# INF242V: Cours de Prolog

Romain Janvier
remerciements à Xavier Girod et Philippe Morat

18 avril 2007

## Table des matières

**I Introduction à Prolog** **4**

**1 Introduction** **4**
1.1 Présentation . . . . . 4
1.1.1 Évolution des langages et historique . . . . . 4
1.1.2 Les atouts de Prolog . . . . . 4
1.2 Approche intuitive de Prolog . . . . . 5
1.2.1 PROgrammer en LOGique . . . . . 5
1.2.2 Un peu de vocabulaire . . . . . 5
1.2.3 Faits-Questions-Règles . . . . . 5

**2 L'univers de gprolog** **8**
2.1 Les termes . . . . . 8
2.1.1 Les atomes . . . . . 8
2.1.2 Les variables . . . . . 8
2.1.3 Les nombres . . . . . 9
2.1.4 Les termes composés . . . . . 9
2.1.5 Représentation des termes . . . . . 9

**3 Arithmétique en gprolog** **9**
3.1 Les expressions arithmétiques . . . . . 9
3.2 Prédicats arithmétiques . . . . . 9
3.3 Le prédicat `is/2` . . . . . 10

**4 Définition d'un programme Prolog** **10**
4.1 Exemples de programmes Prolog . . . . . 10
4.1.1 Le restaurant (suite) . . . . . 10
4.1.2 Élévation à la puissance . . . . . 10

**5 Le fonctionnement de Prolog** **11**
5.1 L'unification . . . . . 11
5.2 Sémantique déclarative . . . . . 11
5.3 Sémantique procédurale . . . . . 12
5.3.1 Principe . . . . . 12
5.3.2 Algorithme de la résolution (simplifié) . . . . . 12
5.3.3 Exemple de résolution . . . . . 13
5.4 Exemples de programmes . . . . . 13
5.4.1 La factorielle . . . . . 13
5.4.2 Concaténation de listes . . . . . 15

6 Solution des exercices 16
6.1 Le restaurant . . . 16
6.2 La multiplication . . . 16

II Concepts algorithmiques 17

7 Spécification de prédicats 17
7.1 Définition . . . 17
7.2 Exemples . . . 17

8 Récursivité et traitement des listes 17
8.1 Introduction . . . 17
8.1.1 Définition récursive d'un prédicat . . . 17
8.1.2 Exemples . . . 18
8.2 Définition récurrente d'un type (ou type inductif) . . . 18
8.2.1 Définition récurrente . . . 18
8.2.2 Définition récurrente des listes d'éléments . . . 18
8.2.3 Définition récurrente des entiers naturels . . . 19
8.3 Analyse récurrente d'un prédicat . . . 19
8.3.1 Exemple sur les listes . . . 19
8.3.2 Exemple sur les entiers . . . 20
8.3.3 Évaluation d'un nombre en base 2 . . . 20

9 Structures arborescentes 21
9.1 Définition et terminologie . . . 21
9.1.1 Arbres $n$-aires, forêts . . . 21
9.1.2 Convention de représentation . . . 21
9.1.3 Arbres binaires . . . 22
9.1.4 Terminologie et définition (p96 du bouquin PCS) . . . 22
9.2 Arbres en Prolog . . . 22
9.2.1 Arbre $n$-aire, forêts . . . 22
9.2.2 Arbre binaires : les trois modèles . . . 22
9.2.3 Traitement récursifs des arbres $n$-aires . . . 24
9.3 Exemples . . . 24
9.3.1 Arbre binaires . . . 24
9.3.2 Arbre $n$-aires . . . 25
9.3.3 Arbre de recherche binaire (ARB) . . . 25

10 Contraintes sur les domaines finis 26
10.1 Introduction . . . 26
10.2 Classification des contraintes . . . 27
10.2.1 Définition des domaines . . . 27
10.2.2 Contraintes arithmétiques . . . 27
10.2.3 Information sur les domaines . . . 28
10.2.4 Contraintes d'énumération . . . 29
10.3 Exemples d'utilisations . . . 29
10.4 Générateur d'entiers . . . 29
10.4.1 Résolution d'équations linéaires . . . 29
10.4.2 La factorielle (le retour) . . . 30
10.4.3 SEND+MORE=MONEY . . . 30
10.4.4 Le carré magique . . . 30

III Aspects techniques 33

11 Un outil de contrôle : le coupe-choix 33
11.1 Problématique . . . 33
11.2 Définition du coupe-choix . . . 34
11.3 Utilisations . . . 35
11.3.1 Contrôle des solutions dans un processus de génération . . . 35
11.3.2 Confirmation du choix d'une règle . . . 36
11.3.3 Expression de nouvelles structures de contrôle . . . 37

12 Quelques prédicats prédéfinis de gprolog 38
12.1 Entrées-sorties . . . 38
12.2 Génération de nombres aléatoires . . . 39
12.2.1 Initialisation . . . 39
12.2.2 Tirage . . . 39
12.3 Trouver toutes les solutions . . . 39
12.3.1 `findall/3` . . . 39
12.3.2 `bagof/3` et `setof/3` . . . 40

IV Applications à la programmation logique 41

13 Introduction 41

14 Problèmes de génération-test (generate & test) 41
14.1 Principe d'un programme de génération et test . . . 41
14.2 Exemples . . . 41
14.2.1 Nombres entiers multiples de 3 entre 0 et 100 . . . 41
14.2.2 Voyelles d'un mot . . . 42
14.2.3 LES MUTANTS . . . 42
14.2.4 Tri par permutation . . . 43
14.2.5 Les huit Reines . . . 43
14.2.6 Le labyrinthe . . . 44
14.2.7 Conversion nombre Romain/Nombre arabe . . . 44
14.2.8 Le fermier, la poule , le renard et le grain . . . 45

15 Solution des exercices 45
15.1 Les huit Reines . . . 45
15.2 Le labyrinthe . . . 46
15.3 Conversion nombre romain/nombre arabe . . . 47
15.4 Le fermier, la poule, le renard et le grain . . . 47

Première partie

# Introduction à Prolog

## 1 Introduction

### 1.1 Présentation

Référence Prolog :
- Prolog : F. Giannesini, H. Kanoui, R. Pasero, M. Van Caneghem. InterEditions 1985
- Programmer en Prolog : W.F. Clocksin, C.S. Mellish. Editions Eyrolles 1985
- Prolog : Fondements et Applications : CONDILLAC. Editions Masson.
- L'ART DE Prolog : L. Sterling & E. Shapiro. Editions Masson 1990.
- Prolog III : Manuel de Référence et d'Utilisation. PrologIA, 1990.
- GProlog : http ://pauillac.inria.fr/ diaz/gnu-prolog/
- GProlog manual : http ://pauillac.inria.fr/ diaz/gnu-prolog/manual/index.html

#### 1.1.1 Évolution des langages et historique

| n° | Langage | paradigme | autres langages | style |
|---|---|---|---|---|
| 1 | PASCAL,C | impératif ou actionnel | Fortran, ADA,JAVA | "Fais ça" |
| 2 | LISP,Scheme | applicatif ou fonctionnel | Caml, ML | "Évalue ça" |
| 3 | Prolog | déclaratif ou relationnel | C-prolog, Prolog3, SQL | "que penses tu de ça ?" |

- Degré d'expressivité : n°
- Degré de contrôle (exprimé par le programmeur) : 1/n°
- Nombre de concepts de base utilisé (implique une grammaire du langage très simple) : 1/n°

Attention : l'expression dense est souvent peu accessible (5 page de Pascal, 1/2 page Lisp, quelques lignes de Prolog)

| n° | Paradigme | le programme est | son exécution consiste à |
|---|---|---|---|
| 1 | impératif | un ensemble d'actions séquentielles | déclencher les actions et modifier l'état de variables |
| 2 | applicatif | une fonction (composition) | évaluation de la fonction avec des paramètres effectifs |
| 3 | déclaratif | un ensemble de règles et de faits | lancer une résolution en tenant compte des règles et de l'état de la base de faits |

Prolog est né dans les années 70. Il est issu des travaux de Alain Colmerauer à l'Université de Marseille LUMIGNY pour les aspect analyse et conception du langage ; et des travaux sur la logique théorique de Warren à Edimburgh.

Le 1er système Prolog interprété a vu le jour à Marseille en 1973, sous la forme d'un démonstrateur de théorèmes.

À l'heure actuelle :
Plusieurs dialectes et interprètes de Prolog existent. Deux écoles se retrouvent ici :
- la syntaxe d'Edimburgh et le langage associé : C-Prolog (Delphia Prolog, quintus, gprolog, SWI-Prolog)
- la syntaxe de Marseille : Prolog II , Prolog III, (Prolog IV)

Le système Prolog a lui même évolué : prise en compte de retardement (freeze) en Prolog II, et introduction des contrainte (numérique, booléenne et sur les arbres) en Prolog III et même Prolog IV (contraintes sur les réels , contraintes non linéaires ).

#### 1.1.2 Les atouts de Prolog

- aspect déclaratif : la formulation est plus proche des spécifications. De plus les connaissances, donnée par l'utilisateur, sont bien séparées du contrôle, réalisé par l'interprète. En Prolog III, la mise en place des contraintes (conditions devant être satisfaites par les composants du programme) permet de se dégager un peu plus de l'aspect contrôle.
- système complet : le système Prolog se présente sous la forme d'un environnement proposant un interprète, un debuggeur et une batterie de prédicats prédéfinis. L'utilisateur lance l'environnement et déroule une "session" Prolog (comme pour CAML).

– Puissance : le système est très puissant du fait des structures de contrôle (Moteur d'inférence) de Prolog. Dans Prolog III, l'interprète sait résoudre des systèmes d'équation linéaire à variables rationnelles et des systèmes booléens.

Plus qu'un langage, on peut considérer Prolog comme un système de démonstration et de résolution de systèmes de contraintes.

## 1.2 Approche intuitive de Prolog

### 1.2.1 PROgrammer en LOGique

Avec Prolog on utilise la logique (système formel) comme langage de programmation. Par analogie, le système formel à la base de SCHEME est le $\lambda$-calcul.
Dans le système formel de la logique on distingue des objets : *Jean*, *la logique*; des propriétés sur les objets : *heureux*; des relations : *aime*.

Exemple : "qui aime la logique est heureux "
"Jean aime la logique" $\Rightarrow$ "Jean est heureux"

### 1.2.2 Un peu de vocabulaire

**Définition 1** *Un* **prédicat** *est une propriété ou une relation.*

**Définition 2** *Un* **argument** *(de prédicat) est un objet sur lequel porte un prédicat*

Exemple : `heureux(jean). aime(jean,la_logique).`

**Définition 3** *Une* **clause** *est une phrase du système exprimant que des conditions entraînent des conclusions.*

Exemple : "si jean aime la logique, alors il est heureux ou fatigué."

**Définition 4** *Une* **clause de Horn** *est une clause constituée au plus d'une conclusion.*

Ce sont les clauses manipulées par Prolog.

**Définition 5** *La partie conclusion est appelé* **tête** *de clause, l'ensemble de conditions est appelé* **queue** *de clause.*

| Exemple : | TETE | | QUEUE |
|---|---|---|---|
| | "Jean est heureux | | s'il aime la logique" |
| Prolog III | `heureux(jean)` | `->` | `aime(jean,la_logique);` |
| C-Prolog | `heureux(jean)` | `:-` | `aime(jean,la_logique).` |

Remarque : on peut lire la clause de droite à gauche : "Si Jean aime la logique alors il est heureux"
**Programmer en Prolog (version 1) :** c'est décrire à l'aide de clause de Horn, les connaissances nécessaires pour résoudre un problème. L'interpréteur se charge ensuite de trouver les solutions.

### 1.2.3 Faits-Questions-Règles

**Faits** La proposition : " l'avocat peut être consommé en entrée" énonce une **propriété** sur **l'objet**.

**Définition 6** *Un* **fait** *est une relation sur des objets considérée comme vraie.*

Il se note :
```
entree(avocat).
calories(avocat,200).
```

**Définition 7** *Une* **base** *de faits est un ensemble de faits décrivant une connaissance donnée.*

Exemple :

```
entree(salade).        dessert(raisin).
entree(avocat).        dessert(melon).
entree(huitre).

viande(steak).         poisson(truite).
viande(escalope).      poisson(daurade).

calories(salade,15).   calories(raisin,70).
calories(avocat,220).  calories(melon,27).
calories(huitre,70).   calories(steak,203).
calories(truite,98).   calories(escalope,105).
calories(daurade,90).

blanc(sauterne).       rouge(lichine).
blanc(chablis).        rouge(vougeot).
```

Prolog considère comme **vrai** tout fait (prédicat) cité dans la base de faits du programme, et considère comme **faux** tout ce qui ne peut pas en être déduit.
Corollairement, si le programme ne comporte que des faits, tout fait non présent dans la base est considéré comme faux.

**Questions et variables** Une base de fait constitue un programme Prolog, on peut en demander une exécution en posant une Question.

**Définition 8** *Une* **Question** *est un prédicat Prolog dont on cherche à savoir s'il est vérifié ou non.*

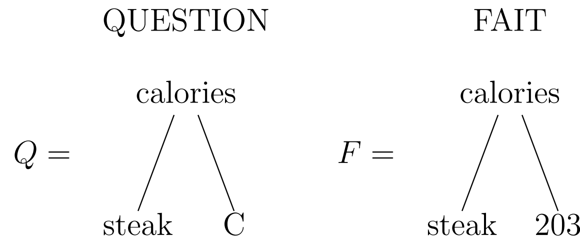
Exemple : "est-il vrai que l'avocat est une entrée"
`entree(avocat).`

Prolog cherche dans sa base la présence de ce fait, il répond vrai, `yes`, s'il est présent, faux, `no`, sinon.

Exemple : " quel est le nombre de calories de 100 grammes de steak"
`calories(steak,C).`
Prolog répond : `C=203`
`yes`

**remarque** : `C` (lettre seule) est une variable Prolog, c'est à dire une entité non encore déterminée qui peut donc prendre une valeur afin de permettre de rendre vraie une formule (analogie avec les inconnues d'équation en maths).

**Mécanisme utilisé pour répondre à une question (approche informelle)** Prolog tente de faire correspondre la question avec les faits disponibles. Si un fait s'accorde avec la question alors les variables de la question prennent les valeurs nécessaires. On dit que l'on **unifie** la question avec le fait et que les variables sont **instanciées**. Cette instanciation se fait par construction d'une **substitution** qui est un ensemble des couples $(variable, valeur)$, généralement représenté sous la forme $\{Var_1 = Val_1, ..., Var_n = Val_n\}$.
Exemple (on représente les prédicat par des arbres) :




$Q$ et $F$ sont unifiables par la **substitution** $\sigma = \{\texttt{C} = 203\}$.

**Définition 9** *Deux prédicat $P1$ et $P2$ sont* **Unifiables** *si on peut trouver une substitution $\sigma$ telle que $P1\sigma = P2\sigma$.*

**Définition 10** *Une* **Substitution** *est une liste de couples $(v, t)$ tels que le premier élément est une variable et le second un terme Prolog contenant éventuellement des variables.*

**Définition 11 Instancier** *un prédicat $P$ avec une substitution $\sigma$ (noté $P\sigma$), c'est remplacer dans $P$ toutes les occurrences d'une variables de $\sigma$ par son expression associée dans $\sigma$.*

Dans l'exemple précédent en instanciant $Q$ par $\sigma$, c'est à dire en remplaçant dans $Q$ les variables par leurs valeurs dans la substitution $\sigma$, on obtient :

$$Q\sigma = \texttt{calories}(\texttt{steak}, \texttt{C})\{\texttt{C} = 203\} = \texttt{calories}(\texttt{steak}, 203) = F$$

Exemple : soit $P1 = \texttt{aime}(\texttt{jean}, \texttt{A})$ et $P2 = \texttt{aime}(\texttt{B}, \texttt{laLogique})$. La substitution $\sigma = \{\texttt{A} = \texttt{laLogique}, \texttt{B} = \texttt{jean}\}$ permet d'unifier $P1$ et $P2$.

**Règles** Un programme Prolog comporte des faits et des règles. Les règles expriment des relations conditionnelles entre les objets (clauses de Horn).

Exemple : "le **plat** principal est soit de la **viande** soit du **poisson**"

Cet énoncé conduit à la définition de deux règles Prolog :

```
r1 :  plat(V) :-  viande(V).
r2 :  plat(P) :-  poisson(P).
```

Remarque : il y a un "ou" implicite entre r1 et r2.

La **tête de la règle** exprime la conséquence, la **queue** exprime une conjonction de conditions (ET implicite). Autrement dit une règle :

**Tete :- Queue .**

peut se lire :

"Pour que `Tete` soit vérifiée il faut que les différentes conditions de `Queue` soient vérifiées".

On remarquera qu'il existe une disjonction (OU implicite) entre les deux règles r1 et r2.

Pour compléter notre exemple nous pouvons exprimer la notion de repas que ce début d'exemple suscite en exprimant qu'il est constitué d'une **entrée** d'un **plat** principal et d'un **dessert**.

```
r3 :   repas(E,P,D) :-  entree(E),plat(P),dessert(D).
```

Remarque : il y a un "et" implicite entre `entree(E)`, `plat(P)` et `dessert(D)` : "Un repas est constitué d'une entrée ET d'un plat ET d'un dessert".

En ajoutant les règles r1, r2 et r3 (ou clauses) à la base de Prolog, on peut maintenant construire des repas.

Des questions peuvent être :

`?- repas(crepes,escalope,raisin).` "Peut-on avoir un repas comportant des crêpes, de l'escalope et du raisin ?"

`no`, réponse NON

`?-repas(E,escalope,D).` "quels sont les repas comportant de l'escalope ?"

```
E=salade
D=raisin ? ;
E=salade
D=melon ? ;
E=avocat
D=raisin ? ;
E=avocat
D=melon ? ;
E=huitre
D=raisin ? ;
E=huitre
D=melon ?
```
6 solutions de repas avec de l'escalope

Remarque : Lorsque gprolog pense qu'il peut y avoir d'autres réponses possibles, il demande à l'utilisateur s'il doit continuer l'exploration en utilisant le caractère "`?`". Si l'utilisateur appuie sur "`;`", gprolog cherche la réponse suivante. S'il en trouve une autre il l'affiche sinon il affiche `no` pour dire qu'il n'y en a pas d'autres. Si l'utilisateur appuie sur "Entrée", gprolog arrête l'exploration. Enfin il est également possible d'appuyer sur `''a''` pour obtenir toutes les autres solutions[1].

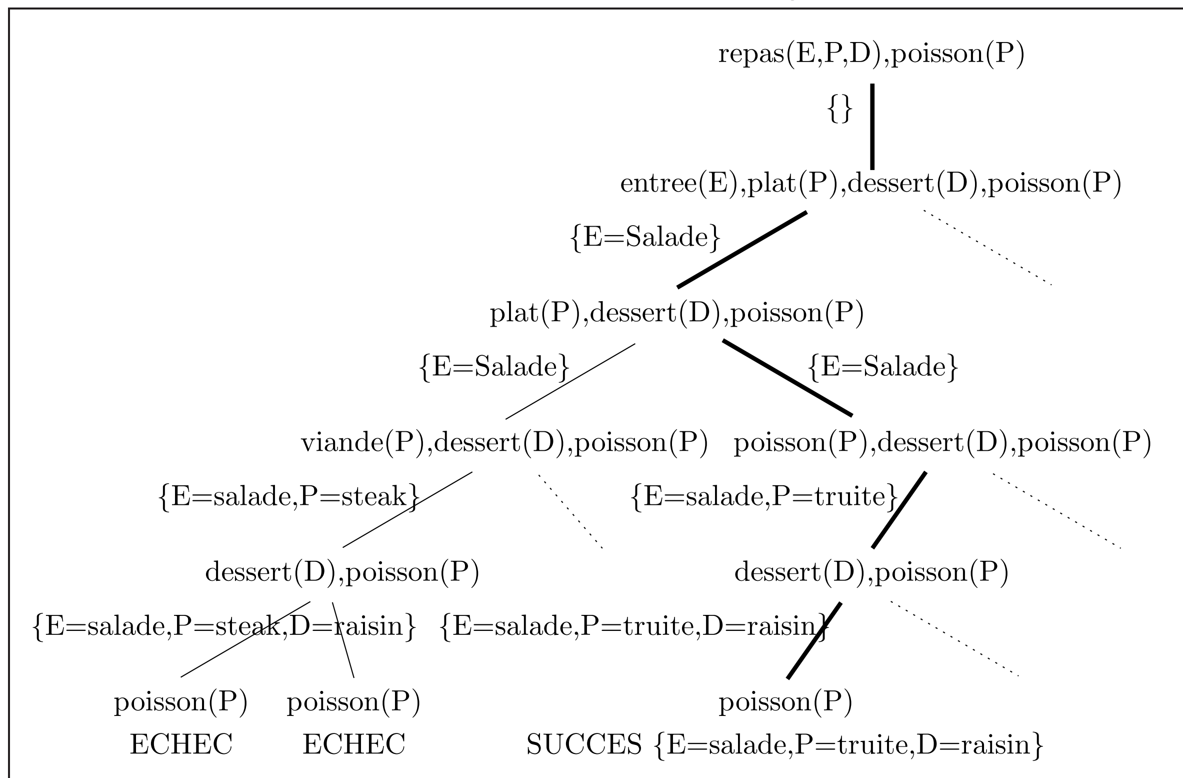
Les questions peuvent être elles-mêmes des conjonctions :

[1] Attention le nombre de solutions est éventuellement très grand, voir infini !

```
?- repas(E,P,D),poisson(P).
```

Cette question demande l'ensemble des repas qui comportent comme plat principal du poisson. Si dans le principe cette question est semblable à `poisson(P),repas(E,P,D)` du point de vue de l'exécution elles se différencient par leur évaluation (et leur coût).

On remarquera que la résolution des buts de la question se fait de gauche à droite.




# 2 L'univers de gprolog

## 2.1 Les termes

Le domaine de gprolog est constitué d'objets appelés **termes**. Il y a quatre sortes de termes : les atomes, les variables, les nombres et les termes composés.

### 2.1.1 Les atomes

Un atome est soit :
- Une chaîne de caractères faite de majuscules, minuscules de chiffres et du blanc souligné, qui commence par une minuscule. Par exemple : `marylin_monroe`, `joe123`.
- Une suite arbitraire de caractères entre guillemets simples. Par exemple `'Je suis un atome'`, `'l"arbre'`. Évidemment il n'est pas possible de mettre de guillemets simples dans un atome.
- Une suite de caractères spéciaux. Par exemple : `,` et `:-` sont des atomes qui ont un sens prédéfini.

Notons que si un atome peut s'écrire sans guillemets simples, il est considéré comme identique au même atomes sans les guillemets : `abcd='abcd'` et `=<='=<'`.

### 2.1.2 Les variables

Une variable est une suite de majuscules, de chiffres et de blancs soulignés qui commence par une majuscule ou par un blanc souligné. Par exemple `X`, `Y`, `Liste`, `_tag`, `X_67` sont des variables.
La variable `_` est appelée la variable anonyme. Elle indique à Prolog que l'on ne s'intéresse pas à la valeur de cette variable. Par exemple la requête `entree(_)` demande juste s'il existe une entrée, peut importe son nom.

### 2.1.3 Les nombres

On peut utiliser les entiers ou les réels (flottants) : `33`, `-33`, `33.0`, `33E+02`.

### 2.1.4 Les termes composés

Les termes sont définis selon la grammaire suivante :

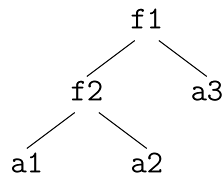
`Terme` ::= `Atome` | `Variable` | `Nombre` | `Terme-composé`
`Terme-composé`::= `Atome`(`Terme`{, `Terme`})

L'atome qui débute un terme composé est le foncteur du terme. Par exemple `calorie` est le foncteur de `calorie(salade,15)`.
L'arité d'un prédicat est le nombre d'arguments des têtes de clauses définissant ce prédicat. Par exemple `entree` est d'arité 1 et `calorie` est d'arité 2. Il est possible de définir plusieurs prédicats d'arités différentes mais avec le même foncteur. On les distingue alors en utilisant la notation `Prédicat/Arité`, représentant le **profil** du prédicat ou du terme. Par exemple `calorie/2` est le profil du prédicat `calorie(X,Y)`.

Il est possible de tester à quelle catégorie appartient un terme en utilisant les prédicats prédéfinis suivants : `atom/1`, `var/1`, `nonvar/1`, `integer/1`, `float/1`, `number/1`, `atomic/1`, `compound/1` et `callable/1`.

### 2.1.5 Représentation des termes

Généralement les termes composés sont représentés sous forme d'arbre. Ainsi le terme `f1(f2(a1,a2),a3)` est représenté par l'arbre suivant :




# 3 Arithmétique en gprolog

## 3.1 Les expressions arithmétiques

Une expression arithmétique est un terme fait de nombre et de foncteurs ou opérations représentant des fonctions arithmétiques. Les principaux opérateurs sont : `+`, `-`, `*`, `/` (division réelle), `//` (division entière), `rem` (reste) et `**` (puissance).

## 3.2 Prédicats arithmétiques

Les opérateurs suivants permettent de manipuler les expression arithmétiques :

1. `Expr1 =:= Expr2` réussit si les *valeurs* des deux expressions sont égales. Par exemple `2*3=:=1+5` réussit mais `2*3=:=5` échoue.
   Il ne faut pas confondre ce prédicat avec le prédicat d'unification `=`. Par exemple `2*3=6` échoue parce que les deux termes ne sont pas unifiables. Le terme `2*3` est un terme composé (en fait il correspond au terme `'*'(2,3)`) alors que `6` est un entier.
2. `Expr1 =\= Expr2` réussit si les valeurs des deux expressions sont différentes.
3. `Expr1 < Expr2` réussit si la *valeur* de `Expr1` est strictement inférieure à celle de `Expr2`.
4. `Expr1 =< Expr2` réussit si la *valeur* de `Expr1` est inférieure ou égale à celle de `Expr2`.
5. `Expr1 > Expr2` réussit si la *valeur* de `Expr1` est strictement supérieure à celle de `Expr2`.
6. `Expr1 >= Expr2` réussit si la *valeur* de `Expr1` est supérieure ou égale à celle de `Expr2`.

Pour chacun de ces prédicats, si une des deux expressions n'est pas évaluable parce qu'elle contient des variables non instanciées, gprolog retourne un message d'erreur. La requête `X=:=3` produit un message d'erreur alors que la requête `X=3,X=:=1+2` réussit.

Les prédicats `=` et `\=` permettent de tester si des termes sont unifiables ou non. La façon dont gprolog gère les unifications est présentée dans la section 5.1.

### 3.3 Le prédicat is/2

La requête `R is Expr` réussit si la valeur de `Expr` s'unifie avec `R`. Ce prédicat sert généralement à donner une valeur à une variable. Par exemple la requête `X is 2+3` donne à `X` la valeur `5` et `X is Y+1` donne à `X` la valeur de `Y` plus 1 si `Y` a une valeur. Si `Y` n'a pas encore de valeur, gprolog produit un message d'erreur.

Attention la requête `2+3 is 5` échoue puisque le terme `2+3` (correspondant en fait au terme `'+'(2,3)`) ne s'unifie pas avec `5`.

# 4 Définition d'un programme Prolog

Un programme Prolog correspond à un prédicat. Il est composé d'un ensemble de clauses qui sont soit des faits, soit des règles. On parle aussi de paquet de clauses. Toutes les têtes des clauses d'un programme Prolog doivent avoir le même profil, c'est à dire le même foncteur et la même arité.

Les clauses :

```
fct(0,1).
fct(X,Y) :- ... .
```

constituent un même programme, correspondant au prédicat `fct/2` alors que les clauses suivantes définissent 2 programmes différents :

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

## 4.1 Exemples de programmes Prolog

### 4.1.1 Le restaurant (suite)

Soit la base de faits Prolog suivante :

```
 entree(artichauts).  viande(grillade).  dessert(sorbet).
 entree(avocat).      viande(poulet).    dessert(fraise).
 entree(cresson).     poisson(dorade).   poisson(bar).
```

et les règles :

```
plat(P) :- viande(P).
plat(P) :- poisson(P).
repas(E,P,D) :- entree(E),plat(P),dessert(D).
repasCal(E,P,D,C) :- repas(E,P,D),calories(E,C1),calories(P,C2),calories(D,C3),
                     C is C1+C2+C3.
```

Exercice :

1. Écrire la clause qui décrit un repas de régime hypocalorique (valeur calorifique $< 600$).
2. Compléter le programme repas en ajoutant une boisson à choisir parmi le vin blanc (`meursault`, `chablis`), le vin rouge (`vougeot`, `cornas`) et l'eau (`vals`, `evian`). Ajouter les règles qui permettent de ne servir que du vin rouge avec la viande et du vin blanc ou de l'eau avec le poisson.

### 4.1.2 Élévation à la puissance

On réalise un prédicat puissance d'arité 3 : `puissance(X,N,P)`, signifiant que $\mathtt{X}^{\mathtt{N}} = \mathtt{P}$.
On s'appuie sur la définition récurrente :

1. $X^0 = 1$
2. $X^n = X^{n-1} * X$, avec $n > 0$

d'où le programme Prolog suivant :

```
puissance(_,0,1).
puissance(X,N,P) :- N>0, N1 is N-1, puissance(X,N1,P1) , P is P1*X.
```

On remarquera l'utilisation de la variable anonyme dans la première règle. C'est parce que la valeur du premier paramètre n'a aucune importance dans le cas où le deuxième vaut `0`. Si nous utilisons une variable "non anonyme" telle que `X`, gprolog affiche un avertissement disant que `X` n'apparaît qu'une fois dans la règle. Il faut alors se demander si `X` peut être remplacé par la variable anonyme ou si on a juste oublié d'utiliser `X` ailleurs dans la règle.

# 5 Le fonctionnement de Prolog

## 5.1 L'unification

L'unification est l'opération élémentaire que réalise Prolog pour rendre deux termes identiques. Cette opération consiste à établir une substitution (ensemble de couples (variable, valeur)) qui permettra d'obtenir une équivalence des deux termes. Cette substitution peut être assimilée à un ensemble de contraintes que l'on fait porter sur les variables de la substitution. Cette opération peut échouer si aucune substitution ne permet d'obtenir cette équivalence. L'unification est dépendante du fait qu'apparaissent ou non des variables dans les termes, que les termes soient atomiques ou non.

- Deux termes atomiques sont unifiables s'ils sont identiques,
- une variable libre s'unifie toujours en s'instanciant à l'autre terme,
- une variable instanciée se comporte comme son terme,
- l'unification de deux variables libres les lie,
- toutes les variables liées ne peuvent s'instancier qu'au même terme,
- deux termes composés s'unifient composant à composant.

| Terme 1 | Terme 2 | ?ok | substitution | commentaire |
|---|---|---|---|---|
| `aa` | `aa` | oui | { } | deux termes atomiques sont unifiables s'ils sont identiques |
| `aa` | `bb` | non | | |
| `A` | `bb` | oui | {`A=bb`} | si un des deux termes est une variable libre, l'unification est toujours possible |
| `a`<br>`b` `X` | `a`<br>`Y` `c` | oui | {`X=c,Y=b`} | |
| `a`<br>`X` `b`<br>`Y` `c` | `a`<br>`Y` `b`<br>`c` `X` | oui | {`X=c,Y=c`} | les variables `X` et `Y` sont liées |
| `a`<br>`X` `b`<br>`Y` `c` | `a`<br>`Y` `b`<br>`d` `X` | non | {`X=Y`,<br>`X=c`,<br>`Y=d`} | l'unification de `X` avec `c` nécessiterait l'unification des deux termes atomiques `c` et `d`. |
| `a`<br>`b` `c`<br>`d` `X` | `a`<br>`X` `Y` | oui | {`X=b`,<br>`Y=c(d,b)`} | on donne la valeur de `Y` en remplaçant `X` par sa valeur |

En gprolog l'unification de deux termes se note en utilisant le prédicat `terme1 = terme2` et la non unification se note en utilisant `terme1 \= terme2`.

## 5.2 Sémantique déclarative

On utilise Prolog avec une approche relationnelle. Prolog permet d'établir des relations entre des objets. Si dans la question on utilise des variables, alors Prolog va fournir la substitution (ensemble de valeurs des variables) qui permet de former une relation déductible de la base initiale.

## 5.3 Sémantique procédurale

### 5.3.1 Principe

On peut considérer le "**moteur**" de la résolution de Prolog comme une **boite noire** qui reçoit en entrée une question et produit en sortie soit un échec soit un succès constitué par un ensemble de substitutions. Pour obtenir les solutions à la question posée Prolog utilise la base initiale pour prouver les affirmations non directement présentes dans la base en faisant des unifications.

### 5.3.2 Algorithme de la résolution (simplifié)

Une question est représentée par une **résolvante** qui est constituée d'une **conjonction de termes**. La **cardinalité** d'une résolvante est le nombre de termes qui la compose.

$$T_1(A_1^1, ..., A_N^1), T_2(A_1^2, ..., A_M^2), ..., T_K(...)$$

Prolog va chercher à **effacer** tous les termes de la résolvante. Il essaie par dérivation successive d'obtenir une résolvante vide (de cardinalité 0). S'il y arrive alors il y a **succès** et la substitution obtenue donne les conditions de succès. S'il n'y parvient pas il y a alors **échec**. Si on note $R_i$ une résolvante, on peut schématiser le principe de Prolog de la manière suivante : $R_1$ est la résolvante initiale correspondant à la question.

$$R_1 \rightarrow R_2 \rightarrow R_3 \rightarrow ... \rightarrow R_N \rightarrow \emptyset$$

L'effacement d'un terme se fait en remplaçant le terme par un équivalent, on peut parler de **réécriture**. Les réécritures sont décrites par les faits et les règles introduites dans la base initiale. La règle "`T :- Q.`" établit que `T` peut être remplacé par `Q`. Le fait "`T.`" établit que `T` se remplace par vide ($\emptyset$).

Pour passer de la résolvante $R_i$ à la résolvante $R_{i+1}$ Prolog doit choisir un terme pour essayer de l'effacer. Prolog fonctionne de **gauche à droite** et en **profondeur** d'abord. Il choisit donc de réécrire le terme $T_1$ et place la réécriture du terme $T_1$ en tête de la nouvelle résolvante. La substitution $\sigma_i$ obtenue pour unifier $T_1$ avec la tête de la règle choisie est appliquée à tous les autres termes de la résolvante :

$$\begin{array}{rcl} R_i & = & T_1(A_1^1...A_N^1), T_2(A_1^2...A_M^2), ..., T^K(...) \\ R_{i+1} & = & Q, T_2(A_1^2, ..., A_M^2)\sigma_i, ..., T_K(...)\sigma_i \end{array}$$

L'effacement d'un terme à l'aide d'un fait est tel que $Q = \emptyset$ et donc :

$$card(R_{i+1}) = card(R_i) - 1$$

On peut donc comprendre que Prolog ne dérive vers la résolvante vide que par l'utilisation de faits de la base initiale. L'effacement d'un terme à l'aide d'une règle est tel que $Q \neq \emptyset$ et donc :

$$card(R_{i+1}) \geq card(R_i)$$

Pour effectuer l'effacement du terme $T_1$ Prolog recherche dans la base initiale un clause dont la tête s'unifie avec le terme $T_1$. Si une telle clause n'existe pas, il ne peut pas y avoir effacement et donc Prolog ne peut dériver vers la résolvante vide ce qui conduit à un échec. Si une telle clause existe :

$T_1(U_1^1...U_N^1)$ `:-` $Q$.

la substitution $\sigma_i$ permettant l'unification de $T_1(U_1^1...U_N^1)$ et de $T_1(A_1^1...A_N^1)$ est mémorisée. En cas de réussite de la résolution, la substitution $\sigma$ permettant le succès est construite à partir de toutes les substitutions $\sigma_i$ utilisées lors des dérivations.

Prolog étant un langage non déterministe, il cherche à produire toutes les solutions (système de contraintes) qui permettent de satisfaire la question. Il va donc chercher à réaliser tous les effacements possibles pour un terme de la résolvante et va parcourir un **arbre de résolution**. Il fait ce parcours en profondeur d'abord.

Afin d'effacer le terme $T_1$ = `pre(A1,...An)`, Prolog parcourt l'ensemble des clauses correspondant au programme `pre/n` dans l'ordre dans lequel elles ont été définies dans la base initiale. Cet ordre a donc une très grande influence sur le comportement à l'exécution d'une résolution. Lorsque Prolog ne peut pas trouver de clause pour réécrire $T_1$ il **backtrack** (retourne en arrière) pour explorer un autre chemin de résolution. Il le fait aussi lorsqu'il a obtenu une résolvante vide (une solution) pour fournir les autres solutions potentielles. Prolog explore donc un arbre de résolvante dont les feuilles sont soit des résolvantes vides soit des résolvantes dont le premier terme (gauche) ne peut être effacé. Lorsqu'il backtrack d'une résolvante à la résolvante précédente Prolog défait toutes les substitutions introduites lors des unifications successives produites à partir de cette résolvante.

### 5.3.3 Exemple de résolution

Prenons l'exemple du calcul de la multiplication d'un entier naturel `X` par un entier `Y` par additions successives (multiplication Égyptienne). Nous donnons volontairement une expression de l'algorithme qui n'est pas réduite.

```
r1      mlt(X,Y,R) :- Y<0,Y1 is -Y,mlt(X,Y1,R1),R is -R1.
r2      mlt(_,0,0). /* 0 est élément absorbant de la multiplication */
r3      mlt(X,1,X). /* 1 est élément neutre de la multiplication */
r4      mlt(X,Y,R) :-Y1 is Y-1,Y1>0,mlt(X,Y1,R1),R is R1+X.
```

Puisque ce programme est récursif, la même règle peut être utilisée plusieurs fois dans la résolution. Afin d'éviter toute confusion sur les variables telles que `X`, `Y` et `R` qui pourraient être apparaître plusieurs fois, nous effectueront des renommage du type `_iX`, `_iY` et `_iR`. Ce renommage est effectué en interne par le moteur de résolution de Prolog.

`mlt(10,-2,A)`
    tentative d'unification avec r1 réussie $\sigma_1 = \{_1X = 10, _1Y = -2, _1R = A\}$
`-2<0,_1Y1 is --2,mlt(10,_1Y1,_1R1),A is -_1R1`
    `-2<0` et `_1Y1 is --2` s'éliminent et produisent $\sigma_2 = \{_1Y1 = 2\}$
`mlt(10,2,_1R1),A is -_1R1`
    tentative d'unification avec r1 réussie avec $\sigma_3 = \{_2X = 10, _2Y = 2, _2R = _1R1\}$
`2<0,_2Y1 is -2,mlt(10,_2Y1,_2R1),_1R1 is -_2R1,A is -_1R1`
    backtrack car `2<0` est faux.
`mlt(10,2,_1R1),A is -_1R1`
    tentative d'unification avec r2 échoue 2 n'est pas nul
    tentative d'unification avec r3 échoue 2 n'est pas égal à 1
    tentative d'unification avec r4 réussie $\sigma_3 = \{_3X = 10, _3Y = 2, _3R = _1R1\}$
`_3Y1 is 2-1,_3Y1>0,mlt(10,_3Y1,_3R1),_1R1 is _3R1+10,A is -_1R1`
    élimination des deux premières clauses avec la substitution $\sigma_4 = \{_3Y1 = 1\}$
`mlt(10,1,_3R1),_1R1 is _3R1+10,A is -_1R1`
    tentative d'unification avec r1 réussie avec $\sigma_5 = \{_4X = 10, _4Y = 1, _4R = _3R1\}$
`1<0,_4Y1 is -1,mlt(10,_4Y1,_4R1),_4R1 is -_3R1,_1R1 is _3R1+10,A is -_1R1`
    backtrack car `1<0` est faux.
`mlt(10,1,_3R1),_1R1 is _3R1+10,A is -_1R1`
    tentative d'unification avec r2 échoue 2 n'est pas nul
    tentative d'unification avec r3 réussie avec $\sigma_6 = \{_3R1 = 10\}$
`_1R1 is 10+10,A is -_1R1`
    élimination des deux derniers termes avec $\sigma_7 = \{_1R1 = 20, A = -20\}$
    réussite avec la solution `A=-20`.

En exercice :

1. Vérifiez qu'il n'y a pas d'autres solutions possibles.
2. Écrivez une version de `mlt` avec seulement 3 règles.

## 5.4 Exemples de programmes

### 5.4.1 La factorielle

Plusieurs programmes peuvent être la solution à un même problème. Prolog ne déroge pas à cette constatation, et nous allons dans cet exemple donner plusieurs versions pour écrire le calcul de la factorielle. Factorielle que nous dénoterons "`fac`" est un prédicat qui associe un entier positif ou nul et un entier positif tel que le second est égal à la factorielle du premier.

Voici une première version du programme :

```
fac(0,1).
fac(N,R) :- N1 is N-1,fac(N1,R1),R is R1*N.
```

Cette définition Prolog est très proche de la définition récurrente que l'on fait de la factorielle. La première clause du programme correspond au cas de base de la récurrence et exprime que la factorielle de 0 est 1. La seconde clause du programme exprime que la factorielle de `N` est égale à $R1 * N$ où `R1` est la factorielle de $N - 1$. Ayant introduit ce programme dans la base initiale, on peut poser des questions concernant ce prédicat.

```
?- fac(0,1).
true ?
yes
```

On a dans cet exemple posé une question sans variable, le système Prolog va donc répondre vrai (`true`) si la relation exprimée dans cette question est vérifiable ou faux (`no`) sinon. Ici la solution est simple puisqu'un fait de la base initiale exprime directement cette relation. Néanmoins puisque le terme de la question peut s'unifier avec la deuxième règle de `fac`, Prolog demande s'il doit continuer l'exploration pour trouver une autre solution. En appuyant sur "Entrée" on lui dit d'arrêter là et il répond `yes` car il a trouvé une solution. Sinon en appuyant sur "`;`" il cherche d'autres solutions et répond `no` puisqu'il n'y en a pas d'autres.

On peut bien évidemment obtenir une réponse similaire par déduction comme dans l'exemple suivant :

```
?- fac(4,24).
true ?
yes
```

On peut poser une question qui comporte des variables, Prolog répond alors en fournissant les contraintes portant sur ces variables. dans l'exemple suivant `fac(4,F)` est une relation valide si `F` est égale à `24`.

```
?- fac(4,F).
F=24 ?
yes
```

Prolog étant relationnel dans un prédicat les arguments ne sont pas différenciables comme dans un système fonctionnel où il y a des arguments en entrée et d'autres en sortie. cela veut dire que n'importe quel argument d'une question peut être remplacé par une variable . Néanmoins ce programme ne marche pas si le premier argument est une variable :

```
?- fac(N,6).
uncaught exception : error(instantiation_eror,(is)/2)
```

Cela vient du prédicat `is/2` qui a besoin d'avoir à droite une expression évaluable. Or dans ce cas, `N` étant libre, `N-1` n'est pas évaluable. Nous verrons plus tard dans le cours une autre version de `fac` qui marche si le premier paramètre est une variable.

Ce programme pose un autre problème : il ne termine pas si on lui demande de chercher une autre solution :

```
?- fac(0,1).
true ? ;
Fatal Error : local stack overflow
```

Observons le déroulement de la résolution :

`fac(0,1)`

  élimination du terme par unification avec la première règle
  demande d'une autre solution, donc retour au terme précédent

`fac(0,1)`

  unification avec la seconde règle avec $\sigma_1 = \{_1N = 0, _1R = 1\}$

`_1N1 is 0-1,fac(_1N1,_1R1),1 is _1R1*0`

  élimination du `is` avec $\sigma_2 = \{_1N1 = -1\}$

`fac(-1,_R1),1 is _1R1*0`

  application de la première règle impossible
  unification avec la seconde règle avec $\sigma_3 = \{_2N = -1, _2R = _R1\}$

`_2N1 is -1-1,fac(_2N1,_2R1),_R1 is _2R1*-1,1 is _1R1*0`

  élimination du `is` avec $\sigma_4 = \{_2N1 = -2\}$

`fac(-2,_2R1),_R1 is _2R1*-1,1 is _1R1*0`

...

L'exécution est infinie puisqu'à chaque fois l'unification réussie avec la seconde règle et qu'il n'est plus possible d'unifier avec la première règle, qui est la seule pouvant arrêter le calcul. La mémoire "explose" parce que la clause grossit à chaque fois.

Le problème vient du fait que rien n'empêche l'utilisation de la seconde règle avec des valeurs négatives pour le premier paramètre. Or la factorielle n'est définie que pour les valeur positives. Il faut donc rajouter ce test dans le programme. Nous obtenons donc :

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

Remarque : l'ajout du test à la fin de la queue n'empêcherait pas la non terminaison. L'ordre dans lequel sont donnés les termes de la queue est donc très importante.

### 5.4.2 Concaténation de listes

Dans gprolog les listes sont représentées en extension, par exemple `[X1,X2,X3]` pour une liste de longueur 3, ou par l'ajout d'un élément à gauche, sous la forme `[X|L]`, où `X` est ajouté à la liste `L`. Une présentation plus complète se trouve dans la section 8.

Il s'agit d'un exemple classique d'utilisation de l'aspect relationnel de Prolog. `concat` est un prédicat qui possède 3 arguments qui sont des listes tel que le 3-ième argument est la concaténation des 2 premiers.

```
concat([],Z,Z).
concat([X|Y],Z,[X|R]) :- concat(Y,Z,R).
```

On peut envisager 4 situations distinctes d'utilisation de ce prédicat en fonction de la nature variable ou constant des arguments.

- **Situation 1 :** tous les arguments sont instanciés.
  On vérifie que les 3 arguments respectent la spécification de la concaténation.
  ```
  ?- concat([a,b,c],[d,e,f],[a,b,c,d,e,f]).
  yes
  ?- concat([1,2],[3],[1,2,3,4]).
  no
  ```
  Prolog répond juste avec `yes` ou `no`.
- **Situation 2 :** un des arguments est libre.
  Prolog cherche alors une substitution permettant de satisfaire la spécification du prédicat :
  ```
  ?- concat([1,2],[3],Z).
  Z=[1,2,3]
  yes
  ?- concat([1,2],Z,[1,2,3]).
  Z=[3]
  yes
  ?- concat(Z,[1,2,3],[1,2,3]).
  Z=[] ?
  yes
  ```
  Remarquez que dans le dernier cas Prolog ne se rend pas compte tout seul qu'il n'y a pas d'autres possibilités. Cela vient du fait que pour discriminer les clauses qui ne permettront pas de donner de solutions, Prolog essaie juste d'instancier le premier argument du premier terme. Dans ce cas `Z` pouvant s'unifier avec `[]` pour la première clause et avec `[X|Y]` pour la seconde clause, Prolog pense que la deuxième clause peut peut-être donner une autre solution.
- **Situation 3 :** deux des arguments sont des variables. Prolog va chercher une substitution pour ce couple de variables. On peut distinguer trois sous cas suivant quel est l'argument constant.
  1. Le troisième argument est constant
     ```
     ?- concat(X,Y,[1,2,3]).
     X = []
     Y = [1,2,3] ? ;
     X = [1]
     Y = [2,3] ? ;
     X = [1,2]
     Y = [3] ? ;
     X = [1,2,3]
     Y = [] ? ;
     no
     ```
     Prolog énumère toutes les substitutions $\sigma$ telles que `[1,2,3]` est la concaténation de $X\sigma$ et $Y\sigma$.
  2. Le premier argument est constant
     ```
     ?- concat([1,2,3],X,Y).
     Y = [1,2,3|X]
     yes
     ```
     Il n'y a qu'une solution mais dans laquelle `X` est une liste quelconque.
  3. Le second argument est constant

```
?- concat(X,[1,2,3],Y).
X = []
Y = [1,2,3] ? ;
X = [A]
Y = [A,1,2,3] ? ;
X = [A,B]
Y = [A,B,1,2,3] ?
yes
```
Il y a une infinité de solutions.

– **Situation 4 :** tous les arguments sont des variables
```
?- concat(X,Y,Z).
X = []
Z = Y ? ;
X = [A]
Z = [A|Y] ? ;
X = [A,B]
Z = [A,B|Y] ? ;
X = [A,B,C]
Z = [A,B,C|Y] ?
yes
```
Prolog fournit toutes les solutions formelles.

# 6 Solution des exercices

## 6.1 Le restaurant

1. Repas hypocalorique :
```
regime(E,P,D) :-repasCal(E,P,D,C),C<600.
```
2. Ajout des boissons :
```
eau(vals).
eau(evian).
rouge(vougeot).
rouge(cornas).
blanc(meursault).
blanc(chablis).
boisson(B) :- rouge(B).
boisson(B) :- blanc(B).
boisson(B) :- eau(B).
repas(E,P,D,B) :- repas(E,P,D),boisson(B).
```
Choix des boissons :
```
repas2(E,P,D,B) :- repas(E,P,D,B),poisson(P),blanc(B).
repas2(E,P,D,B) :- repas(E,P,D,B),poisson(P),eau(B).
repas2(E,P,D,B) :- repas(E,P,D,B),viande(P),rouge(B).
```

## 6.2 La multiplication

Version simplifiée :
```
mlt(_,0,0). /* 0 est élément absorbant de la multiplication */
mlt(X,Y,R) :- Y>0,Y1 is Y-1,mlt(X,Y1,R1),R is R1+X.
mlt(X,Y,R) :- Y<0,Y1 is -Y,mlt(X,Y1,R1),R is -R1.
```

Deuxième partie

# Concepts algorithmiques

## 7 Spécification de prédicats

### 7.1 Définition

La spécification est une description permettant de définir le rôle et les conditions d'utilisation d'un prédicat.
Elle établit un "contrat" entre l'utilisateur du prédicat (de la fonction, du programme) - appelé le "client", et celui qui réalise le prédicat - le "fournisseur".
Pour le client : la spécification énonce les conditions qui doivent être satisfaite par le client avant la résolution et définit les conditions qu'on peut en attendre en retour.
Pour le fournisseur : la spécification définit les conditions que l'on est sur d'avoir en entrée et les conditions de sortie qui doivent être satisfaites.

En Prolog : soit un prédicat de profil P/n, c'est à dire qui se représente sous la forme :
`P(a1,a2,...,an)`
La spécification doit préciser :
- les domaines de définitions de chacun des paramètres
- les conditions d'instanciation des différents paramètres avant l'appel, appelés `modes` :
  - `+a` : indique le paramètre doit être instancié
  - `-a` : indique que le paramètre doit être libre
  - `?a` : instanciation indifférente

  (rem : on peut définir plusieurs modes ex : `P(+a,-b)`, `P(-a,+b)`)
- la relation définie par le prédicat (rôle du prédicat)
- enfin, on peut avoir à préciser si le prédicat est "backtrackant", c'est à dire s'il peut conduire à plusieurs solutions.

### 7.2 Exemples

factorielle :
```
/* N entier >=0, R entier
fact(+N,?R) : R est la factorielle de N */
```
concaténation :
```
/* A, B, C : listes d'éléments
concat(?A,?B,?C) : C est la concaténation de A et B (C=A.B)
concat(-A,-B,?C), concat(-A,?B,-C) sont backtrackants */
```

## 8 Récursivité et traitement des listes

### 8.1 Introduction

(cf cours d'info "langages et programmation" Scholl & al. THÈME 1- notation fonctionnelle)

**Définition 12** *La* **récursivité** *c'est la possibilité de définir une entité, un objet, un type, une fonction, un prédicat ... en terme de lui-même.*

Exemple : un entier naturel se définit par :
  0 ou n+1 où n est un entier naturel.
Elle fait appel au raisonnement par récurrence (maths)

#### 8.1.1 Définition récursive d'un prédicat

La définition d'un prédicat est dite récursive si au moins une règle du programme Prolog correspondant comporte d'un sa queue au moins une occurrence du prédicat. S'il y a plusieurs occurrence du prédicat, on parle de récursivité multiple.

Récursivité directe :
`P(a,b) :- Q(c,d), P(e,f).`
Récursivité croisée :
```
P(a,b) :- Q(c,d).
Q(a,b) :- P(c,d).
```

### 8.1.2 Exemples

les prédicats `fact`, `concat`, `mult` vus précédemment.
Suite de Fibonacci :
définition récurrente : `F(0) = F(1) = 1`
`F(N+2) = F(N+1)+F(N), N≥0`

| | |
|---|---|
| n : | 0 1 2 3 4 5 |
| F(n) : | 1 1 2 3 5 8 |

Programme Prolog :
```
/* N et R entiers >=0
fibo(+N,?R) : R est le Nième nombre de la suite de Fibonacci */
(r1)    fibo(0,1).
(r2)    fibo(1,1).
(r3)    fibo(N,R) :- N>=2, N1 is N-1, N2 is N-2,
                     fibo(N1,R1),fibo(N2,R2),R is R1+R2.
```

## 8.2 Définition récurrente d'un type (ou type inductif)

Un prédicat récursif s'appuie sur les définitions récurrentes des domaines des objets (types) qu'il manipule. Rappel sur le types : un type est un ensemble d'objets munis d'opérations (constructeurs, testeurs, sélecteurs).

Exemple : le type date ($heure, minute, seconde$)
constructeur `date(H,M,S,D)` : `D` est la date construite avec `H`, `M` et `S`
testeurs : `avant(D1,D2)` : la date `D1` est avant la date `D2`
sélecteur : `heure(D,H)` : `H` est l'heure de la date `D` etc...

Un type peut se définir en extension ou en compréhension (par ex. pour les entiers naturels impairs I : extension : I= {1,3,5,7,...}, compréhension : I= $\{x \in \mathbb{N} | (x + 1)$ est divisible par $2\}$) ou de manière récurrente.

### 8.2.1 Définition récurrente

On définit l'ensemble de base et l'ensemble des règles de récurrence :
*base* : ensemble des éléments de base qui permettent de construire les autres éléments.
*règles de récurrence* : règles qui définissent la manière de construire un élément de l'ensemble à partir d'autres éléments.

Pour I :
base= {1}
récurrence= si $x \in$ I , alors $x + 2 \in$ I.

Pour toute définitions récurrentes il faut :
- décrire la base
- décrire les règles de récurrence
- s'assurer que les règles permettent d'atteindre tous les éléments

### 8.2.2 Définition récurrente des listes d'éléments

Une liste d'éléments est un ensemble ordonné d'éléments, pas forcément du même type. Exemple `[1,2,3]` et `[2,1,3]` sont des listes différentes.
Par exemple un texte est une liste d'entier, correspondant au code ASCII des caractères : `"abs"=[97,98,99]`
Définitions récurrentes :

1. base : `[]`
   récurrence : si `L` est une liste alors `X.L` est une liste (ajout à gauche)
   ex : `[1,2,3]=1.(2.(3.[]))`
2. base : `[]`
   récurrence : si `L` est une liste alors `L.X` est une liste (ajout à droite)
3. base : `[]`, `[X]`, pour tout élément `X`.
   récurrence : si `L` est une liste alors `X.L.Y` est une liste (ajout à gauche et à droite)
   ex : `[1,2,3] = 1.[2].3`
   `[1,2,3,4] = 1.(2.[].4).3`

Dans Prolog c'est la première définition qui est utilisée. L'ajout d'un élément à gauche est noté `'.'(X,L)`, ou plus généralement `[X|L]`. L'ajout de plusieurs éléments se note `[X1,X2,X3|L]`.

### 8.2.3 Définition récurrente des entiers naturels

1. base : $0 \in \mathbb{N}$
   récurrence : $x \in \mathbb{N} \Rightarrow x + 1 \in \mathbb{N}$
2. base : $0 \in \mathbb{N}$
   récurrence : $x \in \mathbb{N} \Rightarrow 2x$ et $2x + 1 \in \mathbb{N}$
3. base : $\{0, 1\} \subseteq \mathbb{N}$
   récurrence : $\{x, y\} \subseteq \mathbb{N} \Rightarrow x + y \in \mathbb{N}$

Dans les définitions 1) et 2) il n'y a qu'une façon d'obtenir un entier donné :

$$5 = 2 * 2 + 1 = 2 * (2 * 1) + 1 = 2 * (2 * (2 * 0 + 1)) + 1$$

Dans la définition 3) il y en a plusieurs (un entier est la somme de plusieurs couples d'entiers).

## 8.3 Analyse récurrente d'un prédicat

On s'appuie sur la définition récurrente d'un ou plusieurs paramètre pour faire une analyse par cas. Ces différents cas conduisent à autant de règles Prolog. Pour les règles récursives, on utilise la définition récurrente des types des paramètres pour décomposer ces paramètres (l'appel récursif devant converger, c'est à dire que l'on fait appel à un élément du type qui permet de se rapprocher de la base).

### 8.3.1 Exemple sur les listes

la première définition récurrente des listes conduit à une analyse en deux cas (liste vide, liste non vide) et à une décomposition "premier/fin". La fin d'une liste correspondant à la liste privée de son premier élément.

1. longueur d'un liste d'entiers
   ```
   /* L : liste d'entiers, Y : entier
   longueur(?L,?Y) : Y est la longueur (nbre d'éléments) de L
   longueur(-L,-Y) est backtrackant */
   (1) longueur([],0).
   (2) longueur([_|L],Y) :- longueur(L,Y1), Y is Y1+1.
   ```
   Ce qui se lit : la longueur de la liste vide est zéro (base de la récurrence), soit `Y1` la longueur de `L` (hypothèse de récurrence), alors la longueur de `[_|L]` est `Y1+1`.
   Exemple d'utilisation : `longueur([1,2,3],Y).`

   Remarque : ce prédicat boucle si on demande une autre solution dans le cas `longueur(-L,+N)`. En exercice, trouvez pourquoi.
   En Prolog le prédicat longueur est `length/2`.
2. élément maximum d'une liste d'entiers
   ```
   /* L : liste d'entiers non vide, M : entier
   max(+L,?M) : M est l'élément maximum de la liste L*/
   (r1) max([X],X). /* liste à un seul élément */
   (r2) max([X,Y|L],X) :-max([Y|L],M),X>=M.
   (r3) max([X,Y|L],M) :-max([Y|L],M),M>X.
   ```
   Remarque : le prédicat `max_list/2` est prédéfini en Prolog.

3. comparaisons de deux listes
   ```
   /* L1,L2 : des listes d'éléments
   egalList(?L1,?L2) : L1 et L2 sont des listes identiques egalList(-L1,-L2) est
   backtrackant */
   ```
   remarques : ici le résultat étant booléen on n'utilise pas de paramètre supplémentaire le résultat étant donné par le succès ou l'échec du prédicat. dans ce cas l'analyse récurrente conduit à 4 cas (2 par paramètres).
   (r1) `egalList([],[]). /* cas de deux listes vides */`
   (r2) `egalList([],[X|L]) :- fail. /* on fait échouer le prédicat, en fait on peut omettre la règle ce qui aura le même effet */`
   (r3) `egalList([X|L],[]) :- fail. /* idem */`
   (r4) `egalList([X|L],[X1|L1]) :- egalList(L,L1) , X=X1.`
   où
   (r4bis) `egalList([X|L],[X|L1]) :- egalList(L,L1).`
   Remarque : ce prédicat s'écrit plus simplement `L1=L2`.

### 8.3.2 Exemple sur les entiers

1. un entier est-il pair ?
   avec la définition 1 des entiers
   ```
   /* pair(+X) : X est pair */
   ```
   (r1) `pair(0).`
   (r2) `pair(X) :- X>1,Y is X-2,pair(Y).`

2. multiplication égyptienne (dichotomique)
   avec la def 1 puis avec def 2 des entiers
   ```
   /* mult(+X,+Y,R) avec la def 1 */
   mult(_,0,0).
   mult(X,Y,R) :- Y > 0, Y1 is Y-1, mult(X,Y1,R1), R is R1+X.

   /* mult(+X,+Y,?R) avec la def 2 */
   mult(_,0,0).
   mult(X,Y,R) :- Y > 0, pair(Y), Y1 is Y//2,
                  mult(X,Y1,R1), R is R1*2.
   mult(X,Y,R) :- Y > 0, Y1 is Y-1, pair(Y1), Y2 is (Y-1)//2,
                  mult(X,Y2,R2), R is 2*R2+X.
   ```

En exercice :
- définir les programmes `mult(+X,-Y,+Z)` et `mult(-X,+Y,+Z)`
- arbre de résolution de `mult(3,5,R)` et `mult(3,5,10)`

### 8.3.3 Évaluation d'un nombre en base 2

```
/* B : une liste non vide d'entiers 0 ou 1. V : entier
valeur(+B,?V) <-> V est la valeur décimale du nombre binaire représenté par B */
```
exemple : `valeur([1,0,1],5)`

1. découpage à gauche
   on se base sur la relation suivante : si `R` est la valeur en base 10 de `B` alors $R+X*2^{longueur(B)}$ est la valeur de `[X|B]`
   ```
   valeur([X],X). /* par hypothèse X est soit 0 soit 1 */
   valeur([X|B],V) :- length(B,L), L>0, valeur(B,V1),
                      puiss(2,L,P), V is V1+X*P.
   ```

2. découpage à droite
   On utilise le schéma de Horner :

$$a_n2^n + a_{n-1}2^{n-1} + ... + a_22^2 + a_12^1 + a_0 = ((...(a_n * 2 + a_{n-1}) * 2 + ...a_2) * 2 + a_1) * 2 + a_0$$

Pour cela on suppose que la liste `B` est inversée, c'est à dire `B=[a0,...,an]`.Puis on utilise un découpage à gauche : sur notre liste : si `R` est la valeur en base 10 de `B` alors `R*2+X` est la valeur `[X|B]` d'où :

```
valeur2([X],X).
valeur2([X|B],R) :- valeur(B,R1),R is 2*R1+X.
```

On définit ensuite `valeur` en utilisant `reverse(B,B1)` qui est vrai si et seulement si `B1` est la liste obtenue en inversant la liste `B`.

```
valeur(B,V) :-reverse(B,B1),valeur(B1,V).
```

En exercice : écrire `puiss` (avec def1 et def2 des entiers)

# 9 Structures arborescentes

Intérêts des arbres : temps d'accès, structuration des données, hiérarchisation des données.
exemple : temps d'accès
- liste : $O(n)$
- arbre binaire : $O(log_2(n))$
- arbre $k$-aire : $O(log_k(n))$

## 9.1 Définition et terminologie

cf. livre “Cours d'informatique : langages et programmation ” PC Scholl & al.

### 9.1.1 Arbres $n$-aires, forêts

terminologie :

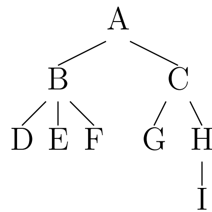
racine (nœud)-> foncteur
liste des fils -> arguments

Chaque fils est un arbre $n$-aire, un arbre est constitué de sous-arbres.

**Définition 13** *Une* **forêt** *est une liste d'arbres.*

### 9.1.2 Convention de représentation

- graphiques : patates (ensembles partition des fils), sous forme de graphe (nœuds, arcs)
  exemples avec :




  Attention : notion d'ordre (entre les fils) est importante $A(B, C) \neq A(C, B)$.
- textuelles :
  Par exemple avec des indentations

```
A
    B
        D
        E
        F
    C
        G
        H
            I
```

  Ou encore de façon parenthésée :
  - préfixe $A(B(D, E, F), C(G, H(I)))$ - (Prolog)

– postfixe : $((D, E, F)B, (G, (I)H)C)A$

#### 9.1.3 Arbres binaires

– Arbre pour lesquels chaque nœud a au plus 2 sous-arbres (fils).
– Notion de fils gauche , fils droit.

#### 9.1.4 Terminologie et définition (p96 du bouquin PCS)

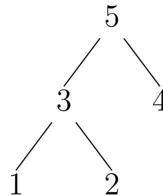
Terminologie des graphes
– **nœud** : élément d'un arbre
– **arc** : relie deux nœuds
Généalogie
– **relation** père/fils
– frère,cousin
Nature
– racine, feuille

**Définition 14** – **chemin (entre deux nœuds)** = *liste unique (si elle existe) des arcs (donc de nœuds) selon la relation père-fils pour aller de N1 à N2 (corollaire : N1 doit être un ascendant de N2)*
– **degré d'un nœud** *: arité = nb de fils ; degré de l'arbre = arité de la racine*
– **niveau d'un nœud** *: longueur du chemin de la racine à ce nœud (racine=niveau 1)*
– **profondeur d'un arbre** *: niveau maximal de l'arbre*
– **largeur d'un niveau** *: nb de nœuds de ce niveau*
– **largeur de l'arbre** *: largeur maximale des niveaux*

## 9.2 Arbres en Prolog

### 9.2.1 Arbre $n$-aire, forêts

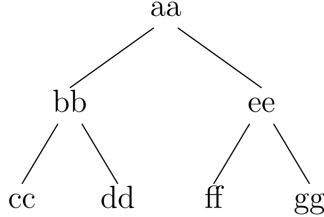
On utilise directement la notion d'arbre (termes) de Prolog.
– tout terme non atomique est un arbre
– `R(A,B,C)` (notation standard) ⇔ `[R,A,B,C]` (notation généralisée)
Une forêt est une liste d'arbres.
Contraintes sur les listes :
– `a=b` (unification)
– `a\=b` (non unification)
– `functor(A,R,N)` (N : arité de l'arbre A de racine R)
Remarque 1 : `functor(a,a,0)` ⇔ `atom(a)` ⇔ `a` est une feuille (arité 0).
Remarque 2 : Les arbres dont les nœuds sont des entiers ne peuvent pas être représentés sous la forme `r(f1,...,fn)` en gprolog puisque les nombres ne sont pas des atomes. Il faut utiliser la forme `[r,f1,...,fn]`. Par exemple `[5,[3,[1],[2]],[4]]` pour l'arbre :




Le prédicat `=..` permet de passer d'une représentation sous forme de terme à une représentation sous forme de liste. Le terme `T` = `r`(`f1`, ..., `fn`) est transformé en une liste `L` = [`r`, `f1`, ..., `fn`]. Ce prédicat est utile pour manipuler des termes dont on ne connaît pas la racine ou le nombre d'arguments.

### 9.2.2 Arbre binaires : les trois modèles

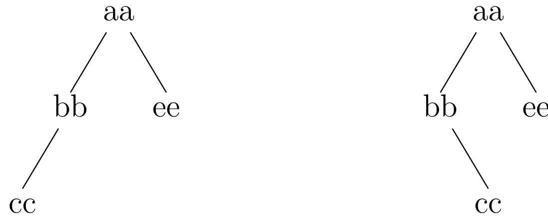
Un arbre binaire est tel que tous ses nœuds sont de degré = 2.
Exemple d'arbre binaire complet : $aa(bb(cc, dd), ee(ff, gg))$




- 7 nœuds
- arbre binaire complet de profondeur $n = 2n - 1$ nœuds

Arbre incomplet :

1. **Modèle 0 :** $aa(bb(cc), dd)$
   pb : $cc$ est-il fils gauche ou fils droit ?

   - si distinction inutile OK
   - sinon : modèle 1 ou 2 (cf ci-dessous)

   Décomposition récursive d'un arbre `A` en modèle 0 :
   - cas de base : feuille ; arbre `A` tel que `atom(A)`
   - cas récursif 1 : fils unique ; arbre `A` tel que `A=R(F)`
   - cas récursif 2 : 2 fils ; arbre `A` tel que `A=R(G,D)`

   Remarque : pas de notion d'arbre vide

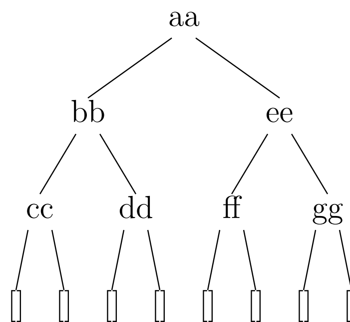
   Exemple : Nombre d'éléments

   ```
   /* nbElt(+A,?N) <-> N est le nombre d'éléments significatif de l'arbre binaire A
   */
   nbElt(A,1) :- atom(A).
   nbElt(A,N) :-A=..[_|[F]],nbElt(F,M), N is M+1.
   nbElt(A,N) :-A=..[_|[G,D]],nbElt(G,N1), nbElt(D,N2),N is N1+N2+1.
   ```

   Exemple d'appel :

   ```
   ?- nbElt(aa(bb(cc),dd),N).
   N=4
   ```

2. **Modèle 1**
   On complète l'arbre : tous les nœuds sont binaires, toutes les feuilles sont désignées par un élément particulier appelé l'arbre vide ([])
   $aa(bb(cc([], []), []), dd([], []))$ ou $aa(bb([], cc([], [])), dd([], []))$

   - inconvénient : beaucoup de feuilles inutiles, les "vraies" feuilles n'en sont pas (nœuds binaires)
   - avantage : décomposition récursive simple (pas de nœud unaire, toute feuille est l'arbre vide)

   Décomposition récursive d'un arbre A en modèle 1 :
   - cas de base : arbre vide `A=[]`
   - cas récursif : arbre binaire `A=R(G,D)`

   Exemple :

   ```
   nbElt([],0).
   nbElt(A,N) :- A=..[_|[G,D]],nbElt(G,N1), nbElt(D,N2),N is N1+N2+1.
   ```

Exemple d'appel :

```
?- nbrElt(aa(bb(cc([],[]),[]),dd([],[])),N) .
N=4
```

3. **Modèle 2**
   On complète uniquement les nœuds de degré 1 par l'arbre vide (pour lever l'ambiguïté fils gauche/fils droit)

   – avantage : pas de feuilles vide inutile
   – inconvénient : décomposition récursive moins simple

   Décomposition récursive d'un arbre A en modèle 2
   – cas de base : feuille non vide atom(A) ($A \neq []$ inutile par construction)
   – cas récursif 1 : nœud unaire gauche $A = R(G,[])$ , (par construction $G \neq []$)
   – cas récursif 2 : nœud unaire droit $A = R([],D)$, (par construction $D \neq []$)
   – cas récursif 3 : nœud binaire $A = R(G,D)$, $G \neq []$ et $D \neq []$

   Exemple :

```
nbrElt(A,1) :- atom(A).
nbrElt(A,N) :- A=..[_|[G,[]]],nbrElt(G,N1),N is N1+1.
nbrElt(A,N) :- A=..[_|[[],D]],nbrElt(D,N1),N is N1+1.
nbrElt(A,N) :-  A=..[_|[G,D]],G\=[],D\=[],
                nbrElt(G,N1), nbrElt(D,N2),N is N1+N2+1.
```

   Remarque : ce dernier programme peut s'écrire plus simplement :

```
nbrElt([],0). // fausse feuille
nbrElt(A,1) :- atom(A),A\=[]. // vraie feuille
nbrElt(A,N) :- A=..[_|[G,D]],nbrElt(G,N1), nbrElt(D,N2),N is N1+N2+1.
```

### 9.2.3 Traitement récursifs des arbres $n$-aires

Un arbre $n$-aire A=R(f1,...,fn) peut également s'écrire A=[R,f1,...,fn] où bien A=[R|[f1,...,fn]].
On en déduit les décompositions récursives suivantes d'un arbre $n$-aire A :
– cas de base : feuille : A est un atome.
– cas récursif : arbre général A=[R|F] où F est la liste des fils (forêt) non vide
  on applique une décomposition récursive à la liste des fils :
– sous cas 1 : A=[R|[f1]] (un seul fils) (rq : on peut directement écrire A=R(f1) )
– sous cas 2 : A=[R|[f1,f2|F]] (au moins deux fils )

## 9.3 Exemples

### 9.3.1 Arbre binaires

1. Nombre d'éléments de valeur e

```
/* nbe(+E,+A,?N) <-> N est le nombre d'occurrences de la valeur E dans un arbre
binaire A (modèle 1) */
nbe(_,[],0).
nbe(E,A,N) :- A=..[E|[G,D]],nbe(E,G,N1),nbe(E,D,N2),N is N1+N2+1.
nbe(E,A,N) :- A=..[X|[G,D]],X\=E,nbe(E,G,N1),nbe(E,D,N2),N is N1+N2.
```

2. Minimum d'un arbre binaire d'entiers
   Rappel : Les arbres dont les nœuds sont des entiers doivent être représentés sous la forme [R,f1,...,fn].

```
/* min(+A,?M) <-> M est la valeur minimum des éléments de l'arbre binaire A
(modèle 2) */
```

```
min([R],R).
min([R|[G,[]]],M) :- min(G,M1),min2(M1,R,M).
min([R|[[],D]],M) :- min(D,M1),min2(M1,R,M).
min([R|[G,D]],M) :- G\=[],D\=[],min(G,M1),min(D,M2),
                    min2(M1,M2,M3),min2(M3,R,M).
```

avec

```
min2(A,B,A) :- A=<B.
min2(A,B,B) :- B<A.
```

### 9.3.2 Arbre $n$-aires

Nombre d'éléments d'un arbre $n$-aire

`/* nbrEltNaire(+A,N) <-> N est le nombre d'éléments de l'arbre n-aire A */`

version 1 : introduction d'un prédicat intermédiaire

```
nbrEltNaire(A,1) :- atom(A) . // A est une feuille
nbrEltNaire(A,N) :- A=..[_|F],F\=[],nbrEltForet(F,N1),N is N1+1. // F est la forêt des fils
/* nbrEltForet(+F,N) <-> N est le nombre d'éléments total de tous les arbres de la
forêt F */
nbrEltForet([],0).
nbrEltForet([F1|F],N) :- nbrEltForet(F,N1),nbrEltNaire(F1,N2),N is N1+N2.
```

rem : récursivité croisée dont la convergence est assurée par les propriétés suivantes :

- `nbrEltNaire` "descend" d'un niveau à chaque appel (donc converge vers les feuilles)
- `nbrEltForet` parcourt la liste des fils de chaque racine donc converge vers []

version 2 : sans prédicat intermédiaire

```
nbrEltNaire(A,1) :- atom(A) .
nbrEltNaire(A,N) :- A=..[R|[F1|F]],T=[R|F],nbrEltNaire(T,N1),
                    nbrEltNaire(F1,N2),N is N1+N2. // la racine sera comptée à la fin
```

version 3 : traitement d'une forêt

```
nbrEltForet([],0).
nbrEltForet([A|L],N) :-A=..[_|F],nbrEltForet(F,N1),nbrEltForet(L,N2),N is N1+N2+1.
```

d'où :

```
nbrEltNaire(A,N) :- nbrEltForet([A],N). // forêt réduite à A
```

### 9.3.3 Arbre de recherche binaire (ARB)

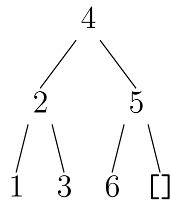
**Définition 15** *Un ARB (BST : Binary Search Tree en anglais) est un arbre binaire* A *tel que : pour tout noeud* `r` *de* `A`*, pour tout noeud* `g` *appartenant au fils gauche de* `r` *et pour tout noeud* `d` *appartenant au fils droit de* `r` *:*

$$\mathtt{g} \leq \mathtt{r} < \mathtt{d}$$

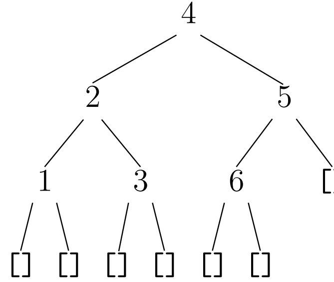
**définition récurrente :** `[]` est un ARB , `[X]` est un ARB (une feuille est un ARB)
`[R|[G,D]]` est un ARB ssi

- `G` est un ARB
- `D` est un ARB
- `max(G)` $\leq$ `R` (si `G` $\neq$ `[]`)
- `R` $<$ `min(D)` (si `D` $\neq$ `[]`)

**représentation en Prolog :** Il faut un modèle dans lequel les arbres unaires gauches et droit peuvent se distinguer : modèle 1 ou 2 mais pas modèle 0.
exemple d'arbre :




modèle 2 :
`[4,[2,[1],[3]],[6,[5],[]]]`




modèle 1 :
`[4,[2,[1,[],[]],[3,[],[]]],[6,[5,[],[]],[]]]`

**traitements sur les ARB :** Intérêt : structurer une collection d'information ordonnée pour optimiser les temps d'accès.

Exemple1 : minimum d'un ARB

1. Rappel arbre non ARB (modèle 2) 9.3.1 c) ci-dessus
2. ARB
```
min([A],A).
min([R,[],_],R).
min([_,G,_],M) :- G\=[],min(G,M).
```

Comparaison de 1) et 2) :

1. parcours tout l'arbre pour trouver le minimum : soit $n$ le nombre d'élément : algorithme en O($n$)
2. parcours qu'une branche, au maximum algo en ordre de la profondeur de l'arbre. Si l'arbre est complet $O(log_2(n+1))$.

rq : 1000 éléments ⇒ 1) mille appels récursifs. 2) < 10 appels récursifs ($1024 = 2^{10}$)

Exemple2 : appartenance

1. arbre non ARB (modèle 1)
```
/* E : un élément possédant un relation d'ordre, A un arbre binaire
app(?E,+A) <=> E est un nœud de l'arbre A */
(r1) /* app(E,[]) :- fail. */
(r2) app(E,[E]).
(r3) app(E,[E,_,_]).
(r4) app(E,[R,G,_]) :- app(E,G),E\=R.
(r5) app(E,[R,_,D]) :- app(E,D),E\=R.
```
Rq : Dans les règles r4 et r5, le test entre `E` et `R` est après l'appel de `app` afin de garantir que `E` a une valeur sinon `E\=R` échoue chaque fois que `E` est une variable.
2. ARB
```
(1) app(E,[E]).
(2) app(E,[E,_,_]).
(3) app(E,[R,G,_]) :- E < R,app(E,G).
(4) app(E,[R,_,D]) :- R < E,app(E,D).
```

Autres opérations : ajout, séquence extraite triée (parcours infixe), problème de la suppression, tri en arbre.

# 10 Contraintes sur les domaines finis

## 10.1 Introduction

**A quoi sert la programmation avec contraintes** La programmation avec des contraintes est une technique logicielle en développement pour la description et la résolution de problèmes importants, particulièrement combinatoires dans les domaines de la planification et de l'ordonnancement.

Elle est employée avec succès dans les domaines suivants
– applications graphiques, pour exprimer des relations géométriques entre divers éléments d'une scène
– traitement des langues naturelles (pour améliorer la performance des analyseurs)
– système de base de données (pour assurer ou restaurer la cohérence des données)

– problèmes de recherche opérationnelle (en particulier les problèmes d'optimisation)
et bien d'autres domaines non cités.

**Prolog et les contraintes** Le langage Prolog est né en 1970. Son apogée a eu lieu dans les années 80, en liaison avec des applications au traitement des langues naturelles et à l'intelligence artificielle. Son déclin provisoire est suivi d'une résurrection avec l'introduction des *contraintes* dans le langage.

Une *contrainte* c'est une relation entre des variables, chacune prenant une valeur dans un *domaine donné*. Une contrainte restreint les valeurs possibles que les variables peuvent prendre.

Donnons un exemple tiré de gprolog. La contrainte `fd_domain([X,Y,Z],1,10)` impose que ces trois variables prennent des valeurs entières entre 1 et 10, la conjonction de cette première contrainte et de la contrainte `X+Y #= Z` restreint les domaines de X et Y à l'intervalle fermé 1..9 (car X est au plus égal au maximum de Z moins le minimum de Y) et le domaine de Z à l'intervalle fermé 2..10 (car le minimum de Z est au moins la somme des minimums de X et de Y).

Prolog et les contraintes ouvre le nouveau domaine de la programmation logique avec contraintes (PLC) : Prolog est étendu avec des résolveurs de contraintes, travaillant sur des données qui peuvent être des entiers, des booléens, *des domaines finis d'entiers*, des rationnels, des intervalles de rationnels.

Les résolveurs de contraintes peuvent être ajoutés à d'autres langages que Prolog, mais l'algorithme d'unification de Prolog est déjà un résolveur des contraintes d'égalité sur les termes, donc la résolution de contraintes s'intègre naturellement à Prolog.

## 10.2 Classification des contraintes

### 10.2.1 Définition des domaines

`fd_domain(LVars,Bi,Bs)` Lvars est une liste de variables, la réussite de la requête lie chacune des variables de la liste à l'intervalle Bi..Bs bornes comprises. Bi et Bs doivent être des entiers.

```
?- fd_domain([X, Y],1, 4).
X = _#3(1..4)
Y = _#25(1..4)
```

La réponse doit être interprétée ainsi : `X` et `Y` sont des variables dont la valeur est comprise entre 1 et 4.

### 10.2.2 Contraintes arithmétiques

**Principe des réductions de domaine** Commençons par des exemples.

```
?- fd_domain([X,Y],0,7),X * Y #= 6, X + Y #=5, X #< Y.
X = 2 Y = 3
```

Le domaine initial des variables X et Y est fixé, puis il est réduit au fur et à mesure de l'accumulation des contraintes. L'ordre des contraintes est indifférent.

On n'indique pas l'algorithme qui réduit les domaines, mais on donne des indications sur son principe de fonctionnement.

1. D'après la première contrainte X et Y sont entre 0 et 7.
2. D'après la deuxième contrainte X et Y ne sont pas nuls donc valent au moins 1, d'après cette deuxième contrainte X et Y valent au plus 6.
3. D'après la troisième contrainte et les bornes précédentes, X vaut au plus 5 moins la valeur minimum de Y, donc au plus 4, et de même pour Y.
4. *Réutilisons* à nouveau la deuxième contrainte, X est au moins égal à 6/4, donc X vaut au moins 2. *Réutilisons* à nouveau la troisième contrainte, puisque X vaut au moins 2, Y vaut au plus 3. En échangeant les rôles de X et Y, on voit qu'après les 3 premières contraintes X et Y sont entre 2 et 3. Donc la dernière contrainte donne la solution

En présence d'un ensemble de contraintes, l'algorithme utilise chaque contrainte en effectuant les réductions de domaine, jusqu'au moment où plus *aucune* contrainte ne peut réduire le domaine des variables.

**Réduction partielle des domaines** Le signe caractéristique des contraintes à domaine fini est le `#`.
La liste des prédicats utilisables est : `#=, #\=, #<, #=<, #>, #>=`

Ces prédicats peuvent être écrits entre leurs opérandes qui sont des expressions arithmétiques et nous avons déjà vus plusieurs exemples de leurs utilisations.

On en présente un exemple surprenant qui met en évidence que gprolog utilise deux représentations des domaines finis.

```
?- fd_domain([X],1,1000), X#\=3.
X = _#3(1..2:4..127@)
```

Après la première contrainte, la valeur de X est comprise entre 1 et 1000. Cet intervalle est représenté par un couple d'entiers. Après la deuxième contrainte, la valeur de X doit être différente de 3. Gprolog représente alors X par un vecteur de 128 bits. Le signe `@` indique que des valeurs ont été perdues suite à ce changement de représentation.

En résumé, il y a deux représentations des domaines finis

1. par un intervalle entre Bi et Bs avec $0 \leq Bi$ et $Bs \leq 2^{28} - 1$
2. par un vecteur de bits d'indice entre 0 et une valeur vector_max. Initialement vector_max = 127. Il est possible par la contrainte fd_set_vector_max(+integer) de changer la valeur de vector_max, ce qui est le premier moyen d'éviter de perdre des valeurs.

Comment chaque contrainte est-elle utilisée ? Soit X une variable de domaine DX, Y une variable de domaine DY, l'exécution de la contrainte r(X,Y) *peut* enlever de DX les valeurs v telles que r(v,Y) n'est vérifiée pour aucune valeur de Y dans le domaine DY et peut réduire de même le domaine DY. Par exemple :

```
?- fd_domain([X,Y],1,10), X#\=2, X+Y#=5.
X = _#3(1:3..4)
Y = _#25(1..4)
```

Noter que la réduction opérée par la dernière contrainte est *partielle*. Puisque X est différent de 2, d'après la dernière contrainte, Y ne peut pas être égal à 3, mais cette valeur n'est pas enlevée du domaine de Y.

Les contraintes de réductions partielles (avec un seul `#`) ne permettent que la réduction des intervalles entre les valeurs minimum et maximum des variables.

**Réduction complète des domaines** La liste des prédicats utilisables est : `#=#, #\=#, #<#, #=<#, #>#, #>=#`

Reprenons l'exemple précédent avec la réduction complète :

```
?- fd_domain([X,Y],1,10), X#\=2, X+Y#=#5.
X = _#3(1:3..4)
Y = _#25(1..2:4)
```

Comparons sur d'autres exemples la réduction partielle et complète

```
| ?- fd_domain([X,Y],1,100),X*Y #= 51.
X = _#3(1..51)
Y = _#25(1..51)
yes
| ?- fd_domain([X,Y],1,100),X*Y #=# 51.
X = _#3(1:3:17:51)
Y = _#25(1:3:17:51)
yes
```

La réduction complète prend plus de temps que la réduction partielle. Si votre problème n'a besoin que d'intervalles, il est inutile de l'utiliser.

### 10.2.3 Information sur les domaines

#### Modèles des prédicats

```
fd_min(+fd_variable, ?integer)
fd_max(+fd_variable, ?integer)
fd_size(+fd_variable, ?integer)
fd_dom(+fd_variable, ?integer_list)
```

#### Description

1. `fd_min(X, N)` réussit si `N` est la valeur minimale du domaine de `X`

2. `fd_max(X, N)` réussit si `N` est la valeur maximale du domaine de `X`
3. `fd_size(X, N)` réussit si `N` est le nombre d'éléments du domaine actuel de `X`
4. `fd_dom(X, Values)` réussit si `Values` est la liste des valeurs du domaine actuel de `X`

### 10.2.4 Contraintes d'énumération

**Modèles des prédicats**

```
fd_labeling(+fd_variable_list)
fd_labelingff(+fd_variable_list)
```

**Description**

1. `fd_labeling(Vars)` affecte une valeur à chaque variable de la liste Vars.

```
?- fd_domain([X],1,2),fd_labeling([X]).
X = 1 ?;
X = 2
yes
```

2. `fd_labelingff(Vars)` affecte une valeur à chaque variable de la liste Vars en choisissant d'abord la variable de plus petit domaine et entre deux variables de même plus petit domaine, celle la plus à gauche.

```
?- fd_domain([X],1,3), fd_domain([Y],1,2),fd_labelingff([X, Y]).
X = 1 Y = 1 ? a
X = 2 Y = 1
X = 3 Y = 1
X = 1 Y = 2
X = 2 Y = 2
X = 3 Y = 2
```

Attention la variable `Y`, qui a le plus petit domaine, est bien énuméré d'abord : pour `Y = 1`, `X` prend les valeurs `1, 2, 3` puis on recommence avec `Y = 2`.

Le prédicat `fd_labeling` peut être utilisé avec des options qui modifient l'ordre d'énumération. L'ordre d'énumération est très important dans la résolution des problèmes combinatoires, puisque, par exemple, le choix d'une valeur pour une variable qui entre dans beaucoup de contraintes, va conduire plus vite à l'échec ou au succès des contraintes comportant cette variable.

Nous renvoyons à la documentation gprolog pour compléter les informations sur les contraintes. En particulier nous n'avons pas évoqué ci-dessus les contraintes dites symboliques (comme `fd_all_different` qui figure dans les exemples) qui posent des contraintes sur des listes de variables.

## 10.3 Exemples d'utilisations

## 10.4 Générateur d'entiers

Une application des contraintes en domaine fini parmi les plus classique est la génération d'entiers dans un intervalle par backtrack :

```
/* enum(-V,+N1,+N2) <-> réussit si V prend une valeur entre N1 et N2 */
enum(X,N1,N2) :-fd_domain(X,N1,N2),fd_labeling(X).
```

### 10.4.1 Résolution d'équations linéaires

Il est également possible de résoudre des équations linéaires. Par exemple :

```
?- X+Y#=2,X+2*Y#=3.
X=1
Y=1
yes
```

Il est parfois nécessaire de forcer l'énumération des solutions :

```
?- X+Y+Z#=6,X+2*Y+Z#=8,X+Y+2*Z#=9,fd\_labeling([X,Y,Z]).
X=1
Y=2
```

```
Z=3 ?;
no
```

### 10.4.2 La factorielle (le retour)

L'utilisation de contraintes sur les domaines finis permet de faire une version de la factorielle qui est inversible, c'est à dire qui peut être appelé en mode `fac(-N,+R)`.

```
fac(0,1).
fac(N,R) :- 0#<N,N#=<R,N1#=N-1,R#=R1*N,fac(N1,R1).
```

### 10.4.3 SEND+MORE=MONEY

Cet exemple est typique de programme Prolog dont la définition est très proche de l'expression formelle. On veut résoudre le système d'équations représenté par "SEND+MORE=MONEY". Il s'agit d'associer aux différents caractères des chiffres tels que l'opération arithmétique correspondante soit correcte. La règle impose qu'une lettre ne soit associée qu'à un seul chiffre, que deux lettres différentes soient associées à deux chiffres différents et que toutes les lettres soient associées. On peut tout d'abord exprimer les premières propriétés énoncées. Nous nommerons le prédicat `sendmory` composé de l'ensemble des variables de ce système d'équations. On sait déjà que tous les éléments de la solution sont différents deux à deux (prédicat `fd_all_different`) et que tous les éléments sont des chiffres (prédicat `fd_domain`). Le système de contraintes du prédicat équivaut au système d'équations que l'on cherche à résoudre et qui correspond au principe de l'addition en base 10.

```
sendmory([S,E,N,D,M,O,R,Y]) :- fd_domain([S,E,N,D,M,O,R,Y],0,9),
                               fd_all_different([S,E,N,D,M,O,R,Y]),
                               equationValide([S,E,N,D,M,O,R,Y]),
                               fd_labeling([S,E,N,D,M,O,R,Y]).

equationValide([S,E,N,D,M,O,R,Y]) :- M=1, /* par définition */
                                     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 #= More+Send.
```

Remarques :
- Le prédicat sendmory trouve la solution [S,E,N,D,M,O,R,Y] = [9,5,6,7,1,0,8,2] en 3s de CPU environ.
- l'arbre de résolution principal est basé sur 10 possibilités pour 8 variables (810 = 1073741824). Cependant toutes ne sont pas explorées car toutes les variables doivent être différentes et le système de contraintes limite l'exploration de certains sous-ensembles de combinaisons.

### 10.4.4 Le carré magique

Un carré magique est une matrice 3x3 d'entiers telle que les sommes des lignes, colonnes et diagonales soient égales.

```
/* carremagique(Carre,Somme) : Carre est un carre magique,
Somme est la valeur de chaque ligne, colonne et diagonale */
carremagique(Carre,Somme) :- /* Contraintes */
                             Carre=[C11,C12,C13,C21,C22,C23,C31,C32,C33],
                             fd_domain(Carre,1,9),fd_all_different(Carre),
                             Somme#=C11+C12+C13,
                             Somme#=C21+C22+C23,
                             Somme#=C31+C32+C33,
                             Somme#=C11+C21+C31,
                             Somme#=C12+C22+C32,
                             Somme#=C13+C23+C33,
                             Somme#=C11+C22+C33,
                             Somme#=C13+C22+C31,
                             /* Génération */
                             fd_labeling(Carre),
                             /* Affichage */
```

```
                              nl,outcarre(Carre,1,3).
/* outcarre(C,R,T) : affiche la liste C en allant à la ligne lorsque R=T */
outcarre([],_,_).
outcarre([E|S],Taille,Taille) :- write(E),nl,outcarre(S,1,Taille).
outcarre([E|S],Rang,Taille) :- Rang<Taille,write(E),Rang1 is Rang+1,
                               outcarre(S,Rang1,Taille).
```

Troisième partie

# Aspects techniques

## 11 Un outil de contrôle : le coupe-choix

### 11.1 Problématique

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. La combinaison des ces deux opérateurs permet d'obtenir des structures classiques. On doit tout de même remarquer que l'introduction de la notion de contraintes dans un langage comme Prolog est un outil qui permet de réaliser un contrôle sur le déroulement de la résolution (exclusivité des clauses en particulier).

Exemple : on désire réaliser la structure de contrôle de choix exclusif (partitionnement de l'ensemble des possibilités pour les conditions). Cette structure de contrôle doit permettre de se protéger des backtrack intempestifs.

**solution 1 : exprimer le choix dans la tête de clause**
Exemple avec `pair` :

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

`pair(0)` et `pair(succ(succ(N)))` ne sont pas unifiables, ils forment une disjonction exclusive de l'ensemble des termes {`pair(X)`}. Soit $P$, le but à résoudre, la résolution revient à exécuter l'algo suivant :
CHOIX
$P$ s'unifie à `pair(0)` : vrai
$P$ s'unifie à `pair(succ(succ(N)))` : résoudre `pair(N)`
FINCHOIX
Le backtrack est de facto impossible car un terme qui s'unifie à la première clause ne pourra s'unifier à la seconde.

**solution 2 : exprimer le choix dans les contraintes de la clause**

```
        appartient(X,[Y|L]) :- X=Y.
appartient(X,[Y|L]) :- X\=Y,appartient(X,L).
```

Le choix est assuré par l'exclusivité des contraintes. Le backtrack entre les deux clause est possible (unification), mais le système échouera sur la contrainte. On peut aussi combiner cette solution avec la solution précédente.

```
        appartient(X,[X|L]).
appartient(X,[Y|L]) :- appartient(X,L).
```

**solution 3 : exprimer le choix dans les buts de la queue de clause**

```
        union([],E2,E2).
union([X|E1],E2,E3) :- appartient(X,E2),union(E1,E2,E3).
union([X|E1],E2,[X|E3]) :- not_appartient(X,E2),union(E1,E2,E3).
```

avec :

```
        not_appartient(X,[]).
not_appartient(X,[Y|R]) :- X\=Y,not_appartient(X,R).
```

Le choix est assuré par l'exclusivité des buts (à la charge du programmeur). Attention : le but déterminant doit être placé en tête de la queue de clause.
Inconvénient : les prédicats `appartient` et `not_appartient` sont exécutés systématiquement pour une liste initial non vide (redondance).

**solution 4 : interdire le backtrack quand une clause a été choisie** Pour cela on utilise un prédicat particulier appelé "coupe-choix" ou "coupure" ou encore "cut" noté " `!` ".

```
/* N entier positif ou nul, fact(+N,?R) <-> R est la factorielle de N */
fact(0,1) :-!.
fact(N,R) :- N>0,N1 is N-1,fact(N1,R1),R is R1*N.
```

Le backtrack, théoriquement possible, est *inhibé* par le coupe-choix. Cela revient à la sémantique suivante :

**si** $P$ s'unifie à `fact(0,1)`
**alors** vrai ( et c'est fini)
**sinon si** $P$ s'unifie à `fact(N,R)`
    **alors** `N>0,N1 is N-1,fact(N1,R1),R is R1*N`
    **sinon** échec

## 11.2 Définition du coupe-choix

Le coupe-choix est un prédicat prédéfini qui réussit toujours. Il a cependant un effet de bord en intervenant sur la résolution elle-même.

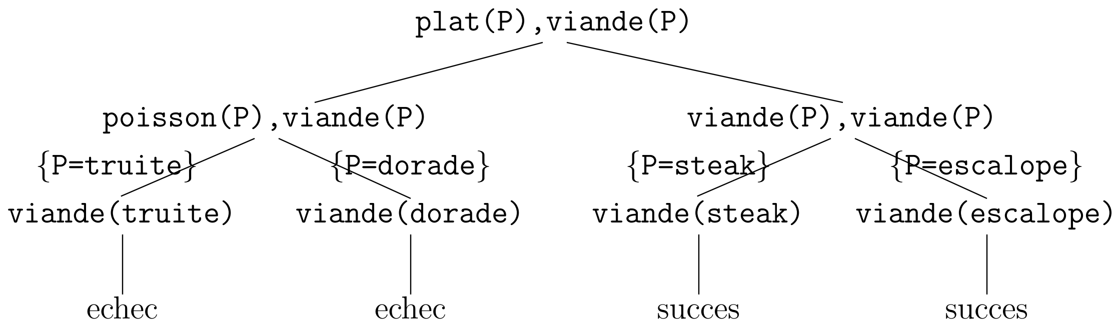
On peut comparer le coupe-choix à une porte dotée d'un mécanisme de tringlerie qui fait que lorsqu'on franchit la porte[2] cela ferme celles qui sont accrochées à la tringle supprimant ainsi des chemins d'exploration. La tringle est accrochée aux portes franchies antérieurement. Pour mieux comprendre son fonctionnement nous étudions son effet quand il apparaît dans une question puis dans une règle.

1. **Le coupe choix dans une question**
   Soit la question : `C,!,D.` avec `C= c1,c2,...,cn` et D=d1,d2,...,dk
   – Prolog résout d'abord les buts de `C`, en utilisant le mécanisme de backtrack si besoin,
   – dès qu'une solution est trouvée pour `C`, le coupe-choix est résolu et coupe tous les choix pendants de la résolution de `C`,
   – `D` est ensuite résolu,
   – si l'un des buts de `D` échoue ou si `D` réussit, le backtrack sur les buts de `D` est lancé,
   – quand aucun backtrack n'est plus possible sur les buts `d1,d2,...,dk`, le coupe-choix empêche le backtrack dans les buts de `C`[3]. On ne peut plus trouver de solutions.
2. **Le coupe choix dans une clause**
   On peut insérer un coupe-choix dans une clause. La sémantique est la même pour les buts de la queue de clause que la sémantique d'une question avec en plus les deux règles suivantes :
   le coupe-choix COUPE LES CHOIX relatifs aux alternatives de la clause en cours
   (1) `C :- C1,!,C2.`
   (2) `C :- C3.`
   Si `C1` est résolu, le `!` coupe l'alternative restante pour `C` (2), ainsi bien sur que tous les autres choix possible pour la résolution de `C1`.
   Si `C1` n'est pas résolu, le `!` n'est pas effacé, le backtrack est possible sur (2).
   Le coupe-choix N'AFFECTE PAS les résolutions englobant la clause le contenant
   ```
   C :- C1,C2.
   C1 :- D1.
   C1 :- D2.
   C2 :- C3,!,C4.
   C2 :- D3.
   ```
   le coupe choix empêche un backtrack sur `C3` et la $2^{eme}$ règle de `C2` mais pas sur `C1`
3. **Exemple** Reprenons un sous-ensemble des règles de l'exemple du menu qui concerne la constitution du plat principal d'un repas. Nous avons les clauses liées au plat et celles liées aux viandes et aux poissons.
   ```
   poisson(truite).
   poisson(daurade).
   viande(steak).
   viande(escalope).
   ```

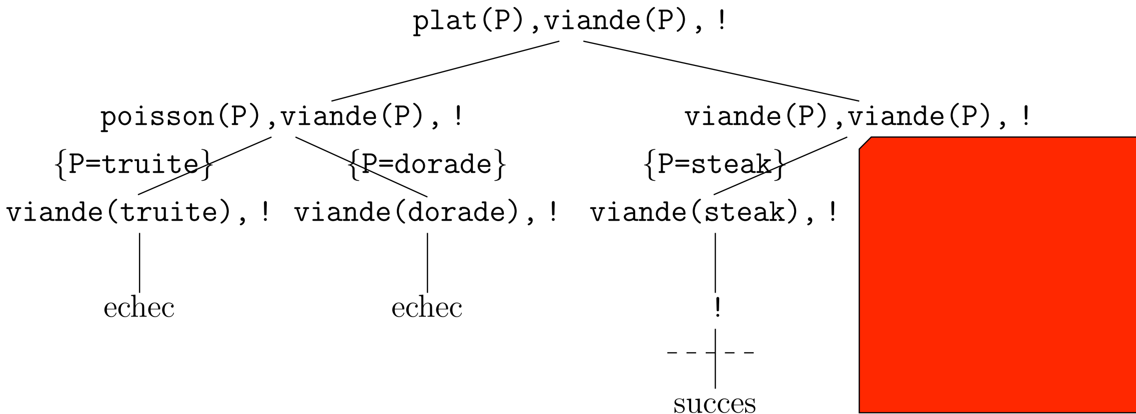
[2]On peut schématiser l'effacement d'un terme dans une résolvante comme le franchissement d'une étape de résolution représentée par une porte.
[3]Le coupe-choix est un prédicat à effet de bord puisque son franchissement en sens inverse ne restitue pas la situation antérieure à son franchissement initial.

```
plat(P) :- poisson(P).
plat(V) :- viande(V).
```

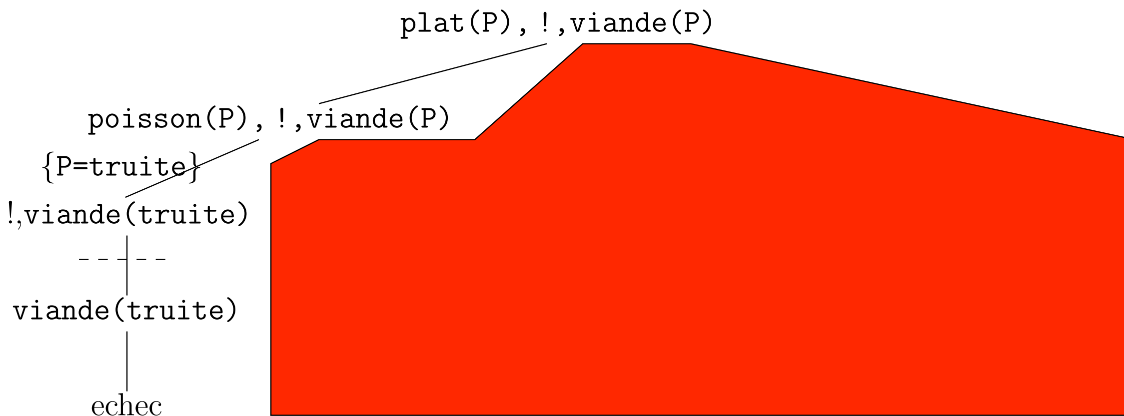
Dessinons l'arbre de résolution de la question `plat(P),viande(P)` qui fournit en réponse tous les plats qui sont de la viande.




Dessinons l'arbre de résolution de la question `plat(P),viande(P),!` qui fournit en réponse le 1er plat qui est de la viande.




Dessinons l'arbre de résolution de la question `plat(P),!,viande(P)` qui ne fournit aucune réponse.




## 11.3 Utilisations

Le coupe-choix doit être utilisé avec précaution du fait de son effet sur la résolution. Les cas d'utilisation sont bien déterminés et devront être observés. En particulier il est souvent incompatible avec les prédicats fonctionnant en mode génération.

### 11.3.1 Contrôle des solutions dans un processus de génération

Comme nous l'avons vu ci-dessus, le cut permet d'affiner une question.
Exemple :
- `repas(E,P,D),poisson(P).` /* quels sont les repas comprenant du poisson ? */
- `poisson(P),repas(E,P,D).` /* idem */

– `repas(E,P,D),!,poisson(P).` /* le premier repas de la base comprend-il du poisson ? */
– `poisson(P),!,repas(E,P,D).` /* construire un repas avec le premier poisson de la base */
– `repas(E,P,D),poisson(P),!.` /* quel est le premier repas contenant du poisson ? */
– `poisson(P),repas(E,P,D),!.` /* idem */
– `!,repas(E,P,D),poisson(P).` /* idem sans cut */

De façon générale, étant donné que PROLOG rend toutes les solutions à la question posée, on pourra lorsque l'on ne désire qu'une seule réponse ajouter un cut à la question :

```
q(X1,X2,...,Xn)
```

⇒ "Quel est l'ensemble des toutes substitutions solutions de la question Q"

```
q(X1,X2,...,Xn) , !
```

⇒ "Quelle est la première substitution solution de la question Q", "première" au sens de la sémantique procédurale de Prolog.

### 11.3.2 Confirmation du choix d'une règle

Comme nous l'avons vu ci-dessus, le coupe-choix permet de rendre des règles exclusives. On insère le ! après la séquence de but qui permet de déterminer que la règle choisie est la bonne.

```
union([],E2,E2) :- !.
union([X|E1],E2,E3) :- appartient(X,E2),!,union(E1,E2,E3).
union([X|E1],E2,[X|E3]) :- union(E1,E2,E3).
```

Attention : ce programme n'est pas strictement équivalent au premier. Considérons la question : `union(E,[1,2],[1,2])`. Le premier programme qui ne comporte pas de coupe-choix répond :

```
?- union(E,[1,2],[1,2]).
E=[] ? ;
E=[1] ? ;
E=[1,1] ?
yes
```

Le second programme répond :

```
?- union(E,[1,2],[1,2]).
E=[] ? ;
no
```

Le premier programme ne termine pas car il ne précise pas qu'il faut que les listes ne contiennent plusieurs fois les mêmes éléments.

Un prédicat contenant des coupe-choix a un comportement en mode génération différent de la version sans coupe-choix. En général l'introduction d'un coupe-choix fait perdre le caractère relationnel du prédicat.

remarque : la règle : `union([X|E1],E2,E3) :- !,appartient(X,E2),union(E1,E2,E3)` conduit à une erreur, le cut placé avant le prédicat déterminant le choix de la bonne règle conduira à ne jamais utiliser la règle 3.

Autre exemple :

Exemple : prédicat appartient avec et sans cut :

```
/* version avec exclusivité assurée par des contraintes */
appartient(X,[X|_]).
appartient(X,[Y|L]) :- appartient(X,L),X\= Y.
?- appartient(X,[1,2,3]).
X=1 ? ;
X=2 ? ;
X=3 ?
yes
/* version avec exclusivité assurée par le cut */
appartient(X,[X|_]) :- !.
appartient(X,[_|L]) :-appartient(X,L).
?- appartient(X,[1,2,3])
X=1 ? ;
no
```

### 11.3.3 Expression de nouvelles structures de contrôle

**La négation par coupe-choix**

Soit le prédicat `voyelle(L)` définit les les voyelles de l'alphabet :

```
voyelle('a').
voyelle('e').
voyelle('i').
voyelle('o').
voyelle('u').
voyelle('y').
```

On désire écrire le `consonne(L)` qui définit les consonnes de l'alphabet. On peut le faire comme pour les voyelles par une définition en extension. Ceci est cependant assez fastidieux. On peut remarquer que les sous-ensembles des voyelles et des consonnes forment une partition de l'ensemble des caractères de l'alphabet permettant de définir l'un comme étant le complémentaire de l'autre. Nous allons utiliser le fait qu'une consonne est un caractère qui n'est pas une voyelle pour définir l'ensemble de consonne en compréhension.

```
consonne(L) :- voyelle(L),!,fail.
consonne(L) :- lettre(L).
```

avec :

```
lettre(L) :-char_code(L,X),X>96,X<123.
```

De façon générale le prédicat de négation de but `not` s'écrira :

```
not(P) :- P,!,fail.
not(_).
```

Exemple d'utilisation : `consonne(L) :- not(voyelle(L)), lettre(L).`

Attention : L'utilisation d'un tel prédicat avec des variables libres peut conduire à des résultats inattendus. Ceci est dû à ce que l'on considère que la non résolution est une négation. Ceci n'est vrai que lorsqu'on travaille sur des variables instanciées ou avec un système de contraintes.

```
?- consonne(X).
no
?- consonne('z').
yes
```

On remarque que tel qu'il est défini le prédicat consonne n'est pas générateur. Pour le rendre générateur et par la même résoudre le problème ci-dessus, nous écrivons une nouvelle version du prédicat `consonne`.

```
consonne(L1) :- char_code('a',A),char_code('z',Z),fd_domain(L,A,Z),
                fd_labeling(L),char_code(L1,L),not(voyelle(L1)).
```

**Écriture du si-alors-sinon**

Le coupe-choix peut servir pour construire les structures contrôle classique des langages traditionnels (conditionnelle,...).

```
si(_condition,_alors,_sinon) :- _condition,!,_alors.
si(_condition,_alors,_sinon) :- _sinon.
```

Exemple : "si x est carnivore alors x mange de la viande, sinon x rumine"

```
        si(carnivore(x),mange_viande(x), rumine(x))
```

ou

```
        union([],E2,E2) :-!.
        union([X|E1],E2,U) :- union(E1,E2,E3),si(appartient(X,E2),U=E3,U=[X|E3]).
```

**Structure de choix (selon)**

choix

$Cond_1 : But_1$
$Cond_2 : But_2$
...
$Cond_n : But_n$

finchoix

En Prolog :

```
choix([[Cond1,But1],[Cond2,But2],...,[Condn,Butn]]).
```

avec :

```
choix([[C,B]|_]) :- C,!,B. /* si la première condition est vraie, on résout le but associé */
choix([_|Reste]) :- choix(Reste). /* sinon on recommence avec la deuxième */
/* remarque : si aucune condition n'est vraie, le choix échoue */
```

Exercice : version avec cas balai (else ou sinon)

Remarque : Gprolog possède des prédicats capables de reproduire ces structures. Le prédicat `C1 :- C2` correspond à *si* `C1` *alors* `C2`. En utilisant " ;", qui correspond au "ou" logique, nous pouvons écrire `C1 :- C2 ; C3` qui signifie *si* `C1` *alors* `C2` *sinon* `C3`.

**Structure itérative : tantque Cond faire But**

En Prolog : `tantque(Cond,But)`

```
tantque(C,B) :- C,!,B,tantque(C,B).
tantque(_,_).
```

Attention : il faut que la condition `C` puisse s'évaluer à faux (donc dépend du contexte du programme - effet de bord)

# 12 Quelques prédicats prédéfinis de gprolog

## 12.1 Entrées-sorties

Gprolog possède plusieurs prédicats permettant l'interaction avec l'utilisateur au cours de la résolution.

Le prédicat `write/1` permet d'afficher un terme à l'écran pendant une résolution.

```
?- write(bonjour).
bonjour
yes
?- write(vive(prolog)).
vive(prolog)
yes
?- write(X).
_16
yes
?- X=3,write(X+Y).
3+_20
yes
?- write('Votez pour moi').
Votez pour moi !
yes
```

Le prédicat `nl/0` permet d'aller à la ligne.

```
?- write(bonjour),nl,write('à tous').
bonjour
à tous
yes
```

Le prédicat `read/1` permet de lire des termes à l'écran. La lecture s'arrête lorsque l'utilisateur appuie sur "." puis "Entrée".

```
?- read(X).
bonjour.
X=bonjour
yes
?- read(X).
'Une petite phrase.'.
X=Une petite phrase.
yes
?- read(X).
Y.
yes
?- read(X).
aa(Y,Z).
```

```
X=aa(_,_). yes
```

Les prédicats `read_atom/1`, `read_integer/1`, `read_number/1` sont similaires à `read/1` mais n'acceptent que des termes d'un certain type. Pour ces prédicats là, il suffit d'appuyer sur "Entrée" pour valider la saisie. L'utilisation de "`.`" à la fin de la saisie est optionnelle.

## 12.2 Génération de nombres aléatoires

Gprolog possède des prédicats permettant la génération de nombres aléatoires.

### 12.2.1 Initialisation

Le générateur aléatoire utilise une "*graine*" pour choisir les valeurs. Une fois la graine fixée la liste des valeurs générées est déterministe. Cette graine est par défaut initialisée à 1. Par conséquent à chaque utilisation de gprolog, les nombres sont générés dans le même ordre. Afin d'éviter cela, il faut modifier la graine.

Le prédicat `set_seed/1` prend en entrée un entier qui sera la nouvelle valeur de la graine. Il est également possible d'utiliser le prédicat `randomize/0` qui choisit une graine en fonction de l'heure du système.

### 12.2.2 Tirage

Les tirages se font en utilisant les prédicats `random/1` et `random/3`.

Le prédicat `random/1` lie la variable donnée en paramètre avec un nombre pris aléatoirement supérieure ou égale à 0 et strictement inférieure à 1.

```
?- random(X).
X = 0.80871582031
yes
```

Le prédicat `random/3` prend en argument deux nombres et une variables et réussit si le premier nombre est plus petit que le second et lie la variable avec une valeur choisie au hasard supérieure ou égale au premier nombre et strictement inférieure au second nombre. Si les deux nombres sont entiers, la valeur tirée sera entière.

```
?- random(1,2.1,X).
X = 1.0873443604
yes
?- random(1,5,X).
X = 2
yes
```

## 12.3 Trouver toutes les solutions

Il est parfois utile d'avoir accès à l'ensemble des réponses à un problème donné. Par exemple on peut vouloir connaître le nombre de repas contenant de la salade. Prolog possède trois prédicats permettant de faire cela.

### 12.3.1 `findall/3`

Le prédicat `findall(V,But,L)` réussit si `L` est la liste des valeurs que peut prendre `V` pour résoudre le but `But`.

```
?- findall(X,repas(X,truite,melon),L).
L = [salade,avocat,huitre]
yes
```

Les valeurs sont données dans l'ordre dans lesquelles elles sont trouvées lors de l'exploration de l'arbre de résolution.

Remarque : `But` peut très bien contenir des variables libres :

```
?- findall(X,repas(X,Y,melon),L).
L = [salade,salade,salade,salade,avocat,avocat,avocat,avocat,huitre,huitre,huitre,huitre]
yes
```

Dans cet exemple puisque dans l'arbre de résolution `X` obtient une valeur avant `Y`, cette valeur est dupliquée 4 fois, puisque `Y` peut ensuite prendre 4 valeurs.

Enfin il est possible d'instancier `V` avec une liste de variables :

```
?- findall([X,Y],repas(X,Y,melon),L).
L = [[salade,steak],[salade,escalope],[salade,truite],[salade,daurade],[avocat,steak],
[avocat,escalope],[avocat,truite],[avocat,daurade],[huitre,steak],[huitre,escalope],
[huitre,truite],[huitre,daurade]]
yes
```

### 12.3.2 bagof/3 et setof/3

Ces deux prédicats s'utilisent comment `findall/3`. Le prédicat `bagof` diffère de `findall` lorsqu'il y a des variables libres dans `But`. Ils séparent les résultats en fonctions des valeurs de ces variables :

```
?- bagof(X,repas(X,Y,melon),L).
L = [salade,avocat,huitre]
Y = daurade ? ;
L = [salade,avocat,huitre]
Y = escalope ? ;
L = [salade,avocat,huitre]
Y = steak ? ;
L = [salade,avocat,huitre]
Y = truite
yes
```

Le prédicat `setof/3` est similaire mais en plus il trie la liste des résultats et élimine les doublons.

Quatrième partie

# Applications à la programmation logique

## 13 Introduction

La programmation logique , à travers un langage comme PROLOG, permet d'aborder de nouveaux problèmes, ou en tout cas , permet de traiter des problèmes avec une nouvelles approche.
Le style de programmation déclaratif, proche des mathématiques, les possibilités d'exploration des solutions (backtrack), la programmation par contraintes permet de décrire des solutions rapidement et de manière élégante. Dans certains cas bien sûr, l'efficacité n'est pas au rendez-vous. Mais là n'est pas notre propos.

En dehors des problèmes classiques mettant en jeu les types inductifs et la récursivité, PROLOG permet de résoudre des problèmes de types variés : génération et test, calcul formel, traitement des langages, "intelligence artificielle"...

## 14 Problèmes de génération-test (generate & test)

Cette classe de problèmes se caractérise par le fait qu'on peut les résoudre avec un mécanisme de génération de solution puis de test.
On produit l'ensemble des possibilités de solutions (phase de génération) puis on filtre celles qui sont des "bonnes" solutions (phase de test) pour obtenir le sous-ensemble des solutions au problème.
Bien sûr du point de vue du langage , la possibilité de backtrack sera utilisée pour la génération des solutions.

### 14.1 Principe d'un programme de génération et test

Tous les problèmes de génération et test sont bâtis sur le même moule :

```
/* solution(S) <-> S est une solution du problème posé. Si S est libre, toutes les
solutions possibles sont générées par backtrack */
solution(S) :- generer(S),tester(S) .
```

Mécanisme :
`generer(S)` est appelé avec une variable libre (`S`) , il génère une première solution.
`tester(S)` est appelé avec une variable instanciée (une proposition de solution `S` fournie par `generer`) et teste la validité de la solution.
Après l'échec ou le succès de tester, le backtrack est lancé sur générer qui fournit une autre solution, et ainsi de suite.

Remarque : l'ordre est important car tester doit recevoir une variable instanciée, sauf si tester ne met en place que des contraintes, dans ce cas il peut être intéressant de mettre tester avant pour des raisons d'efficacité (generer est arrêté dès que les contraintes sont violées).

### 14.2 Exemples

#### 14.2.1 Nombres entiers multiples de 3 entre 0 et 100

Quels sont les nombres entiers multiples de 3 entre 0 et 100 ?

```
multiple3(N) :- generer(N),tester(N).
generer(N) :- enum(N,0,100). /* énumère par backtrack tous les entiers entre 0 et 100 */
tester(N) :- 0 is N rem 3. /* N est égal à 0 mod 3 */
```

Remarque : il existe ici une solution plus simple : générer directement les multiples de 3 (donc pas de test)

```
multiple3(N) :- enum(M,0,33),N is 3*M.
```

### 14.2.2 Voyelles d'un mot

```
voyelleMot(M,V) :- generer(M,V),tester(V).
generer(M,V) :- name(M,M0),length(M0,T),T2 is T-1,enum(T1,0,T2),
                length(M1,T1),concat(M1,[V0|_],M0),name(V,[V0]).
tester(a).
tester(e).
tester(i).
tester(o).
tester(u).
tester(y).
```

### 14.2.3 LES MUTANTS

Cet exemple est typique du mode génération/test. On cherche à écrire un prédicat `mutant(M)` tel que `M` est unifié à un identificateur (nom d'un animal mutant) construit par concaténation avec recouvrement de deux noms d'animaux normaux. Par exemple avec "alligator" et "tortue" on construit le mutant "alligatortue". On pourra, dans une version ultérieure, fixer la taille minimum du recouvrement.
Pour cela on dispose d'une base d'animaux normaux qui servira à la production des mutants.

```
 animal(alligator) .  animal(bouquetin) .    animal(caribou) .
 animal(cheval) .     animal(chevre) .       animal(hibou) .
 animal(lapin) .      animal(lapine) .       animal(pintade) .
 animal(rat) .        animal(ratonlaveur) .
```

Le prédicat mutant engendre par backtrack tous les mutants possibles. La première version que l'on donne n'utilise que peu le mécanisme de contraintes, la seconde version en fait un meilleur usage. On pourra en listant la forme "décompilée" des prédicats s'apercevoir que certaines contraintes sur les arbres sont réalisées par l'introduction de buts associés.

```
/* mutant(M) : M est un animal mutant, construit par combinaison de deux animaux.
Le premier se terminant comme le début du dernier, avec au moins une lettre en commun. */
mutant(M) :-
    /* GENERATION */
    animal(A1),animal(A2),
    /* transformation des identificateurs en listes pour manipulation */
    name(A1,S1),name(A2,S2),
    /* construction d'un mutant */
    concat(D1,F1,S1), /* génère toutes les couples D1,F1 tels que S1=D1.F1 */
    concat(D2,F2,S2), /* génère toutes les couples D2,F2 tels que S2=D2.F2 */
    concat(S1,F2,S_M), /* construit le mutant */
    /* transformation de la liste mutant en identificateur */
    name(M,S_M),
    /* TEST */
    F1=D2, D1\=[], F1\=[],F2\=[].

 ?- mutant(N).
N = alligatorat ? ;
N = alligatoratonlaveur ? ;
N = caribouquetin ? ;
N = chevalligator ? ;
N = chevalapin ? ;
N = chevalapine ? ;
N = hibouquetin ? ;
N = lapintade ? ;
N = ratonlaveurat ? ;
N = ratonlaveuratonlaveur
```

Variante en fixant le nombre minimal de lettres en commun

```
/* mutant(M,N) : M est un animal mutant, construit par combinaison de deux animaux.
Le premier se terminant comme le début du dernier, avec au moins N lettres en commun. */
```

```
mutant(M,N) :-
    /* GENERATION */
    animal(A1),animal(A2),
    /* transformation des identificateurs en listes pour manipulation */
    name(A1,S1),name(A2,S2),
    /* construction d'un mutant */
    concat(D1,F1,S1), /* génère toutes les couples D1,F1 tels que S1=D1.F1 */
    concat(D2,F2,S2), /* génère toutes les couples D2,F2 tels que S2=D2.F2 */
    concat(S1,F2,S_M), /* construit le mutant */
    /* transformation de la liste mutant en identificateur */
    name(M,S_M),
    /* TEST */
    F1=D2, D1\=[], F1\=[],F2\=[],length(F1,L),L>=M.

?- mutant(N,2).
N = caribouquetin ? ;
N = chevalligator ? ;
N = hibouquetin ? ;
N = lapintade
?- mutant(N,3).
N = caribouquetin ? ;
N = hibouquetin ? ;
N = lapintade
?- mutant(N,4)
no
```

### 14.2.4 Tri par permutation

On se propose de réaliser un tri en cherchant parmi toutes les permutations d'une liste donnée, celle qui correspond à la liste ordonnée.

```
/* tri par generation et test */
/* tri(T,T1) : T1 contient tous les éléments de T dans l'ordre croissant */
tri(T,T1) :- permut(T,T1),ordonne(T1) ,!.
```

Remarque : le cut permet de sortir dès qu'on a trouvé une solution (il peut y en avoir plusieurs en cas de doublons).

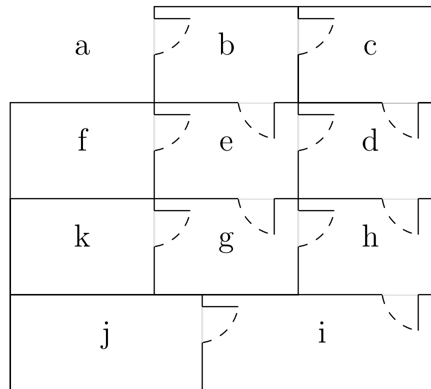
```
/* GENERATION : on génère dans T1 par backtrack toute les permutations de la liste
donnée*/
permut([],[]).
permut([X1|T1],[X2|T2]) :- selectionner(X2,[X1|T1],T3),permut(T3,T2).
/* selectionner(X,T1,T2) : sélection avec backtrack d'un X dans T1, T2=T1 privé de X
*/
/* version1 récursive */
selectionner(X,[X|T],T).
selectionner(X,[Y|T],[Y|T1]) :- selectionner(X,T,T1).
/* ou version2 avec concat */
selectionner(X,T,T0) :- concat(T1,[X|T2],T),concat(T1,T2,T0).
/* TEST : est-ce que la liste est ordonnée ? */
ordonne([_]).
ordonne([X,Y|T]) :- ordonne([Y|T]) , X =< Y.
```

### 14.2.5 Les huit Reines

Le problème est de placer 8 reines sur un échiquier de 8X8 cases de manière à ce qu'aucune d'entre-elles ne soit en prise (on ne peut avoir deux pièces sur une même ligne : horizontale, verticale ou diagonale).

### 14.2.6 Le labyrinthe

Un grand palais est constitué de chambres communicant par des portes. Les chambres et les portes définissent un labyrinthe :




Ce palais est représenté par les passages entre chaque pièce (la pièce ”a” représente l'extérieur) :

```
porte(a,b).  porte(b,c).  porte(i,j).
porte(b,e).  porte(c,d).  porte(d,h).  porte(d,e).
porte(e,f).  porte(e,g).  porte(g,k).  porte(h,i).
```

Une personne arrive en `a` et cherche à téléphoner. Le téléphone se trouve dans une pièce `j`. On notera ce fait :

```
telephone(j).
```

On veut trouver un itinéraire qui permet à la personne de parvenir au téléphone. Pour cela on utilise l'algorithme simple qui consiste à explorer les chambres en notant sur une feuille de papier les chambres visitées :

1. Aller jusqu'à la porte d'une chambre $C$.
2. Si $C$ est sur la liste, ignorer cette chambre et retourner à l'étape 1.
   S'il n'y a pas de nouvelle chambre en vue, revenir à la chambre précédente et chercher une autre chambre.
3. Sinon, entrer dans la chambre $C$ et l'ajouter à la liste.
4. S'il n'y a pas de téléphone dans la chambre, retourner à l'étape 1. Sinon arrêter, la liste contient l'itinéraire qui conduit à la chambre.

**1.1** On rédigera une première version du programme qui imprime les messages du type :
"j'entre dans la pièce P", "j'ai trouvé le téléphone dans la pièce q" et "Zut ! il n'y a pas de téléphone dans ce palais" selon les cas.

**1.2** Dans une deuxième version, le système doit produire l'itinéraire trouvé.
Peut-on trouver d'autres chemins avec ce programme ? Si oui, modifier le programme pour n'en obtenir qu'un seul.

**1.3** On rappelle le prédicat prédéfini `bagof(X,But,Ens)` qui réussit si l'ensemble `Ens` est non vide avec : `Ens`= {`X`|`But` est vrai}.
Exemple : dans la base de faits sur les repas, à la question `bagof(X,plat(X),Les_plats)` Prolog répond `yes` avec `Les_plats = [grillade,poulet,dorade,bar]`.
Écrire le programme Prolog qui produit l'itinéraire le plus court pour trouver le téléphone.

On commencera par définir le prédicat `ca_passe(X,Y)` qui veut dire que les pièces `X` et `Y` sont reliées par une porte et que l'on peut passer de l'une à l'autre. `ca_passe(X,Y) :- porte(X,Y).`
`ca_passe(X,Y) :- porte(Y,X).`

### 14.2.7 Conversion nombre Romain/Nombre arabe

Les nombres romains sont définis par les symboles suivants :

| | | | |
|---|---|---|---|
| I=1 | V=5 | IV=4 | IX=9 |
| X=10 | L=50 | XL=40 | XC=90 |
| C=100 | D=500 | CD=400 | CM=900 |
| M=1000 | | | |

Pour écrire un nombre il faut assembler les symboles dans l'ordre décroissant par rapport à leur valeur.

La valeur du nombre est la somme des valeurs des symboles. Enfin il faut utiliser le nombre minimum de symboles, c'est à dire qu'il faut utiliser, par exemple : IV à la place de IIII.

Par exemple 1498=1000+400+90+5+3=MCDXCVIII.

Nous utilisons les chaînes de caractères pour coder les nombres romains afin évité toute ambiguïté avec les variables Prolog. Dans Prolog les chaînes de caractères sont représentés par les listes d'entier, chaque entier correspondant au code ASCII d'une lettre.

Pour lire un nombre romain il faut lire symbole par symbole, en commençant par la droite tout en testant que le nombre restant est valide. Par exemple après IX il ne peut rien y avoir et après CM il ne peut y avoir qu'un nombre strictement plus petit que 100.

#### 14.2.8 Le fermier, la poule , le renard et le grain

Un fermier va au marché avec un sac de grain, une poule et un renard. Il doit traverser une rivière avec une barque, mais cette barque n'a que deux places. En son absence, il ne peut laisser la poule et le grain ensemble sur une même rive, ni le renard et la poule. Trouver la séquence des traversées en barque pour que tu le monde puisse passer la rivière sans problème.

Un état est décrit par un liste `[L1,L2,B]` où `L1` représente la liste des présents sur la rive 1 et `L2` la liste des présents sur la rive 2, `B` est la rive ou se trouve le bateau.

Lorsqu'on réalise une traversée de la rive 1 à la rive 2 il faut s'assurer de 2 choses :
- on n'a pas laissé un couple "interdit" (loup-poule ou poule-grain)
- on ne retombe pas sur un état passé , sinon on tourne en rond.

Pour cela on stockera dans une liste les états déjà visités.

# 15 Solution des exercices

## 15.1 Les huit Reines

```
/* reines(L) : L est un placement qui résout le pb des 8 reines. L est une permutation
des entiers de 1 à 8 avec la signification suivante : le ième élément de L est le n° de
ligne ou placer la reine dans la ième colonne.
rq : par def toute solution au pb est telle qu'il n'y a qu'une seule dame sur chaque
ligne et une seule dame sur chaque colonne. */
reines(L) :- /* GENERATION */
             liste(8,L1),permut(L1,L),
             /* génération de la liste des entiers
             de 1 à 8 dans un ordre quelconque */
             /* TEST */
             placementOK(L) .

/* liste(N,L) construit une liste L contenant les N premiers entiers (de 1 à N) */
liste(1,[1]).
liste(N, [N|L]) :- N>1,M is N-1,liste(M,L).

/* permutat : cf. ci-dessus */
/* placementOK(L) : le placement représenté par L est une solution au pb des 8 reines */
placementOK([]) .
placementOK([R|L]) :- placementOK(L),not(attaque(R,L)) .

/* attaque(R,L) la reine R (R est la ligne, colonne=1) attaque l'une des reines de L */
attaque(R,L) :- selectRang(L,R1,N),diff(R,R1,N),!.

/* selectRang(L,R,N) : sélectionne par backtrack
un élément R quelconque de L, N est son rang */
selectRang(L,R,N) :- concat(T1,[R|_],L),length(T1,N1),N is N1+1.

/* abs(R-R1) = N */
diff(R,R1,N) :- R1 is R-N.
```

```
diff(R,R1,N) :- R1 is R+N.
```

Solution :

```
?- reines(L),!.
```

résolution en 63 millisecondes

```
L = [8,4,1,3,6,2,7,5]
```

remarque : il existe d'autres solutions (92)

En exercice : programme pour N reines (échiquier NxN), attention : pour N=2 et N=3, il n'y pas de solution.

## 15.2 Le labyrinthe

Voici les 3 versions du programme :

```
/* VERSION 1 */
/* telephoner(Piece) : on a réussit à trouver le téléphone en partant de Piece */
telephoner(Piece) :-
      aller(Piece,P,[Piece]), /* il faut gagner toutes les pièces possible
                               depuis la pièce de départ */
      telephone(P), /* et vérifier si un telephone s'y trouve */
      !, /* si oui , on arrête les recherches */
      nl,write('j ai trouvé le tel dans la pièce '),
      write(P),nl . /* on a réussi */
telephoner(_) :- write('zut ...'). /* on n'a pas pas trouvé le téléphone */

/* aller(X,Y,T) : aller de la pièce X à la pièce Y.
T est l'ensemble des pièce visitées */
aller(X,X,_).
aller(X,Y,T) :- ca_passe(X,Z), /* on passe par Z */
            not(appartient(Z,T)), /* on n'a pas visité Z */
            write('j entre dans la pièce '),write(Z),nl,
            aller(Z,Y,[Z|T]).
/* VERSION 2*/
telephoner(Piece,I) :- /* I est l'itinéraire suivi */
      telephone(P),aller(Piece,P,[Piece],I),affichage(I).

/* aller(X,Y,L1,L2) : L2 est le chemin pour aller de X à Y
L1 est l'ensemble des pièces visitées pour arriver en X */
aller(X,X,_,[X]) :-!.
aller(X,Y,T,[X|S]) :-
      ca_passe(X,Z),
      not(appartient(Z,T)),
      aller(Z,Y,[Z|T],S).

/* affichage de l'itinéraire */
affichage([X]) :-
      write('j ai trouvé le tel dans la pièce '),write(X),nl.
affichage([X,Y|I]) :-
      write('je passe par la pièce '),write(X),nl,affichage([Y|I]).
/* VERSION 3 */
/* en gardant "aller" et "affichage" de la version 2 */
telephoner(Piece,I) :- /* I est l'itinéraire suivi */
telephone(P),
bagof(I,aller(Piece,P,[Piece],I),S),
/* S est l'ensemble des chemins */
min_liste(S,I),affichage(I). /* I est le plus petit chemin */

/* min_liste(LL,L) : L est la plus petite liste de L */ min_liste([L],L).
min_liste([L1,L2|R],L3) :-
```

```
    length(L1,N),min_liste([L2|R],L3),length(L3,M),N>M,!.
min_liste([L1,L2|R],L1). /* L1 est forcement le min */
```

## 15.3 Conversion nombre romain/nombre arabe

```
/* romain(M,S) : M est une chaîne de caractères correspondant à un chiffre romain
valant S */
romain([],0) .
/* "CM"=[67,77] */
romain([67,77|S],S1) :- romain(S,S0),S0<100,S0>=0,S1 is S0+900.
/* "CD"=[67,68] */
romain([67,68|S],S1) :- romain(S,S0),100>S0,S0>=0,S1 is S0+400.
/* "XC"=[88,67] */
romain([88,67|S],S1) :- romain(S,S0),10>S0,S0>=0,S1 is S0+90.
/* "XL"=[88,76] */
romain([88,76|S],S1) :- romain(S,S0),10>S0,S0>=0,S1 is S0+40.
romain("IX",9).
romain("IV",4).
/* "M"=[77] */
romain([77|S],S1) :- romain(S,S0),3000>S0,S0>=0,S1 is S0+1000.
/* "D"=[68] */
romain([68|S],S1) :- romain(S,S0),400>S0,S0>=0,S1 is S0+500.
/* "C"=[67] */
romain([67|S],S1) :- romain(S,S0),300>S0,S0>=0,S1 is S0+100.
/* "L"=[76] */
romain([76|S],S1) :- romain(S,S0),40>S0,S0>=0,S1 is S0+50.
/* "X"=[88] */
romain([88|S],S1) :- romain(S,S0),30>S0,S0>=0,S1 is S0+10.
/* "V"=[86] */
romain([86|S],S1) :- romain(S,S0),4>S0,S0>=0,S1 is S0+5.
/* "I"=[73] */
romain([73|S],S1) :- romain(S,S0),3>S0,S0>=0,S1 is S0+1.
```

## 15.4 Le fermier, la poule, le renard et le grain

```
/* laGrandeTraversee(L) : L est la liste des états par lesquels passer pour faire
traverser tous les animaux */
laGrandeTraversee(L) :- etatInitial(E1),etatFinal(E2),traversee(E1,E2,[E1],L).
/* etatInitial et etatFinal donne les états de départ et d'arrivée */
etatInitial([[poule,renard,grain],[],1]).
etatFinal([[],[poule,renard,grain],2]).

/* traversee(E1,E2,L1,L2) : E1 est l'état actuel, E2 l'état d'arrivée,
L1 est la liste des états parcourus pour arriver en E1,
L2 est la liste des états permettant de passer de E1 à E2 */
traversee(E1,E1,_,[E1]).
traversee(E1,E2,T,[E1|T1]) :-
            /* choix du prochain état */
            transition(E1,E3),
            /* vérification que cet état n'a pas encore été visité */
            not_boucle(E3,T),
            /* passage de E3 à E2 */
            traversee(E3,E2,[E3|T],T1).
/* transition(E1,E2) : passage de E1 à E2 en un voyage
On vérifie qu'on a pas laissé seuls de couple "interdits" */
/* passage d'une rive à l'autre en emportant quelque chose */
transition([L1,L2,1],[L3,L4,2]) :- sur(L3),length(L1,N1),N1>0,N3 is N1-1,
                                   length(L3,N3),concat(L1,L2,L5),permut(L5,L6),
```

```
                                    concat(L3,L4,L6),inclus(L3,L1).
transition([L1,L2,2],[L3,L4,1]) :- sur(L4),length(L1,N1),N1<3,N3 is N1+1,
                                    length(L3,N3),concat(L1,L2,L5),permut(L5,L6),
                                    concat(L3,L4,L6),inclus(L1,L3).
/* passage à "vide" */
transition([L1,L2,1],[L1,L2,2]) :- sur(L1).
transition([L1,L2,2],[L1,L2,1]) :- sur(L2).
/* not_boucle(E,L) : l'état E n'est pas déjà dans la liste L */
not_boucle(_,[]).
not_boucle(E1,[E2|T]) :-not(equiv(E1,E2)),not_boucle(E1,T).
/* equiv(E1,E2) : deux états sont équivalents s'il le fermier est sur la même
berge et que sur la même berge il y a les même choses à une permutation près :
[renard,grain]/[grain,renard] */
equiv([L1,_,B],[L2,_,B]) :- permut(L1,L2).
/* sur(L) : les éléments de la liste L peuvent être laissés ensembles sans
surveillance */
sur([]).
sur([poule]).
sur([renard]).
sur([grain]).
sur([renard,grain]).
/* inclus(L1,L2) : la liste L1 est incluse dans L2 */
inclus([],_).
inclus([X|T],L) :-appartient(X,L),inclus(T,L).
```