arbres binaires « dégénérés » au sens où ils se développent sur leur fils droit. Les listes sont parfois qualifiées de « peignes ».

Raccourcis syntaxiques

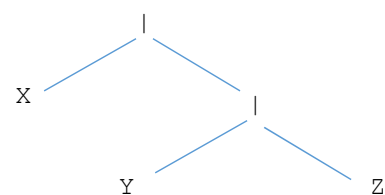
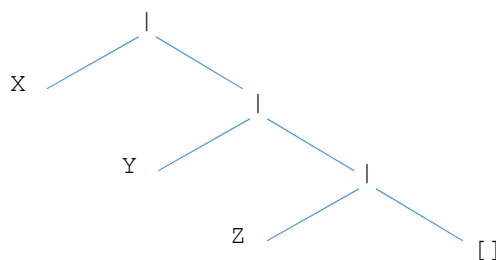
La liste a un élément $[X]$ est un raccourci pour $[X|[]]$.

L'opérateur « virgule » déjà rencontré est en fait un raccourci syntaxique « d'emboîtement » :

- $[X, Y]$ pour $[X|[Y]]$: liste à 2 éléments
- $[X, Y, Z]$ pour $[X|[Y|[Z]]]$: liste à 3 éléments
- $[X, Y | Z]$ pour $[X|[Y | Z]]$: liste (pour peu que $[]$ la termine) qui contient au moins 2 éléments

👉 *Attention* : $[X, Y, Z]$ est différent de $[X, Y | Z]$.

La première liste représente une liste à 3 éléments, la seconde représente une liste (pour peu que $[]$ la termine) qui contient au moins 2 éléments (Z correspondant à la liste moins ses deux premiers éléments). Comparez :



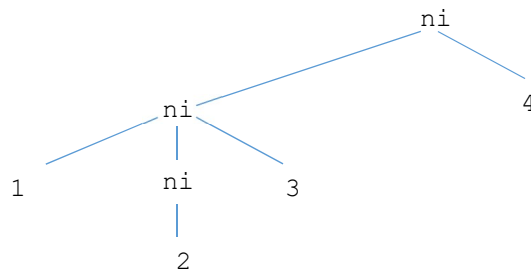
➤ *Exemples* : en supposant l'existence d'un prédicat $eq(X, Y)$ qui réussit si X est unifiable à Y :

- $eq([1, 2, 3], [X|Y])$ fournira $\{X=1, Y=[2, 3]\}$
- $eq([1, 2, 3], [X, Y])$ échouera (impossible d'unifier des listes à 3 et 2 éléments)
- $eq([1, 2, 3], [X, Y | Z])$ fournira $\{X=1, Y=2, Z=[3]\}$
- $eq([1, 2, 3], [1, 2 | [3]])$ réussira

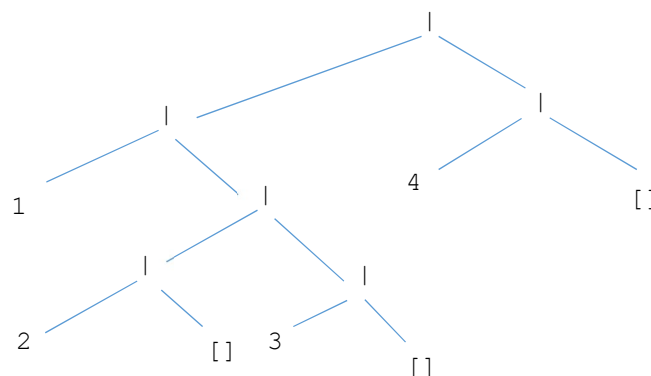
• Arbres binaires d'énumération

La structuration de données peut être nécessaire dans plus d'une dimension d'énumération, il suffit alors de « *mettre des listes dans des listes* » :

➤ *Exemples* : le terme composé $ni(ni(1, ni(2), 3), 4)$ (de foncteur ni pour « nœud interne ») représente l'arbre « intuitif » suivant d'arité variable ne portant des valeurs que sur ses feuilles



Pour une gestion avec des listes, il se codera en Prolog $[[1, [2], 3], 4]$, soit graphiquement :



➤ *miniTP* :

a) En vous aidant si besoin d'un schéma, donnez le résultat de l'unification des listes suivantes :

1. $[X, Y, Z]$ et $[a, b, c]$

2. $[X, Y \mid Z]$ et $[a, b, c]$
3. $[X, Y \mid [Z]]$ et $[a, b, c]$
4. $[X \mid [Y]]$ et $[a, b, c]$
5. $[X \mid [Y, Z]]$ et $[a, b, c]$
6. $[X \mid Y]$ et $[[a, b], c]$
7. $[X, Y, Z]$ et $[[a, b], c]$
8. $[X, Y \mid Z]$ et $[[a, b], c]$
9. $[X, b, Z]$ et $[a, Y, c]$
10. $[X, Y, X]$ et $[a, b, c]$
11. $[[X, Y] \mid Z]$ et $[[a, b], [c, d], e]$

b) Écrire le prédicat *eq* avec deux arguments qui est vrai si le premier et le deuxième argument s'unifient. Vérifiez vos unifications de listes à l'aide de ce prédicat.

☞ Rappels cours précédent

(VIDEO-PPT 3.1)

- Une clause *Prolog* a une interprétation logique : c'est une clause de Horn.
- D'un point de vue logique, montrer que Q est vrai revient à montrer que la négation $\neg Q$ conduit à une contradiction en appliquant la règle de résolution.
- La *SLD-résolution* est l'algorithme utilisé par *Prolog* pour mettre en œuvre ce processus. Cet algorithme introduit des spécificités car il tient compte de l'ordre d'écriture des clauses et de l'ordre des littéraux dans les clauses (évaluation de gauche à droite). Par la suite, ces spécificités devront être prises en compte lors de l'écriture de programmes *Prolog*.
- Les listes non vides se définissent sur des arbres $[T \mid Q]$ constituées d'un élément de tête T et de Q qui est la liste privée du 1^{er} élément.
- Les arbres se mettent en œuvre avec des termes composés et foncteurs ou comme des listes de listes.

4.2. Récursivité et réversibilité

En Prolog la récursivité est une manière naturelle de manipuler des données et des programmes.

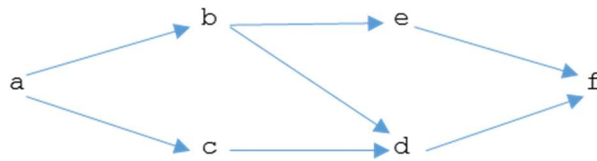
Éléments de méthode :

1. Caractériser les objets à manipuler / en relation.
2. Exprimer le problème sous forme relationnelle.
3. Analyse de cas :
 - cas général ;
 - cas particulier(s) (cas de base).
4. Pour chaque cas, description de la tête de clause.
5. Définition des clauses.
6. Est-ce que tous les cas possibles ont été traités ?

Chaque règle doit avoir un sens logique et donc être compréhensible indépendamment les unes des autres.

➤ *Exemple* : parcours d'un graphe

Trouver les chemins (listes des points intermédiaires) entre un point de départ et un point d'arrivée du graphe suivant :



En Prolog, la description de ce graphe peut être exprimée par la base de faits suivante :

```

arc(a, b).
arc(a, c).
arc(b, e).
arc(b, d).
arc(c, d).
arc(d, f).
arc(e, f).

```

1. Le problème met en relation 3 objets :
 - Un point de départ : D
 - Un point d'arrivée : A
 - Une liste de points intermédiaires : L
2. `chemin(D, A, L)` est vrai si L est la liste des points entre D et A
3. Analyse, 2 cas :
 - L est vide = [] : D et A sont en liaison directe
 - L est non vide = [X|Q] : L contient au moins un élément x, c'est le premier point sur le chemin de D à A.
4. A chaque cas correspond une tête de clause :
 - `chemin(D, A, [])`
 - `chemin(D, A, [X|Q])`
5. Définition des règles :
 - `chemin(D, A, [])` est vrai si `arc(D, A)`
 - `chemin(D, A, [X|Q])` est vrai si `arc(D, X)` et `chemin(X, A, Q)`

D → X-----Q-----A

D'où le programme :

```

chemin(D, A, []) :- arc(D, A).
chemin(D, A, [X|Q]) :- arc(D, X) , chemin(X, A, Q).

```

Des questions :

```

|?- chemin(a, d, L).
    L = [b] ?a
    L = [c]
|?- chemin(a, f, L).
    L = [b,e] ? a
    L = [b,d]

```



```

        L = [c,d]
|?- chemin(D, A, [d]).
    A = f , D = b ? a
    A = f , D = c

|?- chemin(D, A, [I]).
    A = f , D = b , I = b ? a
    A = d , D = a , I = b
    A = d , D = a , I = c
    A = f , D = b , I = d
    A = f , D = c , I = d

```

Ces exemples illustrent la notion de réversibilité des arguments dans une question Prolog. En général Prolog n'impose pas de statut aux paramètres d'un prédicat (la notion de paramètre entrée / sortie n'a pas de sens en logique). Cette caractéristique confère une puissance parfois insoupçonnée au programme *Prolog*.

Sur l'exemple, le prédicat `chemin` a été construit de manière implicite en pensant « si j'ai D et A, quel est le résultat L ». Cependant, du fait de la réversibilité, il peut être utilisé de multiples façons différentes.

Néanmoins, compte tenu du mode de recherche de Prolog (en profondeur d'abord et de gauche à droite), certains arguments peuvent avoir un sens E/S imposé. C'est le problème de l'arrêt de la récursivité qui guide ces choix.

➤ *Autre exemple* : le prédicat `membre`

Le prédicat `membre(X, L)` est vrai si X est un élément de la liste L.

- Si L est vide : échec quel que soit X
- Si L est non vide
 - Soit X est le premier élément de $L = [X|Q]$
 - Soit X est membre de Q avec $L = [Y|Q]$

Le problème est ici d'exprimer que `membre(X, [])` est faux. Il faut se rappeler que la base de connaissances n'exprime que des propositions vraies (hypothèse du monde fermé). Tout ce qui n'est pas dans la base de connaissances est donc considéré comme faux.

D'où le programme :

1. `membre(X, [X|Q]) .`
2. `membre(X, [Y|Q]) :- membre(X, Q) .`

Des questions :

```

|?- membre(1, [1,2,3]).
    true ?a
    yes
|?- membre(X, [1,2,3]).
    X = 1 ? a
    X = 2
    X = 3

```

Cette question est intéressante car elle permet d'énumérer les différents éléments d'une liste. C'est une fonctionnalité qui nous sera utile par la suite (cf programmation par essais successifs).

```
|?- membre(X, L) .
    L = [_ , X | _] ? ;
    L = [_ , _ , X | _] ? ;
    L = [_ , _ , _ , X | _] ? ;
    ...
```

Évidemment cette dernière question entraîne une infinité de réponses (et boucle), ce qui en soit est correct.

Synthèse sur la réversibilité

☞ Un prédicat peut admettre plusieurs possibilités d'appels (de combinaisons de paramètres libres et liés), ces possibilités découlent de la sémantique opérationnelle de Prolog, la résolution de l'appel (la question) doit se terminer.

☞ Il peut être intéressant d'indiquer explicitement les possibilités d'appel en commentaires, les conventions suivantes sont utilisées pour chaque argument a_i d'un prédicat P d'arité n ($i \in 1..n$) :

- $+a_i$ si l'argument doit être lié
- $-a_i$ si l'argument doit être libre
- $?a_i$ si l'instanciation est indifférente

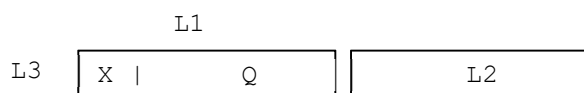
exemple : cf doc GProlog, /* member (?E, +L) */

➤ Mini-TP 1 : Concaténer deux listes

Le prédicat `conc(L1, L2, L3)` est vrai si $L3$ est la concaténation des listes $L1$ et $L2$. Par exemple, `conc([1,2], [3,4], [1,2,3,4])` est vrai.

Raisonnons en considérant le premier argument :

- Si $L1$ est vide : $L3 = L2$
- Si $L1$ est non vide, $L1$ est de la forme $[X|Q]$:



⇒ $L3$ est de la forme X suivi de la concaténation de Q et de $L2$

Ci-dessus aucune hypothèse n'a été faite sur $L2$ (qui peut être vide ou non).

➤ *Mini-TP 2 : Construction d'une sous-liste*

Définir le prédicat `impair(L1, L2)` tel que `L1` soit une liste contenant un nombre pair d'éléments (exemple : `[0, 1, 2, 3]`) et `L2` soit la liste des éléments de rang impair de `L1` (`[0, 2]`).

- Si `L1` est vide : `L2` est vide
- Si `L1` est non vide, `L1` est de la forme `[X, Y|Q]`, `L1` contient au moins 2 éléments
 - `X` est le premier élément de `L2` qui est donc de la forme `[X|Q2]`
 - Par ailleurs, les éléments de rang impair de `Q` correspondent à `Q2`

4.3. Récursivité enveloppée versus terminale

(VIDEO-PPT 3.2)

a) Récursivité enveloppée

➤ *Calcul du nombre d'éléments dans une liste*

La récursivité enveloppée est fondée sur un raisonnement par induction structurale : comment résoudre un problème sur une liste en supposant qu'on sait le résoudre sur une liste plus petite.

- Une liste vide a 0 élément.
- Une liste non vide L a la forme $[X|Q]$ où X est un élément et Q la liste L privée de son premier élément. La question qui se pose est : si on sait calculer le nombre d'éléments de Q , comme calculer le nombre d'éléments de $[X|Q]$?

D'où le programme en *Prolog* :

```
1. nbElements([], 0).  
2. nbElements([X|Q], R) :- nbElements(Q, RQ), R is RQ + 1.
```

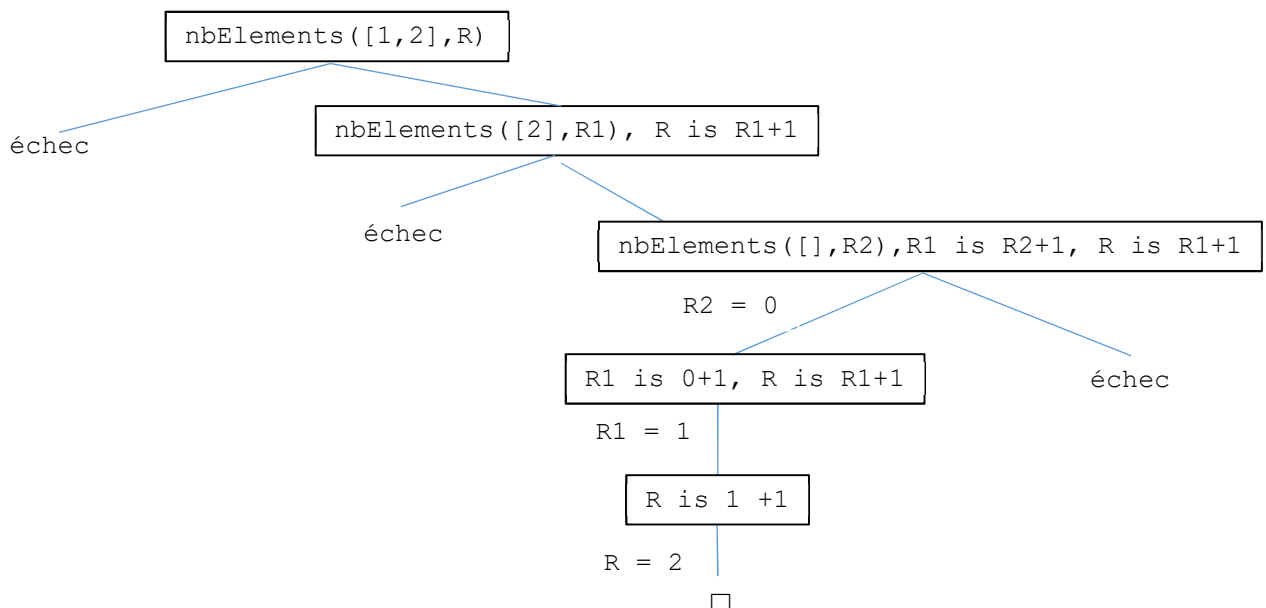
En *Prolog* le pseudo-prédicat $X \text{ is } \text{expr}$ unifie X à l'évaluation de l'expression expr .

☞ expr doit être évaluable, c.à.d que toutes les variables contenues doivent être liées. Ainsi la permutation ci-dessous dans la 2^{de} clause, pourtant logiquement équivalente, déclenchera un déroutement :

~~$\text{nbElements}([X|Q], R) :- R \text{ is } RQ + 1, \text{nbElements}(Q, RQ), .$~~

En revenant à la séquence correcte dans la queue de la règle 2, le résultat de l'appel récursif est utilisé pour ensuite construire le résultat R . Il est dit dans ce sens « enveloppé ». Il faut noter qu'une telle construction procède d'une démarche logique et que chaque règle garde un sens par elle-même, indépendamment des autres.

L'arbre de recherche de la question $\text{nbElement}([1,2], R)$ est :



Cet exemple montre que le nombre de buts de la résolvante augmente à chaque application de la règle 2, jusqu'à ce que la règle 1 arrête la récursivité. Les additions laissées en suspens s'effacent ensuite pour produire le résultat $R = 2$. Une observation attentive permet de constater que les éléments de la liste sont comptés de la droite (fin de liste) vers la gauche (début de liste). Comme cela sera vu par la suite cette caractéristique a des conséquences en termes de complexité algorithmique.

Un programme récursif enveloppé qui bouclerait aurait pour effet d'avoir une résolvante dont la taille augmenterait indéfiniment, ce qui produirait une saturation mémoire et donc un arrêt du moteur.

b) Récursivité terminale

Le problème du calcul du nombre d'éléments dans une liste correspond en fait à une construction itérative. En itératif, le comptage du nombre d'éléments dans une liste nécessite deux variables de travail : un compteur C et une variable P pour parcourir la liste.

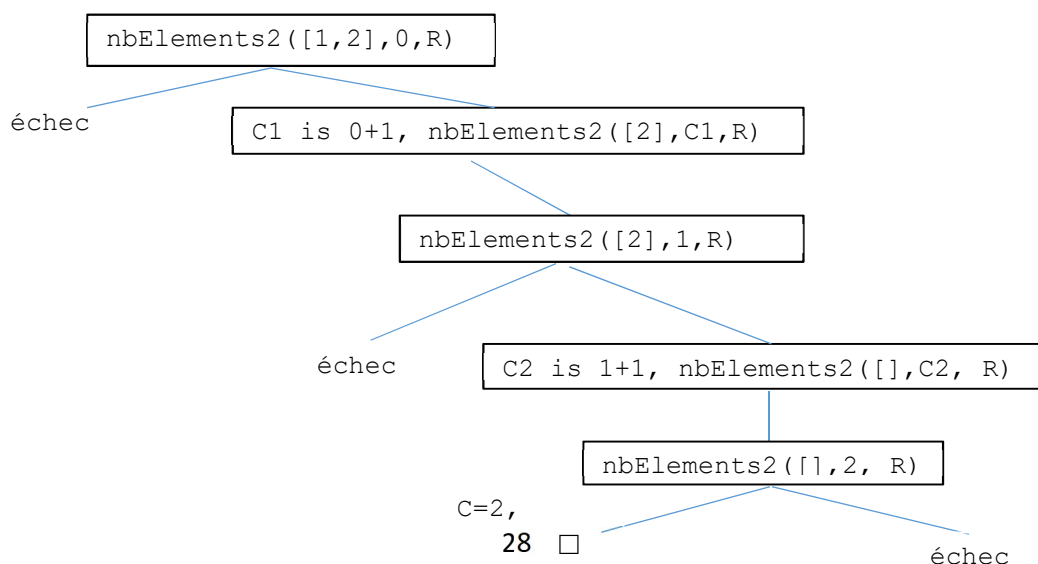
Un invariant de boucle donne le sens de C et P . À une étape du processus de comptage P désigne la liste des éléments restant à compter, C désigne le nombre d'éléments qui ont déjà été comptés. Au démarrage du calcul P désigne toute la liste et $C=0$, car rien n'a été compté. Bien entendu, quand toute la liste a été parcourue, P est vide et C contient alors le nombre d'éléments de la liste.

D'où :

1. `nbElements2([], C, C).`
2. `nbElements2([X|Q], C, R) :- C1 is C + 1, nbElements2(Q, C1, R).`

Une question a la forme : `nbElement([1, 2], 0, R)`. Les deux premiers arguments sont des paramètres d'entrée : la liste et la valeur initiale du compteur. Le dernier argument est le résultat final instancié uniquement lorsque toute la liste a été parcourue. L'appel récursif est à droite dans la règle 2, et s'exécute après l'addition. Dans ce sens, il est terminal. Il faut noter qu'une telle construction nécessite de comprendre le programme dans son ensemble, les règles ne peuvent plus être appréhendées les uns indépendamment des autres, contrairement à la version récursive enveloppée.

L'arbre de recherche de cette question est :



Durant toute la recherche la résolvante contient au plus 2 buts.

Une observation attentive permet de constater que les éléments de la liste sont comptés de la gauche (début de liste) vers la droite (fin de liste).

☞ *La récursivité terminale (ou récursivité à droite) est plus efficace que l'enveloppée. Elle peut en effet être écrite en une itération.*

Exemple hors-Prolog (qui ne permet pas l'itération) :

Pour une fonction récursive terminale dont :

- le 1^{er} argument maigrit pour permettre l'arrêt de la récursivité
- le 2nd argument accumule le résultat en cours de calcul

on a le schéma de programmation récursive terminale suivant (langage de description non typé, paramètres variables) :

```
fonction f(X, A)
  début
    si fin(X) alors
      R := A
    sinon
      X' := amaigrit(X)
      A' := augmente(A) avec (X)
      R := f(X', A')
    fsi
  retourne R
fin
```

qui se réécrit en itératif :

```
fonction f(X, A)
  début
    tant que NON fin(X) faire
      X' := amaigrit(X)
      A := augmente(A) avec (X)
      X := X'
    fait
  retourne A
fin
```

➤ *Mini-TP : factorielle*

1. En utilisant le schéma de récursivité terminale suivant :

```

fonction fact(X, A)
  début
    si X=0 alors
      R := A
    sinon
      X' := X-1
      A' := A*X
      R := fact(X', A')
    fsi
  retourne R
fin

```

qui requiert l'appel `fact(n, 1)` ^(a), écrivez cette fonction en Prolog

(a) le second paramètre correspondant à 0! peut être masqué via une fonction auxiliaire

2. Ecrivez son équivalent itératif en pseudo-code et déroulez le calcul des deux écritures pour 3!

4.4. Application à la manipulation de listes

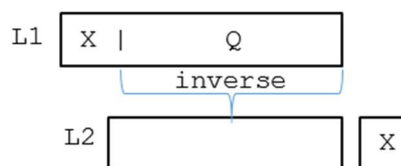
(VIDEO-PPT 3.3)

➤ *Inversion d'une liste*

Version enveloppée :

Le prédicat à deux arguments `inverse(L1, L2)` est vrai si `L2` est la liste inverse de `L1`.

- Si `L1` est vide : `L2` est vide
- Si `L1` est non vide, `L1` est de la forme `[X|Q]`



D'où le programme :

1. `inverse([], []).`
2. `inverse([X|Q], R) :- inverse(Q, QL), conc(QL, [X], R).`

Des questions :

```
| ?- inverse([0, 1, 2], L).
L = [2,1,0]
```

```
| ?- inverse(L, [2,1,0]).
```



Du fait du fonctionnement de l'interpréteur, cette dernière question génère une exécution qui boucle. La récursivité est prévue pour que le premier argument diminue à chaque appel pour atteindre [] et la sélection de la règle 1. Il faut donc que le premier argument soit instancié.

La complexité temporelle de ce programme a été étudiée en cours d'algorithmique. Pour une liste à n éléments, le prédicat `conc` est exécuté n fois, or celui-ci ajoute un élément en queue de liste, cet ajout étant linéaire, l'algorithme est en $O(n^2)$.

Version terminale :

Dans cette approche, la liste inverse de L est construite au fur et à mesure (processus itératif) du parcours de L . Il faut donc un argument auxiliaire pour propager la liste partielle progressivement construite. À chaque étape, le 1^{er} élément de la liste est inséré en tête dans la liste auxiliaire :

Liste	Liste auxiliaire
[1, 2]	[]
[2]	[1]
[]	[2, 1]

D'où le programme :

1. `inverse([], L, L).`
2. `inverse([X|Q], L, R) :- inverse(Q, [X|L], R).`

Lors du 1^{er} appel du prédicat `inverse`, le 1^{er} argument est unifié à la liste à inverser et le second argument (liste auxiliaire) doit être initialisé à [].

```
| ?- inverse([0,1,2], [], L).
```

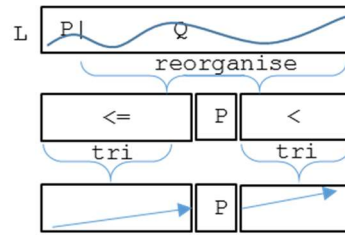
```
L = [2,1,0]
```

Cet algorithme est en complexité linéaire. La différence de complexité entre la version enveloppée et la version terminale est due au fait que la construction de la liste inverse à une liste L est effectuée dans le premier cas en traitant les éléments de L de la droite vers la gauche (ce qui induit des insertions en queue dans la liste inverse) alors que dans le second cas les éléments de L sont traités de la gauche vers la droite (ce qui entraîne des insertions en tête dans la liste auxiliaire).

➤ Tri rapide

Version enveloppée :

Le schéma suivant décrit le principe de construction du tri rapide par induction structurale :



D'où le programme :

```

1. triRapide([], []).
2. triRapide([P|Q], R) :- reorganise(Q, P, L1, L2),
    triRapide(L1, L1trie),
    triRapide(L2, L2trie),
    conc(L1trie, [P | L2trie], R).

```

Avec :

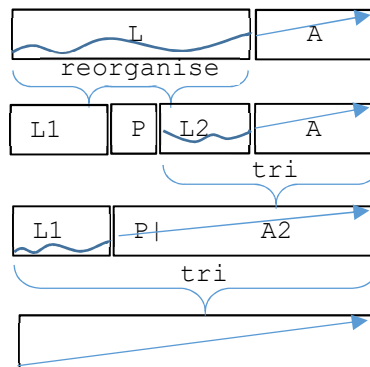
```

1. reorganise([], P, [], []).
2. reorganise([X|Q], P, [X|L1], L2) :- X <= P, reorganise(Q, P, L1, L2).
3. reorganise([X|Q], P, L1, [X|L2]) :- X > P, reorganise(Q, P, L1, L2).

```

Version terminale :

Cette construction est probablement plus difficile car moins intuitive. Il faut se placer dans une situation où une partie du travail a été réalisée. Elle est caractérisée par une liste auxiliaire triée *A* dont les éléments sont supérieurs ou égaux à ceux à la liste *L* des éléments restants non triés. Le principe du tri se schématise de la manière suivante :



D'où le programme :

```

1. triRapide([], L, L).
2. triRapide([P|Q], A, R) :- reorganise(Q, P, L1, L2),
    tri(L2, A, A2),
    tri(L1, [P|A2], R).

```

Question :

```

| ?- triRapide([2,0,4], [], R).

```

$R = [0, 2, 4]$

➤ *MiniTP : Fibonacci*

Définir les versions récursives enveloppée et terminale du prédicat `fib(N, R)` qui calcule dans R le résultat du terme F_n de la suite de Fibonacci définie par :

$$F_0 = F_1 = 1 \text{ et } F_n = F_{n-1} + F_{n-2} \text{ pour } n \geq 2$$

Tester les deux formes de récursivité aux rangs 5 et 25...

Note 1 : En passant par un prédicat auxiliaire :

- l'accumulateur peut être un couple décomposé sur ses deux atomes
- le cas d'arrêt à 0 est plus simple à tester qu'à n

Note 2 : Vous pourrez utiliser les pseudo-prédicats :

`var is expr` qui unifie la variable `var` à `expr` évaluée
`expr1 =\= expr2` qui réussit si `expr1` est différente de `expr2`

4.5. Méthode des d-listes (hors programme)

Cette méthode permet de passer d'une récursivité enveloppée à une récursivité terminale en supprimant le prédicat `conc`.

D-liste : une d-liste est une liste représentée comme une différence de 2 listes.

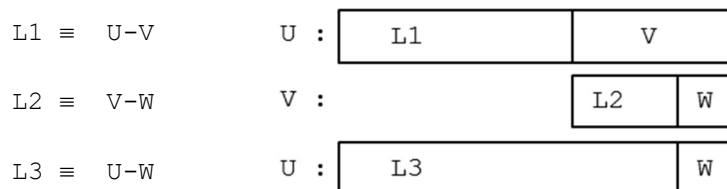
$$\text{➤ } [1, 2] \equiv [1, 2, 3] - [3]$$

$$[1, 2] \equiv [1, 2] - []$$

$$[1, 2 | Q] \equiv [1, 2, 3] - Q$$

Notation : une liste $[1, 2]$ exprimée comme une d-liste est notée : `dl([1, 2 | Q], Q)`

Le prédicat `conc(L1, L2, L3)` peut être écrit à l'aide de d-listes :



D'où le prédicat `concd1` :

```
1. concdl(dl(U, V), dl(V, W), dl(U, W)).
```

Question :

```
|?- concdl(dl([1,2|Q1], Q1), dl([3,4|Q2], Q2), dl([L3, []])
L3 = [1,2,3,4]
```

Ce qui est remarquable dans ce prédicat `concdl`, c'est que la concaténation se fait uniquement grâce à des unifications, elle s'exécute en temps constant.

L'inversion d'une liste en version récursive enveloppée peut être exprimée avec des d-listes et le `concdl` :

```
1. inversedl([], dl(W, W)).
2. inversedl([X|Q], dl(W1, W2)) :- inversedl(Q, dl(U1, U2)),
                                   concdl(dl(U1, U2), dl([X|V], V), dl(W1, W2)).
```

Lors de l'exécution de la règle 2, le `concdl` a pour effet d'unifier les termes suivants :

```
U1 = W1, U2 = [X|V], W2 = V
```

Il est possible de réécrire la règle 2 en tenant compte de ces correspondances pour minimiser le nombre de paramètres :

```
2. inversedl([X|Q], dl(W1, W2)) :- inversedl(Q, dl(W1, [X|W2])),
                                   concdl(dl(W1, [X|W2]), dl([X|W2], W2), dl(W1, W2)).
```

Désormais, lors de l'exécution de la règle 2, le `concdl` vérifie les unifications et est donc toujours vrai. Il est possible de l'omettre :

```
2. inversedl([X|Q], dl(W1, W2)) :- inversedl(Q, dl(W1, [X|W2])).
```

Le prédicat `inversedl` comporte 3 paramètres, la liste à inverser, `w1` et `w2`. Il peut se récrire avec 3 arguments :

```
1. inverse([], W, W).
2. inverse([X|Q], W1, W2) :- inverse(Q, W1, [X|W2]).
```

En comparant cette écriture avec la version récursive terminale du prédicat `inverse` décrite à la section précédente, constat est fait que ces deux écritures sont identiques à ceci près que dans cette dernière écriture la variable auxiliaire se trouve en 3^{me} argument et le résultat en 2^{me} argument.

6. Contrôle de la résolution

(VIDEO-PPT 3.4)

6.1. Possibilités existantes

(emprunts à [R. Janvier])

Prolog offre un niveau d'expression très puissant : description de la solution d'un problème sous forme de relations et de contraintes. L'inconvénient est que l'utilisateur ne possède pas le contrôle de la résolution comme dans un langage impératif (C, Java) par exemple. les seuls opérateurs de contrôle sont le choix : alternatives offertes par les clauses d'un même prédicat ; et le « et » séquentiel entre les buts de la partie droite d'une clause ou d'une question.

Il existe donc dans ce cadre différentes mises en œuvre de « choix » :

a) exprimer le choix dans la tête de clause

- *Exemple* : pour varier dans la représentation des données, en travaillant sur la représentation d'entiers par un arbre de successeurs d'un nombre avec le prédicat pair :

```
pair(0) .  
pair(succ(succ(N)) :- pair(N) .
```

b) exprimer le choix dans les contraintes de clause

- *Exemple* : en revenant aux listes

```
membre(X, [X|Q]) .  
membre(X, [Y|Q]) :- X\=Y, membre(X, Q) .
```

c) exprimer le choix dans les buts de la queue de clause

- *Exemple* : quand une liste encode un graphe

```
chemin(X, Y) :- arc(X, Y) .  
chemin(X, Y) :- arc(X, Z), chemin(Z, Y) .
```

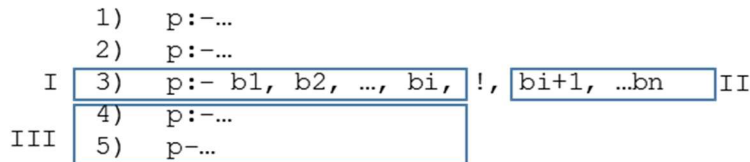
Ces techniques permettent donc d'obtenir l'exclusivité recherchée. Néanmoins, elles ne permettent pas de capturer toutes les situations où il devient inutile de continuer à parcourir l'arbre de recherche de la résolvante.

C'est pourquoi un outil de contrôle de la résolution (ou coupe-choix) a été introduit. Un tel outil n'a rien de logique, il s'utilise dans la vision opérationnelle de Prolog. S'il améliore l'efficacité de certains programmes, il engendre *a contrario* une perte d'expressivité des prédicats qui ne sont plus réversibles (les arguments ont alors un sens en entrée ou en sortie).

6.2. Coupe-choix (ou coupure)

Définition : Le pseudo-prédicat prédéfini **!** s'efface toujours (est toujours vrai) et demande à l'interpréteur de renoncer à tout choix laissé en suspens de la tête de clause courante incluse au **!** et ceux-là uniquement.

➤ Une vue du coupe-choix



Question : `|?- p.`

- Prolog essaie de résoudre `p` avec les règles 1 et 2
- Exécution de la règle 3 :
 - Tant que **!** n'est pas effacé
 - Exécution normale du bloc I (avec éventuellement des retours internes)
 - Un retour jusqu'à la tête de clause déclenche l'exécution du bloc III
 - Si **!** s'efface :
 - Tous les choix laissés en suspens sur I sont coupés, y compris ceux sur la tête de clause, (le bloc III n'est donc pas exécuté)
 - Le bloc II s'exécute normalement sans possibilité de retour avant **!**

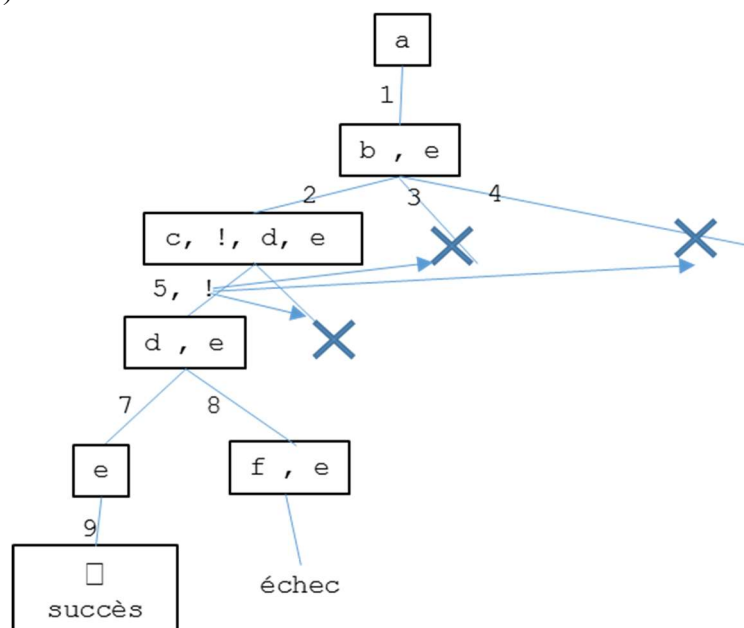
Le coupe-choix est un mécanisme d'anti-retour, une fois franchi le **!** il n'est plus possible de revenir en arrière (backtrack) avant celui-ci.

➤ Autre exemple (synthétique)

Base de connaissances :

1. `a :- b, e.`
2. `b :- c, !, d.`
3. `b :- c.`
4. `b :- e.`
5. `c.`
6. `c :- f.`
7. `d.`
8. `d :- f.`
9. `e`

Question : `|?- a.`



Il convient de remarquer que l'effacement du coupe-choix provoque l'élagage des choix pendants de même niveau (3 et 4) mais aussi de celui du premier sous-but rencontré (6) !

➤ Application : Tri par permutations

Le principe du tri par permutations consiste à générer toutes les listes possibles à partir des éléments d'une liste *L* donnée. Parmi ces listes, l'une est triée.

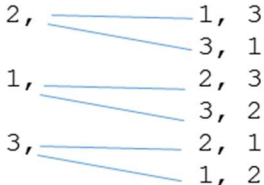
D'où le programme :

```
1. triPermut(L, Lpermut) :- permut(L, Lpermut) , croissant(Lpermut), ! .
```

Le coupe-choix indique à Prolog que dès qu'il a trouvé une solution, il peut s'arrêter (car elle est unique).

Les permutations d'une liste s'obtiennent en concaténant chaque élément *x* de la liste avec chaque permutation de la liste moins *x*.

Permutations de [2, 1, 3] :



Pour réaliser les permutations d'une liste *L*, il faut énumérer les éléments *x* de *L* et obtenir *L* privée de *x*. Le prédicat `enlever` réalise cette décomposition :

```
1. enlever(X, [X|Q], Q) .
2. enlever(X, [Y|Q], [Y|Q1]) :- enlever(X, Q, Q1) .
```

D'où :

```
1. permut([], []).
2. permut(L, [X|L2]) :- enlever(X, L, L1) , permut(L1, L2) .
```

Pour terminer, le prédicat `croissant` vérifie si une liste est croissante.

```
1. croissant([X]).
2. croissant([X, Y|Q]) :- X =< Y , croissant (Q) .
```

L'intérêt de ce tri n'est pas son efficacité mais sa construction algorithmique basée sur la méthode de programmation par essais successifs (ou générer-tester), qui sera abordée dans la suite du polycopié.

➤ Application : Négation par échec

La négation par l'échec découle de l'hypothèse du monde fermé, c'est-à-dire que ce qui n'est pas démontrable doit pouvoir être considéré faux. Le prédicat `non(p)` est donc vrai si `p` n'est pas démontrable. Ainsi avec la base de la famille Kennedy déjà introduite, le but `non(pere(john, X))` ne cherche pas tous les `X` dont `john` n'est pas le père, mais constate que `john` a un enfant et donc répond faux.

Soit `fail` le pseudo-prédicat qui est toujours faux (*remarque* : pour définir `fail` il suffit de **ne pas** le définir dans la base de connaissances). On peut alors écrire le méta-prédicat `non/1` de la façon suivante :

1. `non(P) :- P, !, fail.`
2. `non(P).`

Le méta-prédicat `non` prend en paramètre un prédicat `P` qui est placé en position d'évaluation en partie droite de la règle 1.



Attention : Si le terme utilisé en prédicat `P` contient des variables libres, les résultats peuvent être « surprenants ». Ainsi si l'on cherche à savoir qui est à la fois « un homme » et « non riche », en raison de la sémantique opérationnelle du coupe-choix, seule une séquence de buts fonctionnera, la commutativité de la conjonction sera perdue ! :

1. `homme(daniel).`
 2. `homme(cresus).`
 3. `riche(cresus).`
- ```
?- homme(X), non(riche(X)).
X = daniel

?- non(riche(X)), homme(X).
no
```



*Le prédicat `non/1` est clairement différent de la négation logique (puisque la conjonction logique ne tient pas). Il devra être utilisé en dernier recours car il empêche la réversibilité.*



*Pour cette raison Gprolog ne propose pas de prédicat prédéfini `not(P)` mais utilise la notation `\+(P)` avec `P` unifiable avec une tête de clause pour marquer que ce méta-prédicat n'est pas la négation dans tous les cas d'appel.*

### ➤ Application : Boucles d'interactions

L'interaction avec l'utilisateur est clairement un schéma d'exécution orienté dans lequel le coupe-choix sera précieux.

- Schéma d'une boucle mue par le succès :

```
boucle :- traitement, !, boucle.
boucle.
```

Tant que `traitement` réussit, la boucle est relancée.

Par exemple, avec le prédicat `traitement` défini ci-dessous, `boucle` lit un entier et affiche son carré jusqu'à la lecture d'un 0 (`traitement` échoue dans ce cas et provoque l'arrêt de la boucle).

```
traitement :- read_integer(X) , carre(X).
carre(0) :- ! , write(fin) , fail.
carre(X) :- X2 is X*X , write(X2), nl.
```

- Schéma d'une boucle mue par échec :

```
1. boucle :- traitement , !.
2. boucle :- boucle.
```

Tant que le `traitement` échoue, la boucle est relancée par la règle 2.

### MiniTP :

- 1) *Ecrire le prédicat `nonUnifiable (X, Y)` qui est vrai si `X` et `Y` ne sont pas unifiables.*
- 2) *Vous avez ci-dessous deux versions d'un prédicat `mini`, expliquer pourquoi le résultat de la question `mini(2,3,3)` est correct avec `min1` (répond no) et incorrect avec `min2` (répond yes)*

```
min1(X,Y,X) :- X=<Y .
min1(X,Y,Y) :- X>Y .

min2(X,Y,X) :- X=<Y, ! .
min2(X,Y,Y) .
```



## 7. L'algorithme d'unification (hors programme)

L'unification est un mécanisme puissant et fondamental de Prolog. Elle permet de réaliser : des déclenchements de règles, des instanciations, des passages de paramètres, l'accès à des structures de données, la construction de données structurées...

**Unifier** : Unifier deux termes, c'est chercher à les faire coïncider en imposant des conditions minimales sur les variables composant ces termes pour que cet appariement soit valide.

**Substituer** : Substituer une variable par un terme dans une expression, c'est y remplacer toutes les occurrences de cette variable par ce terme.

**Unificateur** : deux termes  $t_1$  et  $t_2$  sont unifiables s'il existe une substitution  $\sigma$  telle que  $\sigma(t_1) = \sigma(t_2)$ ,  $\sigma$  est un unificateur.

### ➤ Unificateurs

$ff(X, bb)$  et  $ff(aa, Y)$  sont unifiables par la substitution :  $\sigma_{\{X = aa\}} \circ \sigma_{\{Y = bb\}} = \sigma_{\{X = aa, Y = bb\}}$

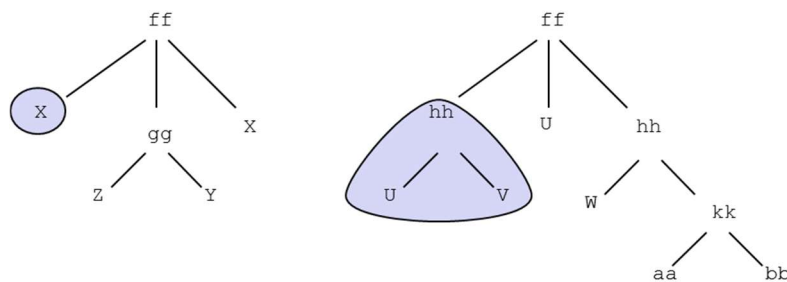
$ff(aa, X)$  et  $ff(aa, Y)$  a plusieurs unificateurs :  $\sigma_{\{X = aa, Y = aa\}}$  ou  $\sigma_{\{X = bb, Y = bb\}}$  ou ...

**Unification** : L'unification consiste à évaluer deux termes  $t_1$  et  $t_2$  par une substitution  $\sigma_g$  la plus générale possible.  $\sigma_g$  est un unificateur général.

### ➤ Unification

$ff(aa, X)$  et  $ff(aa, Y)$  sont unifiables par l'unificateur général :  $\sigma_{\{X = Z, Y = Z\}}$

### ➤ L'unification par l'exemple (algorithme de Robinson) :



Pour unifier ces 2 arbres  $t_1$  et  $t_2$ , on cherche le couple de sous-arbres les plus à gauche ne débutant pas un symbole identique. C'est un couple de désaccord, ici :  $D(t_1, t_2) = (X, hh(U, V))$ .

**si**  $D(t_1, t_2)$  comporte deux termes constants  
**alors**  $t_1$  et  $t_2$ , ne sont pas unifiables  
**sinon** on substitue la variable par la non variable dans  $t_1$  et  $t_2$ .

**et on itère ...**

$$\begin{aligned}
D(t_1, t_2) &= (X, hh(U, V)) \Rightarrow \sigma_1 = \sigma_{\{X = hh(U, V)\}} \\
\sigma_1(t_1) &= ff(hh(U, V), gg(Z, Y), hh(U, V)) \\
\sigma_1(t_2) &= ff(hh(U, V), U, hh(W, kk(aa, bb))) \\
D(\sigma_1(t_1), \sigma_1(t_2)) &= (gg(Z, Y), U) \Rightarrow \sigma_2 = \sigma_{\{U = gg(Z, Y)\}} \\
\sigma_2 \circ \sigma_1(t_1) &= ff(hh(gg(Z, Y), V), gg(Z, Y), hh(gg(Z, Y), V)) \\
\sigma_2 \circ \sigma_1(t_2) &= ff(hh(gg(Z, Y), V), gg(Z, Y), hh(W, kk(aa, bb))) \\
D(\sigma_2 \circ \sigma_1(t_1), \sigma_2 \circ \sigma_1(t_2)) &= (W, gg(Z, Y)) \Rightarrow \sigma_3 = \sigma_{\{W = gg(Z, Y)\}} \\
\sigma_3 \circ \sigma_2 \circ \sigma_1(t_1) &= \\
&ff(hh(gg(Z, Y), V), gg(Z, Y), hh(gg(Z, Y), V)) \\
\sigma_3 \circ \sigma_2 \circ \sigma_1(t_2) &= \\
&ff(hh(gg(Z, Y), V), gg(Z, Y), hh(gg(Z, Y), kk(aa, bb))) \\
D(\sigma_3 \circ \sigma_2 \circ \sigma_1(t_1), \sigma_3 \circ \sigma_2 \circ \sigma_1(t_2)) &= (V = kk(aa, bb)) \Rightarrow \sigma_4 = \sigma_{\{V = kk(aa, bb)\}} \\
\sigma_4 \circ \sigma_3 \circ \sigma_2 \circ \sigma_1(t_1) &= \\
&ff(hh(gg(Z, Y), kk(aa, bb)), gg(Z, Y), hh(gg(Z, Y), kk(aa, bb))) \\
\sigma_4 \circ \sigma_3 \circ \sigma_2 \circ \sigma_1(t_2) &= \\
&ff(hh(gg(Z, Y), kk(aa, bb)), gg(Z, Y), hh(gg(Z, Y), kk(aa, bb)))
\end{aligned}$$

Plus de désaccord : terminé  $\Rightarrow$  les deux arbres sont unifiés.

## 🔊 Rappels cours précédent

(VIDEO-PPT 4.1)

- Prolog a comme caractéristique innovante la réversibilité rendant possible plusieurs interrogations sur un même prédicat.
- Il existe deux formes de récursivité :
  - la récursivité enveloppée s'appuyant sur l'hypothèse de récurrence
  - la récursivité terminale calculant un résultant intermédiaire dans un accumulateur comme dans une boucle
- (hors programme, seulement ici pour la « connaissance » de l'existence)  
Lorsque les prédicats manipulent des listes, il existe une technique syntaxique permettant de passer d'une récursivité enveloppée (plus facile à mettre en place) à une récursivité terminale (plus efficace) : les d-listes.
- Le coupe-choix (anciennement coupure) permet d'éviter des résolutions inutiles mais doit être utilisé en dernier recours car il empêche la réversibilité.

## 8. Extensions au langage Prolog originel

### 8.1. Arithmétique

Gprolog définit des expressions arithmétiques faites de nombre et d'opérations : +, -, \*, / (division réelle), // (division entière), rem (reste), \*\* (puissance).

Par ailleurs les prédicats suivants permettent de manipuler des expressions arithmétiques :

```
E1 == E2 (teste l'égalité de l'évaluation de E1 et E2)
E1 \= E2 (teste la différence de l'évaluation de E1 et E2)
E1 < E2 (teste < sur l'évaluation de E1 et E2)
E1 <= E2 (teste <= sur l'évaluation de E1 et E2)
E1 > E2 (teste > sur l'évaluation de E1 et E2)
E1 >= E2 (teste >= sur l'évaluation de E1 et E2)
```

Attention, si E1 ou E2 comportent des variables, elles doivent être instanciées.

Le prédicat is (X is E) est utilisé pour instancier une variable X la valeur d'une expression E.

Les prédicats = et \= permettent de tester si deux termes sont unifiables.

#### ➤ Le prédicat factoriel

```
1. factoriel(0, 1).
2. factoriel(N, R) :- N >= 1, N1 is N-1, factoriel(N1, R1), R is
 N*R1.
```

### 8.2. Egalité et différence syntaxique

Il s'agit là non pas de tester les valeurs mais les variables atteintes avec == et \==. La comparaison directe de variable libre n'a guère d'intérêt mais devient intéressante quand les variables sont liées car c'est la liaison qui est testée.

#### ➤ Tester si deux paramètres sont indépendants (ne sont pas liés à une même variable, qui peut être encore libre) :

```
predicat(X, Y) :- X\==Y, ...
```

### 8.3. Pseudo-prédicats d'entrée-sortie

Gprolog en tant que langage de programmation propose une panoplie de prédicats prédéfinis permettant des réaliser des interactions avec l'utilisateur, notamment les prédicats read, read-atom, read-integer, write, ln... (se référer au manuel utilisateur).

### 8.4. Pseudo-prédicats d'introspection

Gprolog propose de nombreux prédicats prédéfinis parmi lesquels certains permettent de vérifier la nature des termes manipulés :

```

var(X) (teste si X est libre)
nonvar(X) (teste X est liée)
integer(X) (teste X est liée à un entier)
is_list(X) (teste si X est une liste)
arg(N, S, SN) teste si SN est le nième argument de S

```

## 8.5. Prédicats méta-logiques

Les prédicats méta-logiques sont des prédicats pour mettre à jour la base de connaissances. Ils permettent d'ajouter ou supprimer des clauses en tête ou en queue dans un paquet de clauses.

En Gprolog, les paquets de clauses modifiables de la base de connaissances doivent être préalablement déclarés `dynamic` dans la base de connaissances.

```
:- dynamic(pere/2).
```

Les prédicats de mises à jour de la base de connaissances sont notamment : `asserta` (insertion en tête de paquet de clauses), `assertz` (insertion en queue de paquet de clauses) et `retract` (suppression).


➤ *Exemple* (déjà vu) dans « `familleKennedy3.pl` » la méta-règle :

```

complete_pere_par_mere :- epoux(X,Y), mere(Y,Z),
 non(pere(X,Z)),
 assertz(pere(X,Z)).

```

Ces prédicats dynamiques sont intéressants pour la programmation de bases de connaissance pouvant agréger de nouveaux faits lors de l'exécution (systèmes experts).

 **Attention** : il faut manipuler ces prédicats avec précaution. En effet ils permettent à un programme de s'auto-modifier. Ainsi, deux exécutions successives suite à une même question peuvent éventuellement générer des résultats différents.

*MiniTP : De l'uchronie....*

En repartant de « `familleKennedy.pl` », après une première exécution de `grand_mere(rose,Y)` listant la descendance de `rose`, écrivez le prédicat `si_X_nEtaitPasNe(X)`, puis posez une seconde question :

```
si_X_nEtaitPasNe(john), grand_mere(rose,Y).
```

## 9. Programmation avancée


### 9.1. Programmation par essais successifs

(VIDEO-PPT 4.2)

Le tri par permutation abordé avec le coupe-choix est intéressant pour la méthode utilisée et sa structure. En effet, dans cette approche « *générer-tester* » on ne met pas en place un algorithme qui conduit directement à une solution. L'idée est de proposer *successivement des solutions* possibles puis, pour chaque solution proposée, de la tester afin de ne retenir que la (les) solution(s) effective(s). Dans le cas du tri, il n'y a qu'une seule solution.

En Prolog, c'est le mécanisme de retour arrière qui met en œuvre les essais et tests successifs :

```
problème(X) :- generer(X) , tester(X) .
```

 **Attention** : *L'ensemble des solutions possibles peut se révéler très grand (au point que cette technique peut être inexploitable), un des enjeux est de minimiser la génération des solutions possibles....*

➤ *Premier exemple : Coloriage d'une carte*

*Il s'agit de colorier les différentes zones d'une carte de telle sorte que deux zones ayant une frontière commune aient des couleurs différentes. Une telle carte peut se représenter par un graphe planaire. Il a été démontré qu'une solution n'utilisant que 4 couleurs est toujours possible.*

Soit la carte suivante :

|    |    |    |
|----|----|----|
| x1 | x2 | x3 |
| x4 | x5 | x6 |

Pour colorier cette carte avec 4 couleurs, il faut indiquer dans la base de connaissances quelles sont les couleurs disponibles :

```
couleur(bleu) .
couleur(blanc) .
couleur(rouge) .
couleur(vert) .
```

Puis essayer ces couleurs en respectant les conditions liées aux frontières des zones  $x_i$  :

```
colorier(X1,X2,X3,X4,X5,X6) :- couleur(X1), couleur(X2),
 couleur(X3), couleur(X4),
 couleur(X5), couleur(X6),
 X1\=X2, X1\=X4, X1\=X5, X2\=X3,
 X2\=X5, X2\=X6, X3\=X6, X4\=X5, X5\=X6.
```

Ce programme génère un grand nombre de solutions.

➤ *Deuxième exemple : Résolution d'une énigme*

*Trois amis se trouvent premier, second et troisième d'un concours de programmation. Ils ont tous une nationalité différente, un prénom différent et pratiquent un sport différent. Michael*

*aime le basket et a fait mieux que l'Américain. Simon, l'Israélien, a fait mieux que le tennisman. Le joueur de cricket est le premier. Qui est l'Australien ? Quel est le sport pratiqué par Richard ?*

Il faut identifier le nom, le sport et la nationalité de chacun. Les valeurs possibles sont respectivement, [michael, simon, richard], [tennis, cricket, basket], [israelien, australien, americain]. L'objectif est également de déterminer le classement au concours de programmation.

Pour modéliser ce problème, une liste  $L$  de triplets (*prénom, nationalité, sport*) est construite, l'ordre des triplets dans  $L$  correspond au résultat du concours.

La méthode utilisée est de type « générer-tester ». Différents triplets vont être générés au moyen du prédicat `permut` (cf « *triParPermutation.pl* »). Il faut ensuite vérifier que les conditions du problème sont satisfaites.

```

enigme(L) :- L = [[P1, N1, cricket], [P2, N2, S2], [P3, N3, S3]],
 permut([michael, simon, richard], [P1, P2, P3]),
 permut([israelien, australien, americain], [N1, N2, N3]),
 permut([tennis, basket], [S2, S3]),
 avant([michael, N4, basket], [P5, americain, S5], L),
 avant([simon, israelien, S6], [P7, N7, tennis], L).

avant(X, Y, [X, Y|_]).
avant(X, Y, [_, Z|Q]) :- avant(X, Y, [X|Q]).
avant(X, Y, [Z|Q]) :- avant(X, Y, Q).

?- enigme(L).
L=[[simon, israelien, cricket], [michael, australien, basket],
 [richard, americain, tennis]]

```

### ➤ Troisième exemple : Placer N reines sur un échiquier

*Le problème consiste à placer 4 reines sur un échiquier 4x4. Le résultat attendu est une liste de 4 couples indiquant la position [Ligne, Colonne] des 4 reines bien placées.*

Une solution naïve consiste à générer tous les quadruplets de couples possibles et, pour chacun d'eux, vérifier que les conditions « *ne pas être sur la même ligne, sur la même colonne et sur la même diagonale* » sont vérifiées.

Pour limiter les essais, comme une seule reine peut être sur une ligne et une colonne, dès lors qu'une ligne et une colonne ont été utilisées pour le placement d'une reine, celles-ci ne sont plus disponibles pour générer les positions des reines suivantes.

Une reine est placée en première colonne, puis une seconde reine est placée dans la seconde colonne, en vérifiant qu'elle ne se trouve sur une même diagonale que la reine déjà placée. Le processus est itéré pour placer les reines suivantes sur les colonnes suivantes sans qu'elles soient en prise avec les reines déjà positionnées.

Le prédicat sera de la forme :

```
reines(Ldisponibles, Cdisponibles, Rplacees, Resultat)
```

D'où le programme :

```
reines([], [], R, R).

reines(Ldisponibles, [C1 | Crestants], Rplacees, R) :-
 enlever(L1, Ldisponibles, Lrestants),
 compatible([L1, C1], Rplacees),
 reines(Lrestants, Crestants, [[L1, C1]|Rplacees], R).
```

La structure de ce programme est intéressante car elle met en œuvre à la fois un processus itératif (au moyen d'une récursivité terminale) pour placer chacune des reines et un processus d'essais successifs grâce au retour arrière sur le prédicat `enlever` utilisé ici pour générer des positions possibles sur les colonnes disponibles.

Le prédicat `compatible([L1, C1], Rplacees)` vérifie que la position `[L1, C1]` ne se trouve pas sur une diagonale avec les couples de positions `[L, C]` des reines placées de la liste `Rplacees`.

```
compatible([L1, C1], []).

compatible([L1, C1], [[L2, C2]|Q]) :- noDiag([L1, C1], [L2, C2]),
 compatible([L1, C1], Q).

noDiag([L1, C1], [L2, C2]) :- X1 is C1-C2, Y1 is L1-L2,
 abs(X1, X2), abs(Y1, Y2),
 X2 =\= Y2.

abs(X, Y) :- X < 0, Y is -1 * X.

abs(X, X) :- X >= 0.
```

## 9.2. Notions sur la programmation avec contraintes

(VIDEO-PPT 4.3)

Le but de cette section est d'aborder succinctement quelques notions de la programmation avec contraintes en extension de la programmation logique. Il s'agit d'un domaine en soi portant une méthodologie propre, les « *Constraint Satisfactory Problems* » (CSP). Cette section se bornera à montrer l'intérêt des contraintes au regard des problèmes traités dans la section précédente avec l'approche générer-tester.

**Définition :** Une *contrainte* est une relation entre des variables, chacune prenant ses valeurs dans un domaine donné. Elle restreint donc les valeurs possibles que les variables peuvent prendre.

Dans la résolution d'un problème tant que les contraintes sur les variables ne peuvent pas être calculées (parce que des variables ne sont pas instanciées) ou peuvent être calculées **et satisfaites**, la résolution du problème se poursuit. Par contre, dès qu'une contrainte est insatisfaite, la résolution échoue.

Les contraintes du résolveur de Gprolog portent sur des domaines finis d'entiers. Elles se caractérisent par le signe # : #=, #\=, #<, #=<, #>, #>=.

Le prédicat prédéfini `fd_domain(Lvars, Binf, Bsup)` permet de poser le système de contraintes tel que toutes les valeurs possibles des variables de `Lvars` soient comprises entre `Binf` et `Bsup`.

Le prédicat `fd_labeling(Lvars)` affecte (génère) des valeurs possibles à chaque variable de `Lvars` compte tenu de leur domaine.

➤ *Exemple 1 : Une équation*

```
| ?- fd_domain([X, Y], 1, 5), X + Y #= 8 , fd_labeling([X, Y]).
X = 3
Y = 5 ? a
X = 4
Y = 4
X = 5
Y = 3
```



*La programmation par essais successifs intégrant des contraintes amène à un schéma du type « contraindre-générer » plus efficace que la méthode « générer-tester ». En effet, les contraintes sur les variables du problème devant être satisfaites sont d'abord posées puis les valeurs possibles sur ces variables sont générées mais dès qu'une des contraintes est non satisfaite, le problème échoue. Dans l'approche générer-tester, les tests ne sont faits que lorsque toutes les variables ont été instanciées.*

➤ *Exemple 2 : Coloriage d'une carte*

*Colorier la carte proposée en exemple de la section précédente avec 4 couleurs et rester dans le domaine des entiers.*

En programmation par contraintes le problème s'écrit en codant chaque couleur par un entier entre 1 et 4:

```
1. coloriage(L) :- L = [X1, X2, X3, X4, X5], fd_domain(L, 1, 4) ,
 X1 #\= X2, X1 #\= X4, X1 #\= X5,
 X2 #\= X3, X2 #\= X5, X2 #\= X6,
 X3 #\= X6,
 X4 #\= X5,
 X5 #\= X6,
 fd_labeling(L) .
```

➤ *Exemple 3 : Cryptarithme*

Il s'agit de résoudre l'équation  $SEND + MORE = MONEY$  où chaque caractère est associé à un chiffre (entier entre 0 et 9) et où tous les caractères sont associés à des valeurs différentes.



```

1. cryptarithme(L) :- L = [S, E, N, D, M, O, R, Y], fd_domain(L, 0, 9),
 fd_all_different(L), contraintesEquations(L),
 fd_labeling(L).
2. contraintesEquations([S,E, N, D, M, O, R, Y]) :-
 Send #= 1000*S + 100*E + 10 *N + D ,
 More #= 1000*M + 100*O + 10*R + E,
 Money #= 10000*M + 1000*O + 100*N + 10*E + Y,
 Money #= Send + More.

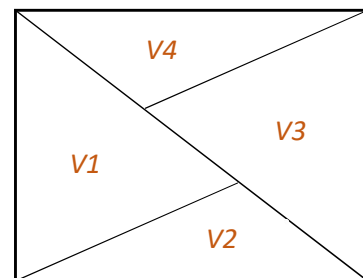
```

Le prédicat prédéfini `fd_all_différent(Lvars)` contraint toutes les variables de la liste `Lvars` à prendre des valeurs différentes.

*MiniTP : un coloriage a 3 couleurs !*

*Soit la carte suivante décrivant les frontières entre quatre villes (V1, ..., V4). On voudrait colorier la carte en utilisant seulement les couleurs rouge, bleu et vert, de sorte que :*

- *V1 soit en rouge ou en vert*
- *V2 et V3 soient en bleu ou en vert*
- *V4 soit en vert.*



### 9.3. Prédicats du second ordre

(VIDEO-PPT 4.4)

En *Prolog* les arguments d'un prédicat sont considérés comme des données (atomes ou termes composés), ils ne sont pas évalués. En programmation du second ordre les arguments d'un prédicat peuvent correspondre à des prédicats, nous avons rencontré un tel méta-prédicat avec `non(P)` (à programmer), gProlog proposant `/(P)`.

Un autre méta-prédicat (prédéfini) très intéressant est `findall(V, But, L)` qui réussit si `L` est la liste des solutions `v` du problème `But`.

➤ *Exemple 1* : Trouver les enfants de john

```

| ?- findall(X, pere(john, X), L).
L = [john_john, caroline]

```

La question `pere(john X)` fournit toutes les solutions `x` séparément. Le prédicat `findall` permet de collecter toutes les solutions d'un problème passé en paramètre.

➤ *Exemple 2* : Parcours en largeur d'un graphe

Le parcours en largeur d'un graphe se fait de manière itérative à l'aide d'une file de nœuds à explorer (cf cours d'algorithmique et de graphes). La boucle gérant un tel parcours retire un

nœud à explorer de la file et ajoute dans la file les descendants de ce nœud et cela tant qu'il y a des nœuds à explorer. Afin d'éviter de tomber dans un cycle, seuls les nœuds qui n'ont pas déjà été observés (déjà explorés ou en attente d'être explorés) sont introduits dans la file.

En *Prolog* il faut décrire le graphe. Cela peut être réalisé en définissant la base de fait des arcs (*exemple vu en TP*) :

```
arc(a,b).
arc(a,c).
arc(b,e).
arc(b,d).
arc(c,d).
arc(e,f).
arc(d,f).

parcoursLargeur([], Explores, R):- inverse(Explores, R).

parcoursLargeur([X|Q], Explores, R):-
 findall(Y, filsNonObservees(X, Y, Explores, Q), Fils) ,
 conc(Q, Fils, L),
 parcoursLargeur(L, [X|Explores], R).

filsNonObservees(X, Y, Explores, Q):- arc(X, Y) ,
 \+(membre(Y, Explores)),
 \+(membre(Y, Q)).
```

Nous sommes ici à nouveau dans un cas d'itération traitée en récursivité terminale avec donc en question `filsNonObservees([a], [], R)`.

*MiniTP : Une famille qui aura beaucoup donné à Prolog....*

En repartant de « *familleKennedy.pl* », collectez avec un seul succès la descendance de rose.

## ☞ Rappels cours précédent

- *Prolog* propose des extensions avec les notions de contraintes, de pseudo-prédicat d'entrée-sortie, de méta-prédicat d'introspection et de modification de la base de connaissances
- Les essais successifs dans le modèle *générer-tester* sont un modèle de programmation qui permet par ailleurs de s'appuyer sur un historique avec la récursivité
- L'usage de contraintes permet de gagner en efficacité de l'approche *générer-tester* qui devient alors *contraindre-tester*
- Avec `findall`, il est possible d'avoir un méta-prédicat réunissant tous les succès en un seul

## 10. Applications (hors programme)

### 10.1. Un analyseur syntaxique

Prolog a pour origine des travaux sur la reconnaissance du langage naturel, il est donc bien adapté pour réaliser un analyseur syntaxique basé sur les spécifications grammaticales d'un langage décrites sous forme de règles syntaxiques.

L'objectif est ici de définir une version très simplifiée d'un analyseur du français pour reconnaître la structure d'une phrase élémentaire comme « le chat mange la souris ».

La structure d'une telle phrase se définit par la grammaire (forme BNF) suivante :

```
<phrase> ::= <groupeNominal> <groupeVerbal>
<groupeNominal> ::= <determinant> <nom>
<groupeVerbal> ::= <verbe> <groupeNominal>
<groupe_verbal> ::= <verbe>
<verbe> ::= mange
<verbe> := ronronne
<determinant> ::= le
<determinant> := la
<nom> ::= chat
<nom> ::= souris
```

En Prolog, une phrase en entrée de l'analyseur prend la forme d'une liste de mots comme [le, chat, mange, la, souris]. L'objectif est de déterminer si la structure d'une telle phrase est correcte en posant la question : `phrase([le, chat, mange, la, souris])`. Il s'agit donc d'identifier les différents composants grammaticaux de la phrase. Pour ce faire, dans une version naïve, une méthode de type générer tester est utilisée :

```
1. phrase(X):- conc(Y, Z, X) , groupeNominal(Y) , groupe_verbal(Z) .
2. groupeNominal(X) :- conc(Y, Z, X) , determinant(Y) , nom(Z) .
3. groupeVerbal(X) :- conc(Y, Z, X) , verbe(Y) , groupeNominal(Z) .
4. groupe_verbal(X) :- verbe(X) .
5. verbe([mange]) .
6. verbe([ronronne]) .
7. determinant([le]) .
8. determinant([la]) .
9. nom([chat]) .
10. nom([souris]) .
```

Le prédicat `conc` est utilisé comme générateur de toutes les sous-listes `Y` et `Z` de `X` telles que `conc(Y, Z, X)`.

Une version plus générale est obtenue en appliquant la méthode des d-listes présentée en IV.4.

La règle 1 se transforme en posant `Y = dl(S0,S1)`, `Z = dl(S1,S)` et `X = dl(S0,S)` :

```
phrase(dl(S,S)):- concdl(dl(S0,S1), dl(S1,S), dl(S0,S)),
 groupeNominal(dl(S0,S1)) , groupe_verbal(dl(S1,S)) .
```

Le prédicat `concd1` n'a pas d'effet, sauf à vérifier que les unifications ont été correctement établies, il peut être supprimé. En appliquant cette approche sur l'ensemble des règles, il en découle le programme suivant :

```

1. phrase(S0, S):- groupeNominal(S0, S1) , groupe_verbal(S1, S).
2. groupeNominal(S0, S) :- determinant(S0, S1) , nom(S1, S).
3. groupeVerbal(S0, S) :- verbe(S0, S1), groupeNominal(S1, S).
4. groupe_verbal(S0, S) :- verbe(S0, S).
5. verbe([mange|S], S).
6. verbe([ronronne|S], S).
7. determinant([le|S], S).
8. determinant([la|S], S).
9. nom([chat|S], S).
10. nom([souris|S], S).

```

L'analyse d'une phrase est déclenchée par une question comme :

```
| ?- phrase([le, chat, mange, la, souris], []).
```

La règle 1 s'interprète de la manière suivante : si de la liste de mots `s0` on retire le groupe nominal et qu'il reste `s1`, que de `s1` on retire le groupe verbal et qu'il reste `s`, alors lorsque qu'on extrait une phrase de `s0`, il reste `s`. Une telle formulation est généralisable à des séquences grammaticales ayant plus de 2 composants.

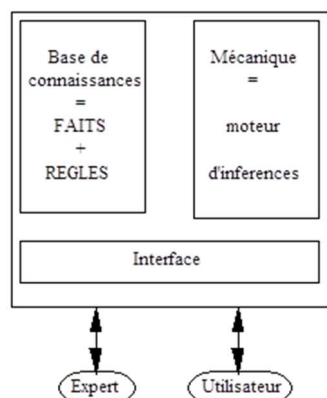
Cette grammaire vérifie qu'une phrase est structurellement correcte mais elle ne tient pas compte de traits grammaticaux (genre : la chat ; nombre : les chat) et de traits sémantiques (sens : la souris miaule). Ces éléments peuvent être traités en associant à chaque composant des paramètres (traits) caractérisant le genre, le nombre, le sens ... et en associant dans la grammaire des composants dont les traits sont compatibles.

## 10.2. Un système expert

Un système expert vise à offrir à ses utilisateurs les deux fonctionnalités suivantes :

1. modéliser un sous-ensemble du monde réel ;
2. obtenir des informations sur ce sous-ensemble.

Un système expert à la structure suivante :



La base de connaissances est constituée de faits et de règles. Les faits permettent de rendre compte d'informations du type "Tim est un chat". Les règles sont du type :

si <conditions> alors <conséquences>.

Conditions et conséquences sont des relations sur des variables ou des constantes symboliques. Les listes de conditions et de conséquences représentent des conjonctions.

Le monde est décrit à l'aide des règles et des faits initiaux suivants :

1. tim ronronne
2. tim est un animal
3. tim a des griffes
4. si x ronronne alors x est un chat
5. si x ronronne et x animal alors x est domestique et x est dorloté
6. si x est un chat alors x est un mammifère et x a des griffes

Ce monde est décrit dans la base de connaissances Prolog comme suit :

1. `dynamic(connu/1).`
2. `depart(ronronne(tim)).`
3. `depart(animal(tim)).`
4. `depart(griffe(tim)).`
5. `regle(1, [ronronne(X)], [chat(X)]).`
6. `regle(2, [ronronne(X), animal(X)], [domestique(X), dorlote(X)]).`
7. `regle(3, [chat(X)], [mammifere(X), aGriffes(X)]).`

Les faits initiaux sont représentés à l'aide du prédicat `depart(Finit)`, les règles à l'aide du prédicat `regle(N, Conditions, Conclusions)` où `Conditions` et `Conclusions` sont respectivement une liste des conditions et une liste des conséquences. Les faits démontrés sont mémorisés dans la base de connaissances à l'aide du prédicat `connu(Fdemontre)`.

Le moteur d'inférences peut fonctionner de deux manières différentes :

1. en chaînage avant : il établit de nouveaux faits par déduction à partir des faits connus.
2. en chaînage arrière : il cherche à démontrer une hypothèse (un but).

Le démonstrateur en **chaînage avant** sature le monde, c'est-à-dire qu'il calcule tous les faits dérivables à partir de l'état initial et l'application des règles. Il s'écrit :

1. `demo :- init , deduire.`
2. `init :- retractall(connu(F)) , depart(F) , assertz(connu(F)) , fail.`
3. `init.`
4. `deduire :- regle(N, Conditions, Conclusions) ,  
                  tester(Conditions), integrer(Conclusions), ! , deduire.`
5. `deduire.`
6. `tester([]).`
7. `tester([X|Q]) :- connu(X) , tester(Q).`

```

8. integrer(Conclusions):- integrer(Conclusions, false).
9. integrer([], true).
10. integrer([C|Cs], B):- connu(C), ! , integrer(Cs, B).
11. integrer([C|Cs], B):- assertz(connu(C)) , integrer(Cs, true).

```

où

- regle sélectionne une règle ;
- tester unifie, si c'est possible, les conditions sur des faits déjà connus ;
- integrer agrandit le monde connu avec les nouvelles conclusions (il répond faux si tous les conclusions sont déjà connues).

Le démonstrateur en **chainage arrière** part d'une liste de buts à démontrer (hypothèses) et « remonte » les règles pour confirmer ou infirmer ces buts. Il s'écrit :

```

1. resout([]).
2. resout([B | Buts]) :- connu(B), ! , resout(Buts).
3. resout([B | Buts]) :- regle(N, Conditions, Conclusions),
 membre(B, Conclusions), conc(Conditions, Buts, Resolvante) ,
 resout(resolvante).

```