# Programmation

# fonctionnelle et logique

# ---

# Dossier sur PROLOG

# Table des matières

# Liste des programmes

PROLOG est un langage qui, comme son nom l'indique (PROgrammation LOGique), utilise un mode de programmation dit 'logique'. Ce mode de programmation a vu le jour grâce à John Robinson qui a posé en 1965 les bases de la logique. Le développement de PROLOG a commencé en 1972 à l'université de Marseille dans le Groupe d'Intelligence Artificielle de Lumigny dirigé par A. Colmerauer. Il a été poursuivi principalement à Marseille et à Edimbourg. Aujourd'hui, PROLOG est un langage mûr ; pourtant il ne possède toujours pas de norme.

Nous allons voir dans un premier temps ce qu'est la programmation logique afin de mieux comprendre la 'philosophie' des programmes PROLOG. Ensuite, nous nous intéresserons à ce que permet de faire PROLOG. Enfin, nous essaierons de comprendre sa méthode de recherche de solutions.

# I. La programmation logique

## I.1. Un mode de programmation à part

Il est important avant d'expliquer ce qu'est PROLOG et d'étudier son utilisation de bien voir ce qu'il n'est pas ; c'est à dire de comprendre sa 'philosophie'. Pour cela, il faut décrire ce qu'est la programmation logique et la comparer aux autres modes de programmation.

### I.1.1. Les autres modes de programmation

#### *I.1.1.1. La programmation impérative*

Cette programmation s'inscrit dans une démarche algorithmique qui décrit la façon de traiter les données pour atteindre un résultat par une série d'actions. Celles-ci sont toujours de trois types : le test, l'ordre (chaque instruction par laquelle le programme passe est impérative) et l'itération (obtenue grâce aux branchement).

Le déroulement du programme est parfaitement déterministe.

**Exemple de langages** : Fortran, Pascal, C …

#### *I.1.1.2. La programmation fonctionnelle*

Le résultat est ici comme la composition de fonctions. Pratiquement, elle est proche de la programmation impérative ; cependant ses fondements (λ-calcul) ainsi que l'absence de branchement et d'affectation (du moins dans sa forme théorique) en fait un mode de programmation à part.

**Rem** : Dans la programmation fonctionnelle, on distingue deux grandes familles :
- les langages fortement typés : ML (ex : Caml)
- les langages non typés : LISP (ex : Scheme)

### I.1.2. La programmation logique

Les modes de programmation décrits juste au-dessus sont dits procéduraux car ils cherchent à obtenir le résultat grâce à une procédure (qui peut être une suite d'actions ou une composition de fonctions). A cela on oppose la programmation logique qui est dite déclarative. En effet, ici on ne s'occupe pas de la manière d'obtenir le résultat ; par contre, le programmeur doit faire la description du problème à résoudre en listant les objets concernés, les propriétés et les relations qu'ils vérifient.

Ensuite, le mécanisme de résolution (pris entièrement en charge par le langage) est général et universel. Il parcourt de façon non déterministe (cela sera détaillé au chapitre I) toutes les possibilités du problème et peut donc retourner plusieurs solutions.

### I.1.3. La programmation orientée objet

Ce mode de programmation a été mis à part car il regroupe en fait tous les modes précédemment vus en utilisant à la fois des techniques déclaratives et d'autres procédurale.

**Exemple de langages** : C++, Java …

## I.2. Constitution d'un programme PROLOG

Nous avons vu que le principe de la programmation logique est de décrire le problème à résoudre. Dans le cas de PROLOG, cela est formalisé grâce à un langage dérivé du calcul des prédicats (ou calcul du premier ordre). Les prédicats servent à qualifier (donner les caractéristiques de) les objets du problème et à décrire les relations dans lesquelles ils sont impliqués.

### I.2.1. Les faits

Les faits sont des données élémentaires qu'on considère vraies. Ce sont des formules atomiques constituées du nom d'un prédicat (c'est à dire d'une relation (au sens mathématique du terme)) (pour plus de renseignement sur le calcul des prédicat, voir la première partie du cour de Génie Logiciel) suivi entre parenthèse d'une liste ordonnée d'arguments qui sont les objets du problème principal ou d'un sous-problème.

Un programme PROLOG est au moins constitué d'un ou plusieurs fait(s) car c'est à partir de lui (d'eux) que PROLOG va pouvoir rechercher des preuves pour répondre aux requêtes de l'utilisateur (voir chapitre III.1.3.2 page 27 pour comprendre comment fonctionne un programme PROLOG) ; ce sont en quelque sorte ses hypothèses de travail.

**Ex** : Henri IV est le père de Louis XIII se traduit par : `pere(henri4,louis13).`
Marie de Médicis est la mère de Henri IV se traduit par : `mere(marie-medicis,henri4).`
Adam est l'ancêtre de tout le monde se traduit par : `ancetre(Adam,X).`
Superman est plus fort que tout le monde se traduit par : `plusfort(superman,X).`
0 ! = 1 se traduit par : `factorielle(0,1).`

**Rem** : Généralement, on place toutes les déclarations de faits au début du programme même si ce n'est pas obligatoire.

### I.2.2. Les règles

Un programme PROLOG contient presque toujours des règles, cependant ce n'est pas une obligation. Si les faits sont les hypothèses de travail de PROLOG, les règles sont des relations qui permettent à partir de ces hypothèses d'établir de nouveaux faits par déduction (si on a démontré F1 et F1⇒F2 alors on a démontré F2).

**Ex** : La relation telle que si on est invincible, alors on est plus fort que tout le monde se traduit par la règle :
`plusfort(X,Y):-invincible(X).`

Les relations qui se traduisent en règle sont de la forme : $H_1$ ? $H_2$ ? … $H_n \Rightarrow C$

Où :

- ? peut être soit une disjonction (∨) soit une conjonction (∧)
- $H_1$, $H_2$,… $H_n$ sont des formules atomiques ou des directives de contrôle
- C est une formule atomique.

**Ex** : La relation telle que si on est le père du père ou de la mère de quelqu'un alors on est son grand-père se traduit par :
```
grandpere(X,Y):-pere(X,Z),pere(Z,Y).
grandpere(X,Y):-pere(X,Z),mere(Z,Y).
```
ou encore par :
```
grandpere(X,Y):-pere(X,Z),(pere(Z,Y);mere(Z,Y)).
```
Par contre, la relation telle que si on est grand-parent alors on est grand-père ou grand-mère n'est pas traduisible par :
```
(grandpere(X,Y);grandmere(X,Y)):-grandparent(X,Y);
```

Les variables utilisées dans une règle sont universellement quantifiées par PROLOG (avec quantificateur ∀). Ainsi, il interprète $H_1$ ? $H_2$ ? … $H_n \Rightarrow C$ par $\forall X_1, X_2, \ldots X_m$ ($H_1$ ? $H_2$ ? … $H_n \Rightarrow C$) si $X_1, X_2, \ldots X_m$ sont les variables des $H_k$.

**Ex** : `grandpere(X,Y):-pere(X,Z),mere(X,Y).` est quantifié
- soit par : ∀X ∀Y ∀Z (pere(X,Z) & mere(X,Y))⇒grandpere(X,Y)
- soit par : ∀X ∀Y ,∃Z / (pere(X,Z) & mere(X,Y))⇒grandpere(X,Y)

`plusintelligent(X,Y):-genie(X).` est quantifié
- soit par : ∀X ∀Y genie(X)⇒plusintelligent(X,Y)
- soit par : ∀X genie(X) ⇒ ∀Y plusintelligent(X,Y)

### I.2.3. Les conventions de SWI-PROLOG©

Il existe plusieurs éditeur/débuggeur PROLOG. Nous utiliserons ici SWI-PROLOG© développé par Jan Wielemaker à l'université d'Amsterdam et qui est distribué gratuitement.

Pour pouvoir fonctionner, un programme SWI-PROLOG© doit obéir à certaines conventions :

- Tous les faits et règles doivent se terminer par un « . ».
- Les variables utilisées dans les faits ou les règles commencent toujours par une majuscule ou « _ ».

**Rem** : Mieux vaut faire commencer les variables classiques par des majuscules. En effet, cela permet de différencier sans ambiguïté dans le mode débuggage les variables du programme, des variables internes générées dynamiquement par PROLOG qui commencent par « _ ».

- A l'inverse, tout le reste commence par une minuscule.
- Il ne faut pas mettre d'espace entre le prédicat et la parenthèse ouvrante qui l'accompagne.
- Les commentaires sont mis entre « /* » et « */ » ou commencent par « % » et se terminent sur la même ligne.

**Rem** : Ces conventions s'appliquent également au requêtes faites par l'utilisateur dans l'interpréteur de commande de SWI-PROLO©

# II. Utilisations de PROLOG

## II.1. Interrogation de bases de données relationnelles

Il est très facile d'utiliser PROLOG pour programmer une base de données et l'interroger. En effet, il suffit de déclarer en tant que faits toutes les données de la base.

**Rem** : Il existe en fait une extension de PROLOG appelée DATALOG qui est justement orientée base de données.

Grâce aux règles, on peut obtenir des sous-tables ou établir des relations entre les tables.

**Rem** : C'est ce qui fait la puissance d'un langage comme PROLOG ou DATALOG par rapport à SQL où toutes les données de la base doivent être explicitement énoncées.

**Rem** : Les bases de données sont un exemple où le programme peut ne contenir que des faits sans aucune règle même si cela fait perdre une partie de l'intérêt de PROLOG.

### II.1.1. Programmation de la base de données

**Programme 1 : Biographie des rois de France**

```
% < ! >Attention : pas d'accent dans les programmes

/***************** Les faits ******************/
/*
Arguemnts du predicat 'bio' :
bio(enfant, sexe, annee_naissance, annee_mort, pere, mere)
*/
bio(louis13,                h, 1601, 1643, henri4,          marie_medicis).
bio(elisabeth_France,       f, 1603, 1644, henri4,          marie_medicis).
bio(marie_therese_Autriche, f, 1638, 1683, philippe4,       elisabeth_france).
bio(louis14,                h, 1638, 1715, louis13,         anne_autriche).
bio(grand_dauphin,          h, 1661, 1711, louis14,         marie_therese_autriche).
bio(louis_bourbon,          h, 1682, 1712, grand_dauphin,   marie_anne_baviere).
bio(philippe5,              h, 1683, 1746, grand_dauphin,   marie_anne_baviere).
bio(louis15,                h, 1710, 1774, louis_bourbon,   marie_adelaide_savoie).
bio(louis_dauphin,          h, 1729, 1765, louis15,         marie_leczcynska).
bio(louis16,                h, 1754, 1793, louis_dauphin,   marie_josephe_saxe).
bio(louis18,                h, 1755, 1824, louis_dauphin,   marie_josephe_saxe).
bio(charles10,              h, 1757, 1836, louis_dauphin,   marie_josephe_saxe).
bio(clotilde,               f, 1759, 1802, louis_dauphin,   marie_josephe_saxe).
bio(louis17,                h, 1785, 1795, louis16,         marie_antoinette).
bio(philippe1,              h, 1640, 1701, louis13,         anne_autriche).
bio(philippe2,              h, 1674, 1723, philippe1,       charlotte_baviere).
bio(louis_orleans,          h, 1703, 1752, philippe,        francoise_marie_bourbon).
```

```
bio(louis_philippe,          h, 1725, 1785, louis_orleans,   augusta_marie_bade).
bio(philippe_egalite,        h, 1747, 1793, louis_philippe,  louise_henriette_bourbon_conti).
bio(louis_philippe1,         h, 1773, 1850, philippe_egalite,
                                            louise_marie_adelaide_bourbon_penthievre).

/***************** Les regles ******************/
/*enfant(enfant,parent)*/
/*R1*/ enfant(X,Y):-bio(X,_,_,_,Y,_).
/*R2*/ enfant(X,Y):-bio(X,_,_,_,_,Y).
/*ptenfant(petit-enfant,grand-parent)*/
/*R3*/ ptenfant(X,Y):-enfant(X,Z),enfant(Z,Y).
/*descendant(descendant,ancetre)*/
/*R4*/ descendant(X,Y):-enfant(X,Y).
/*R5*/ descendant(X,Y):-enfant(X,Z),descendant(Z,Y).
```

Ce programme définit avec les faits une table ‘bio’ dont les attributs sont le nom du fils, son sexe, son année de naissance, de mort, le nom de son père et de sa mère.

Grâce aux règles il définit également trois sous-tables : ‘enfant’, ‘ptenfant’ et ‘descendant’ qui ont comme attributs :

- enfant(NomEnfant, NomParent)
- ptenfant(NomPetitEnfant, NomGrandParent)
- descendant(NomDescendant, NomAncetre)

### II.1.2. Interrogation de la base de données

La base de données étant en place, nous allons l’interroger grâce à des requêtes PROLOG. Il est possible de l’interroger de nombreuses manières. En voici quelques exemples.

#### *II.1.2.1. Vérification de la présence d’une donnée dans la table*

Il est bien sûr possible de savoir si une donnée existe bien dans la base.

**Ex** : Regardons si Louis XIII qui a vécu entre 1601 et 1643 est le fils de Henri IV et Marie de Médicis :

```
?- bio(louis13, h, 1601, 1643, henri4, marie_medicis).
Yes
```

C’est bien le cas.

**Ex** : Regardons si Louis XVII qui a vécu entre 1785 et 1795 est le fils de Louis XVI et Marie de Médicis :

```
?- bio(louis17, h, 1785, 1795, louis16, marie_medicis).
No
```

Ce n’est pas le cas car sa mère est Marie-Antoinette.

#### *II.1.2.2. Recherche d’une liste simple*

On peut extraire de la table ‘bio’ une sous-table à un seul attribut.

Cela correspond à l’opération : $\Pi_X$ ($\sigma_{Conditions}$ (bio) ) où

- X est un (et un seul) attribut de la table ‘bio’
- Conditions est un ensemble de conjonctions et de disjonctions

**Ex** : Quelles sont les femmes qui figurent comme enfant ?
C’est à dire que contient $\Pi_{NomEnfant}$ ($\sigma_{Sexe=f}$ (bio) ) ?

```
?- bio(Qui,f,_,_,_,_).
Qui = elisabeth_france ;
Qui = marie_therese_autriche ;
Qui = clotilde ;
No
```

Il y a donc trois filles dans la table.

**Ex** : Qui sont les enfants de Henri IV ?

```
?- bio(Qui,_,_,_,henri4,_).
Qui = louis13 ;
Qui = elisabeth_france ;
No
```

Henri IV a eu 2 enfants.

La sous-table retournée par PROLOG est complète dans le sens où les possibilités solution de la requête sont affichées autant de fois qu’elles répondent au problème.

**Ex** : Quelles sont les femmes qui figurent comme mère ?

```
?- bio(_,_,_,_,_,Mere).
Mere = marie_medicis ;
Mere = marie_medicis ;
Mere = elisabeth_france ;
Mere = anne_autriche ;
Mere = marie_therese_autriche ;
Mere = marie_anne_baviere ;
Mere = marie_anne_baviere ;
Mere = marie_adelaide_savoie ;
Mere = marie_leczcynska ;
Mere = marie_josephe_saxe ;
Mere = marie_josephe_saxe ;
Mere = marie_josephe_saxe ;
Mere = marie_josephe_saxe ;
Mere = marie_antoinette ;
Mere = anne_autriche ;
Mere = charlotte_baviere ;
Mere = francoise_marie_bourbon ;
Mere = augusta_marie_bade ;
Mere = louise_henriette_bourbon_conti ;
Mere = louise_marie_adelaide_bourbon_penthievre ;
No
```

#### *II.1.2.3. Recherche d'une liste multiple*

On peut aussi extraire de la table 'bio' une sous-table ayant plus d'un seul attribut.

Cela peut correspondre à l'opération : $\Pi_{X1,X2,\ldots Xn}(\sigma_{Conditions}(bio))$ où

- $X_1, X_2, \ldots X_n$ sont des attributs de la table 'bio'
- Conditions est un ensemble de conjonctions et/ou de disjonctions

**Ex** : Qui sont les parents de Louis XIV ?

```
?- bio(louis14,_,_,_,Papa,Maman).
Papa = louis13
Maman = anne_autriche ;
No
```

Le père de Louis XIV est Louis XIII et sa mère est Anne d'Autriche.

Mais cela peut aussi correspondre à une jointure : $\Pi_{X1}(\sigma_{Conditions}(bio)) \triangleright\!\triangleleft \Pi_{X2,\ldots Xn}(\sigma_{Conditions}(bio))$

**Ex** : Qui sont les parents de Louis XIV ?

```
?- bio(louis14,_,_,_,Pere,_),bio(louis14,_,_,_,_,Mere).
Pere = louis13
Mere = anne_autriche ;
No
```

**Rem** : Si les deux buts de la requête ne sont pas réunis par une conjonction mais par une disjonction, on obtient un tout autre résultat :

```
?- bio(louis14,_,_,_,Pere,_);bio(louis14,_,_,_,_,Mere).
Pere = louis13
Mere = _G481 ;
Pere = _G473
Mere = anne_autriche ;
No
```

En effet, on obtient une sous-table avec deux attributs contenant non pas une réponse mais deux. _G481 et _G473 sont deux variables internes utilisées par SWI-PROLOG© ; si elles apparaissent dans le résultat cela signifie que le résultat est bon quelles que soient leurs valeurs.

Ce résultat peut donc s'interprète par :

- Si le père de Louis XIV est Louis XIII, sa mère peut être n'importe qui.
- Si la mère de Louis XIV est Anne d'Autriche, son père peut être n'importe qui.

Ce qui est bien sûr incorrect.

Par rapport à cette dernière remarque, il faut ajouter qu'il est quand même possible d'employer une disjonction dans la requête. Cependant, ce connecteur étant équivalent à l'opération de réunion ($\cup$) dans l'algèbre relationnelle, il supprime un attribut à la sous-table résultat.

**Ex** : Qui sont les parents de Louis XIV ?

```
?- bio(louis14,_,_,_,Parent,_);bio(louis14,_,_,_,_,Parent).
Parent = louis13 ;
Parent = anne_autriche ;
No
```

On obtient ici une sous-table à un seul attribut avec deux données. De ce fait, on perd l'information sur le sexe des parents puisqu'ils sont tous les deux réunis sous le même attribut.

#### *II.1.2.4. Recherche dans un intervalle de valeurs*

PROLOG peut gérer un intervalle de valeurs.

**Ex** : Quels sont les personnages nés entre 1750 et 1800 ?

```
?- bio(Qui,_,N,_,_,_),1750=<N,N=<1800.
Qui = louis16
N = 1754 ;
Qui = louis18
N = 1755 ;
Qui = charles10
N = 1757 ;
Qui = clotilde
N = 1759 ;
Qui = louis17
N = 1785 ;
Qui = louis_philippe1
N = 1773 ;
No
```

Il y a six enfants qui sont nés entre 1750 et 1800.

La recherche de solutions dans un intervalle de valeur fait apparaître quelques caractéristiques intéressantes de PROLOG :

- L'implémentation de PROLOG ne respecte pas complètement la logique du premier ordre : une requête n'est pas traitée comme une formule complète mais elle est divisée en formules atomiques (les buts) reliées par des disjonctions et/ou des conjonctions et ces sous-formules sont traitées les unes après les autres dans l'ordre où elles se trouvent sur la ligne (de gauche à droite).
- PROLOG considère à priori toute variable numérique comme faisant parti des réels.

**Ex** : Quels sont les personnages nés entre 1750 et 1800 ?

```
?- 1750=<N,N=<1800,bio(Qui,_,N,_,_,_).
ERROR: Arguments are not sufficiently instantiated
```

Par rapport au premier exemple, nous n'avons fait qu'inverser les formules de la requête ce qui en logique ne change rien puisque l'opérateur $\wedge$ est commutatif. Mais comme nous l'avons dit juste au-dessus, PROLOG découpe les requêtes en formules atomiques. Il va donc d'abord s'attacher à résoudre le but `1750=<N`. Or N étant à priori un réel, il y a une infinité de solution à ce but. Il arrive à le détecter et alors il s'arrête.

**Rem** : Nous venons de voir dans cet exemple un des rares cas où PROLOG effectue un contrôle. Sans cela, il chercherait tous les N (réels) ≤ 1750 avant de passer à la formule atomique suivante. Donc il ne terminerait jamais.

Pour contourner ces deux difficultés (les règles sont découpées en formules atomiques et les variables numériques sont considérées comme des réels) dans le cas d'une recherche dans un intervalle de valeurs, PROLOG introduit le mot clé

`between(debut,fin,X)` où X est une variable considérée comme un entier et comprise entre 'debut' et 'fin'.

**Ex** : Quels sont les personnages nés entre 1750 et 1800 ?

```
?- between(1750,1800,N),bio(Qui,_,N,_,_,_).
N = 1754
Qui = louis16 ;
N = 1755
Qui = louis18 ;
N = 1757
Qui = charles10 ;
N = 1759
Qui = clotilde ;
N = 1773
Qui = louis_philippe1 ;
N = 1785
Qui = louis17 ;
```

```
No
```

Cette fois PROLOG veut bien s'exécuter car il y a un nombre fini d'entiers compris entre 1750 et 1800. On retrouve bien les même résultats qu'avec le premier exemple

PROLOG permet également de façon tout à fait naturelle de rechercher des solutions n'appartenant pas à un ensemble de valeurs.

**Ex** : Quels sont les personnages qui ne sont pas nés entre 1750 et 1800 ?

```
?- bio(Qui,_,N,_,_,_),(1750>N;N>1800).
Qui = louis13
N = 1601 ;
Qui = elisabeth_France
N = 1603 ;
Qui = marie_therese_Autriche
N = 1638 ;
Qui = louis14
N = 1638 ;
Qui = grand_dauphin
N = 1661 ;
Qui = louis_bourbon
N = 1682 ;
Qui = philippe5
N = 1683 ;
Qui = louis15
N = 1710 ;
Qui = louis_dauphin
N = 1729 ;
Qui = philippe1
N = 1640 ;
Qui = philippe2
N = 1674 ;
Qui = louis_orleans
N = 1703 ;
Qui = louis_philippe
N = 1725 ;
Qui = philippe_egalite
N = 1747 ;
No
```

**Rem** : Ici non plus on ne peut pas inverser l'ordre des formules atomiques de la requête.

```
?- (1750>N;N>1800),bio(Qui,_,N,_,_,_).
ERROR: Arguments are not sufficiently instantiated
```

On obtient le même type d'erreur que lorsqu'on recherchait des solutions dans un intervalle de valeurs.

#### *II.1.2.5. Utilisation des sous-tables*

Lorsque nous avions programmé la base de données, en plus de la table 'bio', nous avions également déclaré trois autres sous tables : 'enfant', 'ptenfant' et 'descendant' par l'intermédiaire de règles. Il est tout à fait possible d'interroger ces dernières comme nous l'avons fait avec 'bio'.

**Ex** : Quels sont les descendants de Louis XIV ?

```
?- descendant(X,louis14).
X = grand_dauphin ;
X = louis_bourbon ;
X = philippe5 ;
X = louis15 ;
X = louis_dauphin ;
X = louis16 ;
X = louis18 ;
X = charles10 ;
X = clotilde ;
X = louis17 ;
No
```

Louis XIV a 10 descendants dans la base de données.

**Ex** : Quels sont les ancêtres de Louis XVII ?

```
?- descendant(louis17,X).
```

```
X = louis16 ;
X = marie_antoinette ;
X = louis_dauphin ;
X = marie_josephe_saxe ;
X = louis15 ;
X = marie_leczcynska ;
X = louis_bourbon ;
X = marie_adelaide_savoie ;
X = grand_dauphin ;
X = marie_anne_baviere ;
X = louis14 ;
X = marie_therese_autriche ;
X = louis13 ;
X = anne_autriche ;
X = henri4 ;
X = marie_medicis ;
X = philippe4 ;
X = elisabeth_france ;
X = henri4 ;
X = marie_medicis ;
No
```

Louis XVII a 19 ancêtres dans la base de données. Pourtant, on obtient 20 réponses. En effet, Henri IV apparaît deux fois car deux liens de parenté différents les relient.

## II.2. Formalisation de systèmes experts

### II.2.1. PROLOG et les systèmes experts

PROLOG est parfaitement adapté pour formaliser des systèmes experts. En effet, un système expert est un programme informatique simulant l'intelligence humaine dans un champ particulier de la connaissance ou relativement à une problématique déterminée. Or PROLOG a justement été conçu dans cette optique là puisqu'il a été fait par des chercheurs en intelligence artificielle.

Un système expert a trois composantes essentielles :

- une base de connaissances, formée des énoncés relatifs aux faits de tous ordres constitutifs du domaine
- un ensemble de règles de décision, consignant les méthodes, procédures et schémas de raisonnement utilisés dans le domaine
- un moteur d'inférence, sous-système qui permet d'appliquer les règles de décision à la base de connaissances.

Or ces trois points sont extrêmement simples à implémenter dans programme PROLOG :

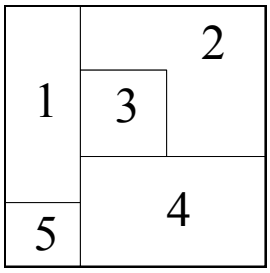
- la base de connaissances est constituée par les faits et quelques règles pour éviter l'énumération exhaustive de tous les faits
- les règles de décision sont des règles (au sens de PROLOG)
- le moteur d'interface est l'interpréteur PROLOG lui-même.

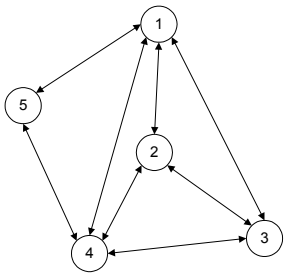
### II.2.2. Constitution de la base de connaissance

En guise d'exemple de système expert, nous allons formaliser le problème de coloriage de région. Les règles de ce problème sont les suivantes :

- Une surface est découpée en un certain nombre de régions de surfaces et de formes variables
- Chaque région doit être coloriée
- Deux régions adjacentes doivent avoir deux couleurs différentes

Nous essaierons de colorier les régions suivantes :

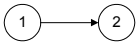


que nous représenterons plutôt par



La deuxième représentation permet de s'abstraire de toute géométrie :

- Un sommet représente une région
- Une arête reliant deux sommets signifie que les régions équivalentes sont adjacentes

C'est cela qu'on va traduire dans le programme PROLOG par le prédicat `adjacent`

**Ex** : `adjacent(1,2).` signifie 

ce qui n'est pas exactement « La région 1 est adjacente à la région 2. »

#### *II.2.2.1. Enumération exhaustive*

La première solution pour constituer la base de connaissance est de faire une énumération exhaustive des régions adjacentes.

**Programme 2 : Base de connaissances exhaustive**

```
%Liste exhaustive des regions adjacentes
adjacent(1,2).          adjacent(2,1).
adjacent(1,3).          adjacent(3,1).
adjacent(1,4).          adjacent(4,1).
adjacent(1,5).          adjacent(5,1).
adjacent(2,3).          adjacent(3,2).
adjacent(2,4).          adjacent(4,2).
adjacent(3,4).          adjacent(4,3).
adjacent(4,5).          adjacent(5,4).
```

Cette base de connaissance fonctionne parfaitement.

**Ex** : Les régions 1 et 2 sont adjacentes :

```
?- adjacent(1,2) ,adjacent(2,1).
Yes
```

La région 2 n'est pas adjacente à la région 11 (qui n'existe pas)

```
?- adjacent(2,11).
No
```

La région 2 n'est pas adjacente à la région 5

```
?- adjacent(2,5).
No
```

Cependant, on voit très bien que ce genre de base de connaissances est limité : si le problème devient trop complexe, on ne peut pas saisir à la main toutes les possibilités. De plus cette écriture sous-exploite le potentiel de PROLOG qui grâce aux règles peut grandement alléger cette écriture.

#### *II.2.2.2. Ecritures condensées*

Nous allons condenser l'écriture exhaustive en remplaçant certains faits par des règles ce qui comme nous le verrons n'est pas sans danger.

##### *II.2.2.2.1. Règle de commutation*

Essayons tout d'abord d'introduire une règle de commutativité. En effet, cette règle si naturelle pour nous quand on parle de coté adjacent fait cruellement défaut à PROLOG et permettrait de réduire de moitié le nombre de faits.

**Programme 3 : Base de connaissances commutative (erronée)**

```
%Liste des régions adjacentes 1 fois (sans commutation)
adjacent(1,2).
adjacent(1,3).
adjacent(1,4).
adjacent(1,5).
adjacent(2,3).
adjacent(2,4).
adjacent(3,4).
adjacent(4,5).
```

```
/*La regle de commutation */
adjacent(X,Y):-adjacent(Y,X).
```

Cette règle de commutation fonctionne très bien dans certains cas.

**Ex** : Le programme détecte parfaitement les régions adjacentes :

```
?- adjacent(2,4).
Yes
?- adjacent(4,2).
Yes
```

2 et 4 sont bien deux régions adjacentes

Malheureusement, dans d'autres cas elle engendre des boucles infinies.

**Ex** : Le programme boucle systématiquement pour détecter des régions non adjacentes :

```
?- adjacent(2,5).
Action (h for help) ? abort
Execution Aborted
```

**Ex** : Enumérer toutes les régions adjacentes à 1

```
?- adjacent(1,X).
X = 2 ;
X = 3 ;
X = 4 ;
X = 5 ;
X = 2 ;
X = 3 ;
X = 4 ;
X = 5 ;
X = 2 ;
X = 3 ;
X = 4 ;
X = 5
…
```

PROLOG ne boucle pas tout seul mais il retourne toujours la même série de solutions tant que l'utilisateur lui en demande encore : il ne s'arrête pas tout seul pour signifier qu'il n'y en a plus d'autres.

En fait, la règle de commutativité remplie parfaitement son rôle pour déterminer que deux régions sont adjacentes car soit elle n'est jamais invoquée soit elle est utilisée une seule fois avant de trouver le fait permettant d'arrêter la recherche.

**Ex** :

```
[debug]  ?- adjacent(2,4).
T Call:  (  8) adjacent(2, 4)
T Exit:  (  8) adjacent(2, 4)
```

PROLOG trouve la preuve directement dans les faits.

```
[debug]  ?- adjacent(4,2).
T Call:  (  8) adjacent(4, 2)
T Call:  (  9) adjacent(2, 4)
T Exit:  (  9) adjacent(2, 4)
T Exit:  (  8) adjacent(4, 2)
```

Cette fois il est obligé d'appliquer la règle de commutativité 1 fois (2ème call) puis il trouve un fait pour elle, il en sort et il répond à la requête.

Par contre, si la requête porte sur deux régions non adjacentes, PROLOG va appeler une infinité de fois la règle de commutativité.

**Ex** :

```
[debug]  ?- adjacent(2,5).
T Call:  (  8) adjacent(2, 5)
T Call:  (  9) adjacent(5, 2)
T Call:  ( 10) adjacent(2, 5)
T Call:  ( 11) adjacent(5, 2)
T Call:  ( 12) adjacent(2, 5)
T Call:  ( 13) adjacent(5, 2)
T Call:  ( 14) adjacent(2, 5)
T Call:  ( 15) adjacent(5, 2)
T Call:  ( 16) adjacent(2, 5)
T Call:  ( 17) adjacent(5, 2)
T Call:  ( 18) adjacent(2, 5)
...
```

Les zone 2 et 5 n'étant pas adjacentes, ni `adjacent(2,5)`, ni `adjacent(5,2)` ne fait partis des faits.
La première action de PROLOG est de regarder si la requête est un fait. Puisque ce n'est pas le cas, il applique la règle de commutation en espérant que cela lui permet de conclure. Il reparcourt donc les faits mais il ne trouve encore aucune preuve pour répondre à la requête. Donc il applique une nouvelle fois la règle de commutation.

Dans le cas d'une énumération de zones adjacentes à une autre, la règle de commutation crée en quelque sorte une infinité de faits en répétant les faits une fois à l'endroit, une fois en commutant les paramètres et ceci une infinité de fois.

**Ex** :
```
[debug]  ?- adjacent(1,X).
T Call:  (  7) adjacent(1, _G304)
T Exit:  (  7) adjacent(1, 2)
X = 2 ;
T Exit:  (  7) adjacent(1, 3)
X = 3 ;
T Exit:  (  7) adjacent(1, 4)
X = 4 ;
T Exit:  (  7) adjacent(1, 5)
X = 5 ;
T Redo:  (  7) adjacent(1, _G304)
T Call:  (  8) adjacent(_G304, 1)
T Redo:  (  8) adjacent(_G304, 1)
T Redo:  (  8) adjacent(_G304, 1)
T Call:  (  9) adjacent(1, _G304)
T Exit:  (  9) adjacent(1, 2)
T Exit:  (  8) adjacent(2, 1)
T Exit:  (  7) adjacent(1, 2)
X = 2
…
```
Au 1er passage, PROLOG trouve tous les faits correspondant à la requête (les régions 2, 3, 4 et 5 sont adjacentes à la 1). Ensuite il appelle la règle de commutation `adjacent(X,Y):-adjacent(Y,X)` et relit les faits. Cette fois aucun ne convient mais il arrive de nouveau à la règle de commutation. Il l'applique et recommence à lire les faits. Après deux commutations, il est revenu (malheureusement il ne le sait pas) à l'état initial ; il trouve donc les mêmes solutions qu'au premier passage…

**Rem** : Il existe des méthodes pour prévenir ces boucles, cependant pour des raisons d'efficacité, elle ne sont pas utilisées dans PROLOG.

*II.2.2.2.2. Enumération par variables*

Une autre façon de condenser l'écriture est de faire l'énumération des quatre premiers faits (et leurs images commutées) par l'intermédiaire d'une variable. En effet, on remarque que ces faits sont tous de la forme `adjacent(1,x)` avec $2 \leq x \leq 5$.

**Programme 4 : Base de connaissances avec énumération par variable (erronée)**
```
% Liste des regions adjacentes autres que la 1
adjacent(2,3).
adjacent(2,4).
adjacent(3,4).
adjacent(4,5).
adjacent(3,2).
adjacent(4,2).
adjacent(4,3).
adjacent(5,4).

% Regles concernant la region 1
adjacent(1,X):-X>=2,X=<5.
adjacent(X,1):-X>=2,X=<5.
```
Ce programme marche très bien avec des requêtes simples

**Ex** : On peut savoir que les régions 3 et 1 sont adjacentes :
```
?- adjacent(1,3),adjacent(3,1).
Yes
```
ou que les régions 5 et 2 ne le sont pas :
```
?- adjacent(5,2).
No
```

Par contre, une erreur peut apparaître lorsqu'on emploie des requêtes un peu plus complexes.

**Ex** : Enumérer toutes les régions adjacentes à 2
```
?- adjacent(2,X).
```

```
X = 3 ;
X = 4 ;
X = 1 ;
No
```

La région 2 est bien adjacente aux régions 1, 3 et 4.

**Ex** : Enumérer toutes les régions adjacentes à 1

```
?- adjacent(1,X).
ERROR: Arguments are not sufficiently instantiated
^  Exception: (7) _G158>=2 ? abort
% Execution Aborted
```

PROLOG refuse d'exécuter la requête. En effet, il s'agit d'un des rares contrôle de PROLOG. Sans ce contrôle, il regarderait `adjacent(1,X)` pour tous les $X \geq 2$ puis pour tous les $X \leq 5$ ; or il y en a une infinité dans les deux cas et il entrerait donc dans une boucle infinie.

Il existe une solution à ce problème. PROLOG fournit un système pour borner une variable entière : `between`.

Ainsi, il faut remplacer les deux règles concernant la région 1 par :

```
% Regles concernant la region 1
adjacent(1,X):-between(2,5,X).
adjacent(X,1):-between(2,5,X).
```

*II.2.2.2.3. Relations entre paramètres*

On remarque que concernant les régions (2,3), (3,4) et (4,5) les deux paramètres `X` et `Y` du prédicat `adjacent` sont liés par la relation Y=X+1. On pourrait donc écrire explicitement cette relation.

**Programme 5 : Base de connaissances avec relation entre paramètres (erronée)**

```
% Regions adjacentes explicitement declarees
adjacent(2,4).
adjacent(4,2).

% Regles concernant la region 1
adjacent(1,X):-between(2,5,X).
adjacent(X,1):-between(2,5,X).

% Regles concernant les couples (2,3), (3,4) et (4,5)
adjacent(X,X+1).
adjacent(X+1,X).
```

Ce programme présente plusieurs défauts :

- D'abord, il est syntaxiquement incorrect. En effet, PROLOG ne reconnaît pas « `X+1` » comme étant la valeur de `X` augmenté de 1. Pour cela, il dispose du mot clé `is` :

  ```
  adjacent(X,Y):-Y is X+1.
  ```

  ```
  adjacent(X,Y):-Y is X+1.
  ```

- Ensuite, comme nous l'avons vu au chapitre précédent, il faut spécifie que `X` est un entier borné en utilisant `between`

  ```
  adjacent(X,Y):-between(2,4,X), Y is X+1.
  ```

  ```
  adjacent(X,Y):-between(2,4,X), Y is X+1.
  ```

Finalement, nous avons pu condenser la base de connaissances en :

**Programme 6 : Base de connaissances condensée (finale)**

```
% Regions adjacentes explicitement declarees
adjacent(2,4).
adjacent(4,2).

% Regles concernant la region 1
adjacent(1,X):-between(2,5,X).
adjacent(X,1):-between(2,5,X).

% Regles concernant les couples (2,3), (3,4) et (4,5)
adjacent(X,Y):-between(2,4,X),Y is X+1.
adjacent(Y,X):-between(2,4,X),Y is X+1.
```

Cette écriture condensée est exactement équivalente au programme exhaustif page 9.

**Ex** : Test sur deux régions adjacentes

```
?- adjacent(1,2),adjacent(2,1).
Yes
```

1 et 2 sont bien adjacentes

**Ex** : Test sur deux régions non adjacentes

```
?- adjacent(2,5).
No
```

2 et 5 ne sont effectivement pas adjacentes

**Ex** : Enumération de régions adjacentes

```
?- adjacent(1,X).
X = 2 ;
X = 3 ;
X = 4 ;
X = 5 ;
No
```

1 est adjacente exactement à 2, 3, 4 et 5

### II.2.3. Mise en place des règles de décision

Pour les règles de décision, on introduit deux nouveaux prédicats :

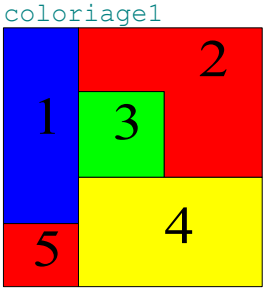
- `conflit(Coloriage)` qui permet de voir si un coloriage des régions respecte les contraintes que nous nous sommes fixées
- `conflit(X,Y,Coloriage)` qui permet de savoir quelles régions adjacentes ont la même couleur

Un coloriage des régions est une fonction qui à chaque région associe une couleur. Elle est réalisée par le prédicat :

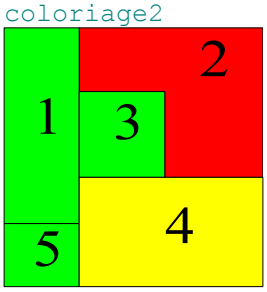
```
color(Region,Couleur,Coloriage)
```

Trois coloriages ont été définis directement dans le programme :

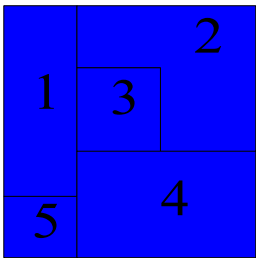
coloriage1




coloriage2




coloriage3




**Programme 7 : Formalisation d'un système expert (complète)**

```
% Description des zones
adjacent(2,4).
adjacent(4,2).
adjacent(1,X):-between(2,5,X).
adjacent(X,1):-between(2,5,X).
adjacent(X,Y):-between(2,4,X),Y is X+1.
adjacent(Y,X):-between(2,4,X),Y is X+1.

% Règles de décisions
conflit(Coloriage):-adjacent(X,Y),color(X, Couleur, Coloriage),color(Y, Couleur, Coloriage).
conflit(X,Y,Coloriage):-adjacent(X,Y),color(X, Couleur, Coloriage),color(Y, Couleur,
Coloriage).

% Exemples de coloriages possibles
/* Coloriage sans conflit */
color(1,bleu,coloriage1).
color(2,rouge,coloriage1).
color(3,vert,coloriage1).
color(4,jaune,coloriage1).
color(5,rouge,coloriage1).

/* Coloriage avec conflit */
color(1,vert,coloriage2).
color(2,rouge,coloriage2).
color(3,vert,coloriage2).
color(4,jaune,coloriage2).
```

```
color(5,vert,coloriage2).

/* Coloriage avec conflit */
color(1,bleu,coloriage3).
color(2,bleu,coloriage3).
color(3,bleu,coloriage3).
color(4,bleu,coloriage3).
color(5,bleu,coloriage3).
```

Nous pouvons maintenant vérifier si un coloriage est valide et dans le cas contraire, connaître les régions adjacentes qui ont la même couleur.

**Ex** : Un coloriage sans conflit

```
?- conflit(coloriage1).
No
```

Le coloriage 1 ne possède effectivement pas de régions adjacentes de même couleur.

**Ex** : Un coloriage avec conflit

```
?- conflit(coloriage2).
Yes
```

Les régions adjacentes de même couleur sont :

```
?- conflit(X,Y,coloriage2),between(X,5,Y).
X = 1
Y = 3 ;
X = 1
Y = 5 ;
No
```

La région 1 a la même couleur que la région 3 et la région 5.

**Ex** : Toutes les régions de la même couleur

```
?- conflit(coloriage3).
Yes
```

Toutes les régions adjacentes entrent en conflit

```
?- conflit(X,Y,coloriage3),between(X,5,Y).
X = 2
Y = 4 ;
X = 1
Y = 2 ;
X = 1
Y = 3 ;
X = 1
Y = 4 ;
X = 1
Y = 5 ;
X = 2
Y = 3 ;
X = 3
Y = 4 ;
X = 4
Y = 5 ;
No
```

Effectivement, il s'agit bien là de toutes les régions adjacentes.

## II.3. Calculs

PROLOG permet évidemment d'effectuer des calculs numériques. C'est ce que nous allons voir en essayant de programmer la factorielle. Pour cela, plusieurs approches sont possibles.

### II.3.1. Utilisation d'une pile

Il s'agit probablement de la façon la plus intuitive de programmer la factorielle. En effet, on définit mathématiquement cette fonction par :

$$\begin{aligned} \textit{fact} : \quad & N \rightarrow N \\ & 0 \rightarrow 1 \\ & n \rightarrow n \times (n-1)! \end{aligned}$$

Or il est possible d'utiliser une syntaxe très proche en PROLOG.

**Programme 8 : Calcul avec pile**

```
/* Calcul de factorielle en utilisant une pile */
fact(0,1.
fact(X,Y):-X>0 , X1 is X-1 , fact(X1,Z) , Y is Z*X.
```

**Ex** : Calcul de 5!

```
?- fact(5,Y).
Y = 120 ;
No
```

PROLOG calcule 5! Et il ne trouve bien qu'un seul résultat à cette requête.

**Ex** : On peut énumérer une liste de factorielles.

```
?- between(1,10,X),fact(X,Y).
X = 1
Y = 1 ;
X = 2
Y = 2 ;
X = 3
Y = 6 ;
X = 4
Y = 24 ;
X = 5
Y = 120 ;
X = 6
Y = 720 ;
X = 7
Y = 5040 ;
X = 8
Y = 40320 ;
X = 9
Y = 362880 ;
X = 10
Y = 3628800 ;
No
```

**Rem** : On peut lui demander de calculer de grands nombres :

```
?- fact(170,Y).
Y = 7.25742e+306
Yes
```

On a bien 170! = 7,257415615307998967396728211129 3e+306.

Cependant, PROLOG possède quand même des limites en calcul numérique probablement dues au codage des flottants en interne.

```
?- fact(171,Y).
ERROR: Arithmetic: evaluation error: `float_overflow'
^  Exception: (8) _G161 is 7.25742e+306*171 ? goals
      [8] _G161 is 7.25742e+306*171
      [7] fact(171, _G161)
^  Exception: (8) _G161 is 7.25742e+306*171 ? abort
% Execution Aborted
```

Une erreur se déclenche soit quand il essaie de calculer 7.25742e+306*171 soit quand il essaie de stocker le résultat dans une variable interne ; il ne peut pas dépasser une limite qui se situe aux alentours de $10^{307}$.

Malheureusement cette façon de programmer factorielle a théoriquement un inconvénient. Pour calculer n! on utilise une pile de hauteur n ce qui n'est pas forcément recommandé. En effet, le principe du programme est de stocker n, n-1, …1 dans une pile puis d'en faire la multiplication.

**Ex** : Calcul de 5!

```
[debug]  ?- fact(5,X).
 T Call: (7) fact(5, _G287)
 T Call: (8) fact(4, _G367)
 T Call: (9) fact(3, _G370)
 T Call: (10) fact(2, _G373)
 T Call: (11) fact(1, _G376)
 T Call: (12) fact(0, _G379)
 T Exit: (12) fact(0, 1)
```

```
 T Exit: (11) fact(1, 1)
 T Exit: (10) fact(2, 2)
 T Exit: (9) fact(3, 6)
 T Exit: (8) fact(4, 24)
 T Exit: (7) fact(5, 120)
X = 120
Yes
```

Au cours des 6 premières lignes du traçage, on peut voir les appels récursifs à la règle « `fact(X,Y):-X>0 , X1 is X-1 , fact(X1,Z) , Y is Z*X.` » avec son premier argument qui est décrémenté à chaque fois et le niveau de l'appel (entre parenthèses) qui augmente. On remarque aussi que sur ces mêmes lignes le deuxième argument est une variable interne sans valeur. Ce n'est qu'une fois qu'on a atteint la règle « `fact(0,1).` » qu'on fait le calcul de 5! (les 6 dernières lignes) en dépilant les niveaux d'appel qui contiennent les valeurs 5, 4, 3, 2 et 1.

**Rem** : Cependant ceci n'est pas réellement un problème puisque comme nous le verrons au chapitre III.1.3.2 pour exécuter un programme PROLOG utilise nécessairement deux piles (une pile de buts et une pile d'environnement). Ainsi, la liste des entiers n, n-1, …, 1est stockée dans la pile d'environnement et ne nécessite pas la création d'une pile dédiée.

### II.3.2. Appel récursif terminal

Si on ne tient pas compte de la remarque précédente, on peut avoir envie d'améliorer ce programme en supprimant la pile des entiers n, n-1, …, 1.Pour cela, nous allons utiliser dans cette deuxième version de la fonction factorielle un appel récursif terminal.

**Programme 9 : Calcul avec appel récursif terminal**

```
/* Calcul de factorielle avec accumalation du résultat dans le 3eme argument du predicat */
facacc(X,Y):-facacc(X,Y,1).
facacc(0,Z,Z).
facacc(X,Y,Z):-X>0,U is Z*X,X1 is X-1,facacc(X1,Y,U).
```

**Rem**: il est possible de définir deux prédicats différents avec le même nom s'ils n'ont pas le même nombre d'arguments.

**Ex** : Calcul de 5!

```
?- facacc(5,X).
X = 120
Yes
```

Comme nous l'avions déjà calculé avec le premier programme de factorielle on retrouve 120.

**Ex** :

```
?- between(1,10,X) , facacc(X,Factorielle).
X = 1
Factorielle = 1 ;
X = 2
Factorielle = 2 ;
X = 3
Factorielle = 6 ;
X = 4
Factorielle = 24 ;
X = 5
Factorielle = 120 ;
X = 6
Factorielle = 720 ;
X = 7
Factorielle = 5040 ;
X = 8
Factorielle = 40320 ;
X = 9
Factorielle = 362880 ;
X = 10
Factorielle = 3628800 .
Yes
```

**Rem** : Comme avec le premier programme, on est limité par les capacités de calcul de PROLOG :

```
?- facacc(171,X).
ERROR: Arithmetic: evaluation error: `float_overflow'
^  Exception: (175) _G1879 is 5.17091e+307*4 ? abort
% Execution Aborted
```

Au passage, on peut rehausser la limite de PROLOG qu'on avait située aux alentours de $10^{307}$ à $10^{308}$.

Cette fois-ci, un résultat intermédiaire est calculé à chaque appel récursif et est 'stocké' dans le troisième argument de `facacc`.

**Ex** : Décomposition des étapes du calcul de 5! :

```
[debug]  ?- facacc(5,X).
 T Call: (7) facacc(5, _G287)
 T Call: (8) facacc(5, _G287, 1)
 T Call: (9) facacc(4, _G287, 5)
 T Call: (10) facacc(3, _G287, 20)
 T Call: (11) facacc(2, _G287, 60)
 T Call: (12) facacc(1, _G287, 120)
 T Call: (13) facacc(0, _G287, 120)
 T Exit: (13) facacc(0, 120, 120)
 T Exit: (12) facacc(1, 120, 120)
 T Exit: (11) facacc(2, 120, 60)
 T Exit: (10) facacc(3, 120, 20)
 T Exit: (9) facacc(4, 120, 5)
 T Exit: (8) facacc(5, 120, 1)
 T Exit: (7) facacc(5, 120)
X = 120
Yes
```

La première action de PROLOG (2ème ligne, niveau de but 8) est d'initialiser la variable dans laquelle le résultat va être progressivement calculé (il applique la règle `facacc(X,Y):-facacc(X,Y,1).`). Ensuite, il va faire cinq appels récursifs (sur les 5 lignes suivantes on remarque que le niveau de but augmente à chaque fois) pendant lesquels il calcule 5×4×3×2×1. Une fois la factorielle calculée (lorsque le premier argument de `facacc` vaut 0), il retourne le résultat en mettant le deuxième argument de `facacc` à la même valeur que son troisième argument (grâce au fait `facacc(0,Z,Z).`) .

**Rem** : On s'aperçoit que bien que ce calcul n'utilise pas de pile, une fois le résultat atteint, celui-ci n'est pas immédiatement retourné car il faut sortir de tous les appels récursifs. On s'aperçoit donc qu'on n'a rien gagné par rapport au programme avec pile : au lieu de faire le calcul de n! en sortant des appels récursifs, on le fait en y entrant.

### II.3.3. Valeur approchée

Naturellement, PROLOG est aussi capable de faire du calcul numérique approché.

#### *II.3.3.1. Première approximation*

Ainsi, on peut utiliser la formule de James Sterling : $n! \approx \sqrt{2\pi n} \times \left(\frac{n}{e}\right)^n$

**Programme 10 : Calcul approché**

```
/************ Pour le calcul exact de X! ***********/
facacc(X,Y):-facacc(X,Y,1).
facacc(0,Z,Z).
facacc(X,Y,Z):-X>0,U is Z*X,X1 is X-1,facacc(X1,Y,U).

/********** Pour le calcul approche de X ! **********
facnum (X,Y,Z,E)
     X: element dont on veut calculer la factorielle
     Y: valeur exacte de X!
     Z: valeur numerique approchee de X!
     E: erreur relative en %
facnum2 (X,Y)
     X: element dont on veut calculer la factorielle
     Y: valeur numerique approchee de X!
************************************************/
facnum(X,Y,Z,E):-Z is sqrt(2*pi*X)*(X/e)**X,
                 facacc(X,Y),
                 E is 100*abs(Y-Z)/Y.
facnum2(X,Y):-Y is sqrt(2*pi*X)*(X/e)**X.
```

**Rem** : Le prédicat `facnum` fait plus que nous donner la valeur approchée de n!, il nous fournit également la valeur réelle et l'écart entre ces deux valeurs. Si on ne veut que la valeur approchée, une seule ligne est nécessaire :
`facnum2(X,Y):-Y is sqrt(2*pi*X)*(X/e)**X.`

**Ex** : Calcul de 5!

```
?- facnum(5,FactExact,FactApp,Err).
FactExact = 120
FactApp = 118.019
Err = 1.65069
```

```
Yes
```

L'écart entre la valeur exacte (120) et la valeur approchée (118.02) est de 1.6 %.

**Ex** : Evolution de l'erreur

```
?- between(1,10,N),facnum(N,FactExact,FactApp,Err).
N = 1
FactExact = 1
FactApp = 0.922137
Err = 7.7863 ;
N = 2
FactExact = 2
FactApp = 1.919
Err = 4.04978 ;
N = 3
FactExact = 6
FactApp = 5.83621
Err = 2.72984 ;
N = 4
FactExact = 24
FactApp = 23.5062
Err = 2.0576 ;
N = 5
FactExact = 120
FactApp = 118.019
Err = 1.65069 ;
N = 6
FactExact = 720
FactApp = 710.078
Err = 1.37803 ;
N = 7
FactExact = 5040
FactApp = 4980.4
Err = 1.18262 ;
N = 8
FactExact = 40320
FactApp = 39902.4
Err = 1.03573 ;
N = 9
FactExact = 362880
FactApp = 359537
Err = 0.921276 ;
N = 10
FactExact = 3628800
FactApp = 3.5987e+006
Err = 0.829596 ;
No
```

On s'aperçoit que l'erreur relative diminue rapidement lorsqu'on tend vers l'infini comme on s'y attendait.

```
?- facnum(170,FactExact,FactApp,Err).
FactExact = 7.25742e+306
FactApp = 7.25386e+306
Err = 0.0490075
Yes
```

On voit que pour n=170 l'approximation est très bonne.

Contrairement aux programmes vus précédemment, le calcul de la valeur approchée est une simple évaluation d'expression numérique : il n'y a aucun mécanisme de récursion ici.

**Ex** : Décomposition des étapes du calcul de 10! :

```
[debug]  ?- facnum2(10,FactApp).
 T Call: (6) facnum2(10, _G284)
 T Exit: (6) facnum2(10, 3.5987e+006)
FactApp = 3.5987e+006
Yes
```

Le calcul de 10! Ne fait appel à aucun autre sous-calcul (du moins au niveau utilisateur) ; en effet, on reste toujours au même niveau dans la pile des buts.

**Rem** : Ici encore, on est limité par les capacités de calcul de PROLOG :

```
?- facnum2(171,X).
ERROR: Arithmetic: evaluation error: `float_overflow'
^  Exception: (7) _G158 is sqrt(2*pi*171)* (171/e)**171 ? abort
% Execution Aborted
```

On voit bien ici que cette limitation est due à l'incapacité de PROLOG à manipuler des nombres trop grands et non pas à un manque de mémoire causé par l'empilement d'un trop grand nombre de buts puisque ce programme n'est pas récursif.

#### *II.3.3.2. Calcul logarithmique*

On peut également s'amuser à calculer le logarithme de factorielle :

$$\log_{10}(n!) = \log_{10}\left(\sqrt{2\pi} \times \frac{n^{n+1/2}}{e^n}\right) = \frac{1}{2}\log_{10}\left(\sqrt{2\pi}\right) + \left(n + \frac{1}{2}\right) \times \log_{10} n - n \times \log_{10} e$$

**Programme 11 : Calcul approché logarithmique**

```
/*********************************************************************
     logfac(N, Y,E):N: element dont on veut calculer la factorielle
                    Y: valeur numerique approchee de log(N!)
                    E: erreur relative (de N!) approchée en %
*********************************************************************/

logfac(N,Y,E):-Y is 0.5*log10(2*pi)+(N+0.5)*log10(N)-N*log10(e),
               E is 100*1/(12*N).
```

**Ex** : Calcul du logarithme de factorielle

```
?- between(1,10,N),logfac(N,FactApp,Err).
N = 1
FactApp = -0.0352045
Err = 8.33333 ;
N = 2
FactApp = 0.283076
Err = 4.16667 ;
N = 3
FactApp = 0.766131
Err = 2.77778 ;
N = 4
FactApp = 1.37118
Err = 2.08333 ;
N = 5
FactApp = 2.07195
Err = 1.66667 ;
N = 6
FactApp = 2.85131
Err = 1.38889 ;
N = 7
FactApp = 3.69726
Err = 1.19048 ;
N = 8
FactApp = 4.601
Err = 1.04167 ;
N = 9
FactApp = 5.55574
Err = 0.925926 ;
N = 10
FactApp = 6.55615
Err = 0.833333
Yes
```

On trouve pour l'erreur relative une valeur approchée très proche de ce qu'on avait obtenu un peu plus haut.

**Rem** : En calculant le logarithme, on peut contourner la limitation de PROLOG sur la grandeur des nombres.
Ainsi on peut calculer des factorielles approchées très grandes avec l'erreur relative.

```
?- logfac(3000,FactApp,Err).
FactApp = 9130.62
Err = 0.00277778
Yes
```

On voit qu'à ce niveau là l'erreur relative est très faible. En effet, la valeur exacte de $\log_{10}(3000!)$ est 9130.617981… ($3000! = 4{,}149\ldots\times 10^{9130}$)

#### *II.3.3.3. Approximation plus fine*

Enfin, il est tout à fait possible d'obtenir une meilleure précision si nécessaire en prenant plus de termes dans la série de n! Pour notre approximation. Par ex : $n! \approx \left( \sqrt{2\pi n} + \frac{1}{12}\sqrt{\frac{2\pi}{n}} + \frac{1}{288}\sqrt{\frac{2\pi}{n^3}} - \frac{139}{51840}\sqrt{\frac{2\pi}{n^5}} \right) \times \left(\frac{n}{e}\right)^n$

**Programme 12 : Calcul approché précis**

```
/************ Pour le calcul exact de N! ***********/
facacc(N, Y):-facacc(N,Y,1).
facacc(0,Z,Z).
facacc(N,Y,Z):-N>0,U is Z*N,N1 is N-1,facacc(N1,Y,U).

/************************************************
facnum (N,Y,Z,E)
     N: element dont on veut calculer la factorielle
     Y: valeur exacte de N!
     Z: valeur numerique approchee de N!
     E: erreur relative en %
************************************************/
facnum(N,Y,Z,E):-Z is (sqrt(2*pi*N)+1/12*sqrt(2*pi/N)+1/288*sqrt(2*pi/N**3)
                                                    -139/51840*sqrt(2*pi/N**5))*(N/e)**N,
                  facacc(N,Y),
                  E is 100*abs(Y-Z)/Y.
```

**Ex** : Evolution de l'erreur relative

```
?- between(1,10,N),facnum(N,FacExact,FacApp,Err).
N = 1
FacExact = 1
FacApp = 0.999711
Err = 0.0288927 ;
N = 2
FacExact = 2
FacApp = 1.99999
Err = 0.000725814 ;
N = 3
FacExact = 6
FacApp = 6
Err = 2.39367e-005 ;
N = 4
FacExact = 24
FacApp = 24
Err = 1.44699e-005 ;
N = 5
FacExact = 120
FacApp = 120
Err = 1.17144e-005 ;
N = 6
FacExact = 720
FacApp = 720
Err = 7.57474e-006 ;
N = 7
FacExact = 5040
FacApp = 5040
Err = 4.84592e-006 ;
N = 8
FacExact = 40320
FacApp = 40320
Err = 3.17773e-006 ;
N = 9
FacExact = 362880
FacApp = 362880
Err = 2.14893e-006 ;
N = 10
FacExact = 3628800
```

```
FacApp = 3.6288e+006
Err = 1.49708e-006
Yes
```

Comme on l'a déjà constaté, l'erreur relative diminue quand on tend vers l'infini. Cependant contrairement à la première approximation (chapitre II.3.3.1), la valeur approchée est très proche de la valeur exacte dès 1!.
Elle est quasiment nulle pour les valeurs de n élevées :

```
?- facnum(170,FacExact,FacApp,Err).
FacExact = 7.25742e+306
FacApp = 7.25742e+306
Err = 2.86351e-011
Yes
```

## II.4. Représentation spécialisée de nombres

PROLOG est particulièrement souple en ce qui concerne la représentation (syntaxique) des nombres. Il est en effet possible de définir et utiliser une représentation différente de celle qu'il implémente par défaut.

Essayons par exemple de représenter les entiers naturels en unaire.

### II.4.1. Définition d'un système numérique

#### *II.4.1.1. Représentation syntaxique*

Habituellement, le système unaire est écrit comme une succession de 1 (ou de 0). Cependant, cette représentation n'est pas utilisable ici car PROLOG ne peut pas faire la différence entre « 111 » signifiant cent onze et « 111 » signifiant trois.

Il est en fait nécessaire d'utiliser à fois des symboles de constantes et des symboles de fonctions pour implémenter le système unaire. On introduit alors :

- Un symbole de constante : `0`
- Un symbole de fonction : `s`

Maintenant, on serait tenté d'écrire en unaire : `0`, `s0`, `ss0`, `sss0`, … Malheureusement, ceci n'est pas non plus possible. En effet, PROLOG interpréterait chaque bloque `s0`, `ss0`, `sss0` … comme des identificateurs différents. PROLOG doit comprendre que le `s` successifs sont des applications. Pour cela, il faut :

- Soit mettre des parenthèses : `0`, `s(0)`, `s(s(0))`, `s(s(s(0)))`…
- Soit mettre des espaces : `0`, `s 0`, `s s 0`, `s s s 0`…

Nous utiliseront dans les exemples suivants l'écriture avec des espaces.

**Programme 13 : Définition d'un système unaire**

```
% Definition des entiers unaires
:-op(100,fy,s).
entier(0).
entier(s X):-entier(X).
```

**Rem** : `:-op(x,y,z)` définit une fonction où
-`x` indique le niveau de priorité de l'opérateur (+ : 500, * : 400, ** : 200)
-`y` signifie qu'on désire une écriture préfixe non parenthésée
-`z` est le symbole de l'opérateur

On peut maintenant travailler sur les entiers naturels avec une représentation unaire.

**Ex** : Enumération des huit premiers entiers (0 à 7).

```
?- entier(X).
X = 0 ;
X = s 0 ;
X = s s 0 ;
X = s s s 0 ;
X = s s s s 0 ;
X = s s s s s 0 ;
X = s s s s s s 0 ;
X = s s s s s s s 0
Yes
```

#### *II.4.1.2. Conversion entre systèmes*

Après avoir créer une représentation syntaxique, il est intéressant de pouvoir établir des liens avec d'autres systèmes.

**Programme 14 : Conversion unaire / décimal**

```
% Definition des entiers unaires
:-op(100,fy,s).
entier(0).
entier(s X):-entier(X).

% Conversion unaire / décimale
convU_D(0,0).
convU_D(s X,Y):-convU_D(X,Z),Y is Z+1.
```

**Ex** : Conversion unaire → décimal

```
?- convU_D(0,D).
D = 0
Yes
```

Dans notre système, `0` représente 0.

```
?- convU_D(s s s s s s s s s s s s 0,D).
D = 12
Yes
```

Dans notre système `s s s s s s s s s s s s 0` représente 12.

**Ex** : Conversion décimal → unaire

```
?- convU_D(U,0).
U = 0
Yes
```

0 se représente `0` dans notre système.

```
?- convU_D(U,12).
U = s s s s s s s s s s s s 0
Yes
```

12 se représente `s s s s s s s s s s s s 0` dans notre système .

### II.4.2. Définition d'opérateurs

Pour pouvoir utiliser cette nouvelle représentation, il faut définir de nouveaux opérateurs qui permettent de manipuler ces nombres.

**Programme 15 : Définition d'un système numérique complet (avec les opérateurs)**

```
% Définition des entiers
:-op(100,fy,s).
entier(0).
entier(s X):-entier(X).

% Conversion unaire / décimale
convU_D(0,0).
convU_D(s X,Y):-convU_D(X,Z),Y is Z+1.

% Addition
add(X,0,X):-entier(X).
add(X,s Y,s Z):-add(X,Y,Z).

% Multiplication
mult(X,0,0):-entier(X).
mult(X,s Y,Z):-mult(X,Y,T),add(X,T,Z).

% Factorielle
fact(0,s 0).
fact(s X,Y):-fact(X,Z),mult(s X,Z,Y).
```

**Ex** : L'addition

```
?- add(s s s 0, s s s s s s s 0,S).
S = s s s s s s s s s s 0
Yes
```

La somme de `s s s 0` et `s s s s s s s 0` vaut `s s s s s s s s s s 0`.

En effet, 3+7=10. Ce qu'on voit très clairement avec :

```
?- convU_D(X,3), convU_D(Y,7), add(X,Y,A), convU_D(A,Somme).
X = s s s 0
Y = s s s s s s s 0
```

```
A = s s s s s s s s s s 0
Somme = 10
Yes
```

**Ex** : La multiplication

```
?- mult(s s 0, s s s 0,Z).
Z = s s s s s s 0
Yes
```

2×6=6 donc la multiplication de `s s 0` pas `s s s 0` vaut `s s s s s s 0`

**Ex** : La factorielle

```
?- fact(s s s 0,Factorielle).
Factorielle = s s s s s s 0
Yes
```

3!=6 donc la factorielle de `s s s 0` est `s s s s s s 0`

**Rem** : Notre système de représentation n'est pas très performant. En effet, il consomme beaucoup de mémoire.

```
?- fact(s s s s s s s s s s s s s s s 0,Factorielle).
ERROR: Out of global stack
```

Ainsi, si notre système n'est pas mis en défaut sur le plan théorique, il est limité par les capacités de la machine sur laquelle on travaille.

A priori, nous n'avons programmé que les opérations d'addition, multiplication et factorielle. En réalité, nous avons fait beaucoup plus que ça puisque nous avons en même temps programmé les opérations inverses (sur les entiers naturels).

**Ex** : La soustraction est entièrement définie

```
?- add(s s 0,X,s s s s s 0).
X = s s s 0
Yes
```

On retrouve bien que 5-2=3.

**Ex** : La division a été partiellement programmée.

Pas de problème si le résultat est un entier :

```
?- mult(s s 0,X,s s s s 0).
X = s s 0
Yes
```

Par contre, si la division est à reste non nul PROLOG ne termine jamais

```
?- mult(s s 0,X,s s s s s 0).
Action (h for help) ? abort
% Execution Aborted
```

Cela est un comportement tout à fait logique sur l'ensemble des entiers naturels, cependant on aurait préféré que PROLOG s'arrête de lui-même.

**Ex** : L'inverse de la factorielle fonctionne de la même manière que la division

```
?- convU_D(F,120),fact(N,F).
F = s s s s s s s s s s...
N = s s s s s 0
Yes
```

En effet, 5!=120

```
?- fact(N,s s s s s 0).
Action (h for help) ? abort
% Execution Aborted
```

PROLOG boucle à l'infini car $\forall n \in \mathbb{N}\ n! \neq 5$ mais il ne le saura qu'après avoir essayé tous les n, or il y en a une infinité.

# III. La recherche de solution

## III.1. L'unification

Pour répondre aux requêtes qu'on peut lui poser PROLOG utilise un mécanisme appelé unification que nous allons définir ainsi que quelques autres éléments participant à ce mécanisme.

### III.1.1. Termes

Un programme PROLOG est constitué de trois types d'éléments différents :

- Des symboles de constantes
  **Ex** : `0`, `1`, `2002`…
  `louis13`, `superman`…
- Des symboles de fonctions
  **Ex** : `s`, `+`, `sqrt`…
- Des variables (tout ce qui commence par une majuscule)

  **Ex** : `X`, `FacExac`, `Qui`…

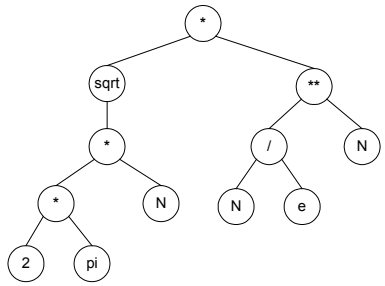
PROLOG traite ces éléments en les regroupant en « termes » (pour plus de renseignement sur les algèbres de termes, voir la première partie du cour de Génie Logiciel).

**Définition** :
- toute variable est un terme
- toute constante est un terme
- si f est un symbole de fonction à n arguments et si $t_1, t_2, \ldots t_n$ sont des termes alors $f(t_1, \ldots t_n)$ est un terme.

**Ex** : - `X`, `Qui`…
- `2`, `pi`, `louis13`…
- `2+pi`, `sqrt(2*pi*N)`, `f(X1,2)`…

En fait, il est possible de représenter n'importe quel terme par un arbre . Et inversement tout arbre ou sous-arbre est un terme.

**Ex** : $\sqrt{2\pi n} \times \left(\frac{n}{e}\right)^n$ ≡ `sqrt(2*2pi*N)*(N/e)**N` est un terme représentable par :




et `2*pi*N` (par ex) est aussi un terme

**Rem** : Un prédicat n'est pas un terme.

### III.1.2. Substitutions

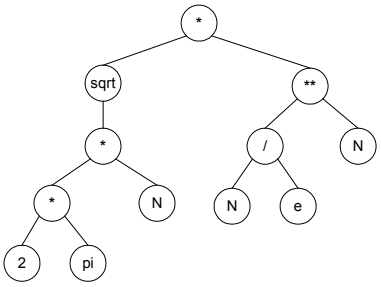
**Définition** :
Soient $Y_1, \ldots, Y_k$ k variables toutes différentes
$\tau_1, \ldots, \tau_k$ k termes différents ou non
Une substitution σ est l'association qui fait correspondre : $\tau_1$ à $Y_1$
…
$\tau_k$ à $Y_k$

**Rem** : Si on reprend la représentation d'un terme par un arbre, une substitution correspond à la greffe d'un arbre $\tau_i$ à chaque occurrence de $Y_i$.

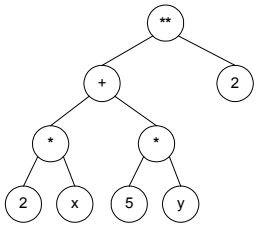
**Ex** : Soit la substitution σ : $n \rightarrow (2x+5y)^2$

Appliquons σ à $\sqrt{2\pi n} \times \left(\frac{n}{e}\right)^n$

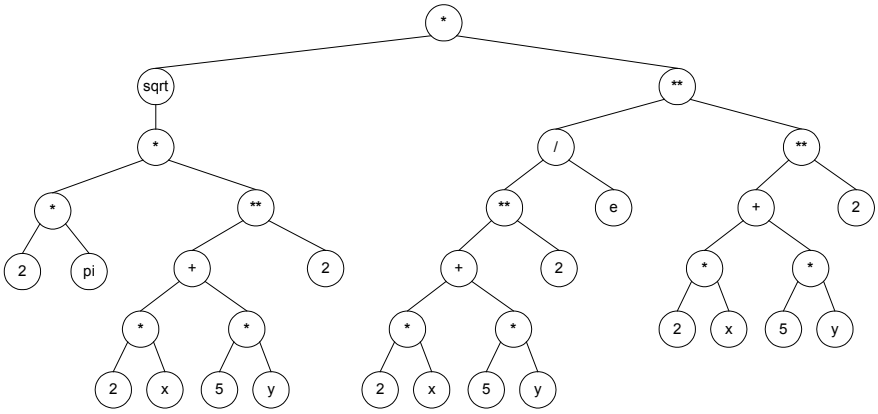
A partir des arbres




et




on obtient l'arbre :



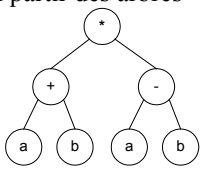

qui représente $\sqrt{2\pi(2x+5y)^2}\times\left(\frac{(2x+5y)^2}{e}\right)^{(2x+5y)^2}$

Lorsque la substitution agit sur plusieurs variables, tous les remplacements sont faits en parallèle..

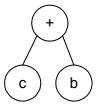
**Ex** : Soit la substitution σ : $\begin{cases} a \to c+b \\ b \to c-b \end{cases}$

Appliquons σ à $(a+b)\times(a-b)$

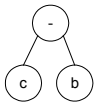
A partir des arbres



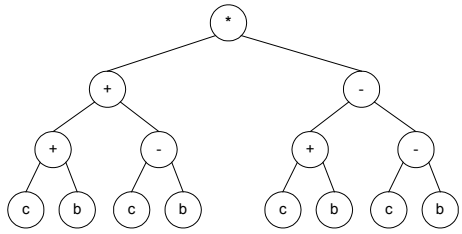

et




et



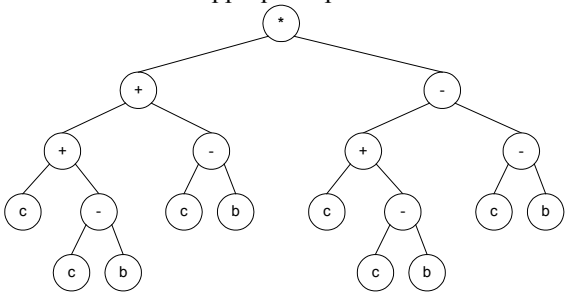

on obtient l'arbre :




qui représente $\big((c+b)+(c-b)\big)\times\big((c+b)-(c-b)\big)$

car les deux transformations sont faites en même temps.

Si on avait d'abord appliqué la première transformation puis la deuxième, on aurait obtenu l'arbre :




qui ne correspond pas à ce que l'on veut.

Enfin, il est possible de composer les substitutions

**Propriété** :
Soit X une variable
Soient σ et ρ deux substitutions telles que :

$$\sigma : \begin{cases} Y_1 \to \tau_1 \\ \vdots \\ Y_k \to \tau_k \end{cases} \quad \text{et} \quad \rho : \begin{cases} Z_1 \to \lambda_1 \\ \vdots \\ Z_k \to \lambda_k \end{cases}$$

Alors $\vartheta = \rho \circ \sigma$ est aussi une substitution telle que :

- si $X=Y_i$ alors $\vartheta(Y_i) = \rho(\sigma(Y_i))$
- si $X=Z_j$ tel que $Z_j \neq Y_i$ alors $\vartheta(Z_j) = \rho(Z_j)$
- si $\forall i,j$ $X \neq Y_i$ et $X \neq Z_j$ alors $\vartheta(X) = X$

### III.1.3. Unification

#### *III.1.3.1. Définition*

**Définition** :
Soient deux termes $t_1$ et $t_2$
Unifier $t_1$ et $t_2$ c'est trouver la substitution la plus générale qui permet de faire de $t_1$ et $t_2$ un seul et même terme.

**Théorème** :
Si deux termes sont unifiables, il existe une façon la plus générale pour les unifier.

**Rem** : En PROLOG l'unification se note « = ».

**Ex** : Unification directe de deux termes

```
?- f(a,Y)=f(Z,T).
Y = _G158
Z = a
T = _G158
Yes
```

Pour unifier ces deux termes, PROLOG tente d'unifier simultanément $\begin{cases} Z \leftrightarrow a \\ T \leftrightarrow Y \end{cases}$

Une solution possible est $Y = Z = T = a$ mais ce n'est pas la solution la plus générale. En effet, elle peut se déduire de la solution $\begin{cases} Z = a \\ T = Y \end{cases}$ qui est bien la plus générale comme le montre le résultat de PROLOG.

**Ex** : Unification directe entre deux termes impossible

```
?- f(a,Y)=g(Z,T).
No
```

L'unification n'est possible que si le nom des termes (c'est à dire leur racine dans la représentation en arbre) est le même pour les deux termes.

```
?- f(a1,a2)=f(X1,X2,X3).
No
```

L'unification est également impossible si les deux termes ont un nombre différent d'arguments.

**Ex** : Unification indirecte entre deux termes

```
?- X=f(a,Y),X=f(Z,T).
X = f(a, _G158)
Y = _G158
Z = a
T = _G158
Yes
```

Ici PROLOG ne peut pas 'voir' que cette requête est la même que celle du premier exemple. Il résout alors deux unifications en parallèle.
On retrouve bien la même solution qu'un peu plus haut.

**Ex** : Deux unifications directes simultanées

```
?- f(a,Y)=f(Z,T),h(b,Z)=h(Y,U).
Y = b
Z = a
T = b
```

```
U = a
Yes
```

Pour résoudre ces deux unifications simultanées PROLOG se ramène à quatre unifications :

$$\begin{cases} a \leftrightarrow Z \\ Y \leftrightarrow T \\ b \leftrightarrow Y \\ Z \leftrightarrow U \end{cases}$$

La solution la plus générale est, comme PROLOG l'a trouvée :

$$\begin{cases} Z = U = a \\ T = Y = b \end{cases}$$

#### *III.1.3.2. Algorithme de J. Herbraund*

Jacques Herbraund a établi un algorithme non déterministe pour unifier une paire (c'est à dire deux termes ou deux prédicats). Celui-ci s'appuie uniquement sur les six règles que nous allons détailler.

##### *III.1.3.2.1. Désaccord*

Soient $\forall i, x_i$ et $y_i$ des termes.

Si une paire est de la forme :

- $f(x_1, x_2, \ldots x_n) = g(y_1, y_2, \ldots y_n)$ avec f et g deux termes ou prédicats différents
- ou $f(x_1, x_2, \ldots x_n) = f(y_1, y_2, \ldots y_m)$ avec m ≠ n

L'unification est impossible et s'arrête donc sur un échec.

**Ex** :
```
?- f(X1,X2,X3)=g(Y1,Y2,Y3).
No
?- f(X1,X2,X3)=f(Y1,Y2,Y3,Y4).
No
```

##### *III.1.3.2.2. Eclatement*

Soient $\forall i, x_i$ et $y_i$ des termes.

Si une paire est de la forme $f(x_1, x_2, \ldots x_n) = f(y_1, y_2, \ldots y_n)$ on la remplace par les n paires :

$$\begin{cases} x_1 = y_1 \\ x_2 = y_2 \\ \quad \vdots \\ x_n = y_n \end{cases}$$

**Ex** :
```
?- f(X1,X2,X3)=f(Y1,Y2,Y3).
X1 = _G157
X2 = _G158
X3 = _G159
Y1 = _G157
Y2 = _G158
Y3 = _G159
Yes
```

##### *III.1.3.2.3. Suppression*

Si une paire est de la forme $t = t$ avec *t* un terme.

On la supprime car il n'y a rien à faire.

##### *III.1.3.2.4. Elimination d'une variable*

Soient $X$ une variable et $t$ un terme.

Si une paire est de la forme $X = t$ et que *X* ne figure pas dans *t*

- on la garde
- et partout on remplace $X$ par $t$

**Ex** :
```
?- X=f,Y=X.
X = f
Y = f
Yes
```

*III.1.3.2.5. Renversement*

Soient $X$ une variable et $t$ un terme.

Si une paire est de la forme $t = X$ on la remplace par $X = t$

**Ex** :
```
?- t=X.
X = t
Yes
```

*III.1.3.2.6. Test d'occurrence*

Soient $X$ une variable et $t$ un terme.

Si une paire est de la forme $X = t$ telle que

- $X$ figure dans $t$
- $t$ n'est pas réduit à $X$

On stoppe l'unification par un échec pour éviter une boucle infinie.

**Rem** : Cette sixième règle de l'algorithme de Herbraund n'est pas appliquée par défaut dans SWI-PROLOG© pour des raisons de performances.
```
?- X=t(X,J).
X = t(t(t(t(t(t(t(t(t(t(...), ...), _G158), _G158), _G158), _G158), _G158),
_G158), _G158), _G158), _G158)
J = _G158
Yes
```
Cependant, comme on peut le voir ci-dessus la version 5.0.2 sait s'arrêter et ne poursuit pas l'unification à l'infinie.

*III.1.3.2.7. Indéterminisme*

Comme nous l'avons dit cet algorithme est non déterministe. Ainsi, quand il est possible d'appliquer plusieurs règles, il faut toutes les appliquer et quelque soit l'ordre d'exécution, le résultat sera le même.

**Ex** : Essayons d'unifier $f(a,g(Y),h(X,g(a))) = f(X,g(h(a,X)),h(Z,g(X)))$

Première possibilité :

1) Eclatement : $\begin{cases} a = X \\ g(Y) = g(h(a,X)) \\ h(X,g(a)) = h(Z,g(X)) \end{cases}$

2) Renversement : $\begin{cases} X = a \\ g(Y) = g(h(a,X)) \\ h(X,g(a)) = h(Z,g(X)) \end{cases}$

3) Elimination : $\begin{cases} X = a \\ g(Y) = g(h(a,a)) \\ h(a,g(a)) = h(Z,g(a)) \end{cases}$

4) Eclatement : $\begin{cases} X = a \\ Y = h(a,a) \\ h(a,g(a)) = h(Z,g(a)) \end{cases}$

5) Eclatement : $\begin{cases} X = a \\ Y = h(a,a) \\ a = Z \\ g(a) = g(a) \end{cases}$

6) Suppression : $\begin{cases} X = a \\ Y = h(a,a) \\ a = Z \end{cases}$

7) Renversement : $\begin{cases} X = a \\ Y = h(a,a) \\ Z = a \end{cases}$

Autre possibilité :

1) Eclatement : $\begin{cases} a = X \\ g(Y) = g(h(a,X)) \\ h(X,g(a)) = h(Z,g(X)) \end{cases}$ 2) Renversement : $\begin{cases} X = a \\ g(Y) = g(h(a,X)) \\ h(X,g(a)) = h(Z,g(X)) \end{cases}$

3) Eclatement : $\begin{cases} X = a \\ g(Y) = g(h(a,X)) \\ X = Z \\ g(a) = g(X) \end{cases}$ 4) Elimination : $\begin{cases} X = a \\ g(Y) = g(h(a,a)) \\ a = Z \\ g(a) = g(a) \end{cases}$

5) Suppression : $\begin{cases} X = a \\ g(Y) = g(h(a,a)) \\ a = Z \end{cases}$ 6) Renversement : $\begin{cases} X = a \\ g(Y) = g(h(a,a)) \\ Z = a \end{cases}$

7) Eclatement : $\begin{cases} X = a \\ Y = h(a,a) \\ Z = a \end{cases}$

Il existe encore beaucoup d'autres possibilités qui mènent au même résultat.

```
?- f(a,g(Y),h(X,g(a)))=f(X,g(h(a,X)),h(Z,g(X))).
Y = h(a, a)
X = a
Z = a
Yes
```

**Rem** : Lorsque PROLOG est implémenté (par exemple SWI-PROLOG©), l'algorithme perd forcément son caractère non déterministe.

## III.2. Exécution d'un programme PROLOG

### III.2.1. L'interactivité par les requêtes

Un programme PROLOG est interactif. En effet, dans un langage comme le C, une fois le programme compilé l'utilisateur ne peut utiliser celui-ci que de la manière dont le programmeur l'a voulue. En PROLOG, les faits et les règles servent à définir le comportement général des outils mis à la disposition de l'utilisateur pour travailler mais ensuite ce dernier peut faire ce qu'il lui plait. Cette interactivité est obtenue par l'intermédiaire de requête que l'utilisateur soumet à l'interpréteur PROLOG.

#### *III.2.1.1. Des questions et des réponses*

Le déroulement d'un programme PROLOG est un jeu de questions / réponses : l'utilisateur interroge l'interpréteur PROLOG qui lui répond toujours au moins par oui ou non. Parfois, il instancie également les variables d'une requête s'il a trouvé une preuve (voir le chapitre III.2.2 ci-dessous) de validité de cette instanciation.

**Ex** : Avec le programme de base de données (page 3), nous avons demandé à PROLOG si Louis XIII a vécu entre 1601 et 1643 et est le fils de Henri IV et Marie de Médicis :

```
?- bio(louis13, h, 1601, 1643, henri4, marie_medicis).
```

Nous lui avons aussi demandé d'énumérer tous les personnages nés entre 1750 et 1800 :

```
?- bio(Qui,_,N,_,_,_),1750=<N,N=<1800.
```

Il faut noter que l'utilisateur peut poser toutes les requêtes qu'il désire en se servant des règles comme il l'entend.

**Ex** : Si on reprend l'exemple du système numérique unaire page 22, nous avons programmé une multiplication. A partir de là, l'utilisateur peut très bien utiliser ce prédicat pour faire une simple multiplication

```
?- mult(s s 0, s s s s 0, Resultat).
```

ou pour l'intégrer dans un calcul plus complexe :

```
?- convU_D(X,3),convU_D(Y,5),convU_D(Z,6),
   | fact(X,F), mult(Y,Z,M), add(F,M,R),
   | convU_D(R,Resultat).
```

S'il le veut, il peut même utiliser `mult` pour faire des divisions entières :

```
?- mult(s s 0, Div, s s s s s s 0).
```

Cependant, cette interactivité n'est pas sans risque. De fait, pour des questions de performances, PROLOG fait très peu de contrôles. Ainsi, si l'utilisateur ne sait pas ce qu'il fait, il peut facilement mettre l'interpréteur PROLOG en défaut notamment en rentrant dans une boucle infinie.

**Ex** : Toujours avec le programme du système unaire : si on utilise le prédicat `mult` pour effectuer une division avec reste, on entre dans une boucle infinie.

```
?- mult(s s 0, Div, s s s s s 0).
```

#### *III.2.1.2. Un programme évolutif*

Mais surtout là où l'interactivité de PROLOG est remarquable c'est qu'il est possible de modifier le programme pendant son exécution. En effet, PROLOG a été conçu à l'origine pour l'intelligence artificielle, un programme PROLOG doit donc être capable d'évoluer durant son existence. Cette évolution peut être automatisée ou forcée par l'utilisateur.

**Ex** : Dans le problème des régions adjacentes page 13, nous avons rentré de nouveaux faits afin définir trois coloriages pour tester les règles de décision. Pourtant ce procédé n'est pas très naturel : on voudrait pourvoir tester n'importe quels coloriages, même des coloriages imprévus. Pour cela, il faut utiliser le mot clef `assert`.
Considérons qu'on ait supprimé du programme page 13 les trois exemples de coloriage ; essayons de redéfinir les deux premiers en cours d'exécution et de les utiliser.
Définition de `coloriage1`

```
?- assert(color(1,bleu,coloriage1)),
 | assert(color(2,rouge,coloriage1)),
 | assert(color(3,vert,coloriage1)),
 | assert(color(4,jaune,coloriage1)),
 | assert(color(5,rouge,coloriage1)).
Yes
```

Test de conflit

```
?- conflit(coloriage1).
No
```

Comme nous l'avons vu au chapitre II.2.3, ce coloriage ne comporte aucune région adjacente de même couleur.
Définition de `coloriage2`

```
?- assert(color(1,vert,coloriage2)),
 | assert(color(2,rouge,coloriage2)),
 | assert(color(3,vert,coloriage2)),
 | assert(color(4,jaune,coloriage2)),
 | assert(color(5,vert,coloriage2)).
Yes
?- conflit(coloriage2).
Yes
?- conflit(X,Y,coloriage2),between(X,5,Y).
X = 1
Y = 3 ;
X = 1
Y = 5 ;
No
```

`coloriage2` possède encore deux fois deux régions adjacentes de même couleur.

### III.2.2. La recherche de preuves par l'unification

#### *III.2.2.1. Fonctionnement général*

##### *III.2.2.1.1. La logique*

Pour répondre à une requête, PROLOG cherche à prouver celle-ci par la logique.

En effet, nous avons vu au chapitre I.2.2 que chaque règle PROLOG `C:-H.` est interprétable par « H ⇒ C ». Donc pour démontrer C (la conclusion) PROLOG va essayer de démontrer H (les hypothèses). Si H est démontrable alors C l'est aussi, sinon il essaie avec une autre règle et si aucune règle ou fait ne permet de prouver C alors C est faux.

Ce procédé est itératif.

##### *III.2.2.1.2. L'implémentation*

**Rem** : On appelle but une conclusion que PROLOG cherche à démonter.
Le premier but qu'une requête est toujours la requête elle-même.

Pour trouver une preuve et afficher la solution associée, PROLOG utilise deux piles de piles :

- La pile des buts dans laquelle il stocke tous les niveaux de buts qu'il a cherché à atteindre. Un niveau de buts est lui-même représenté par une pile de buts qui contient toujours au moins un but sauf lorsqu'on a atteint la fin de la démonstration (on peut alors retourner le résultat en cas de succès ou `no` sinon). Une preuve est donc l'ensemble des buts par lesquels on est passé avant d'obtenir une pile de niveau de buts vide.
- La pile d'environnement est elle-aussi une pile de piles car à chaque niveau de buts est associée une pile d'environnement de niveau de buts. PROLOG stocke dans chacune de ces sous-piles les variables intervenues dans l'unification qui l'a amené à ce niveau. Dans la pile d'environnement, il conserve donc toutes les variables qui ont été utilisées pendant la démonstration. Elle lui sert ainsi à retrouver le résultat de la preuve en cas de succès.

Le changement de niveau de but correspond à un pas d'itération dans la démonstration logique (cf. chapitre précédent) et comme nous l'avons vu juste au-dessus, il est validé par une unification. En effet, pour prouver un but PROLOG :

- Soit trouve un fait pour lequel il existe une unification avec le but. Dans ce cas le but courant est démontrable et itérativement la requête est démontrable.
- Soit essaie d'appliquer la règle dont le membre gauche (la conclusion) est unifiable avec le but. S'il en existe une, alors il se fixe comme nouveau(x) but(s) la (ou les) formule(s) atomique(s) du membre droit de cette règle. Il passe ainsi à un autre niveau de but.

  Si plusieurs règles ont un membre gauche unifiable, PROLOG choisit la première dans le code du programme. Si celle-ci mène à une démonstration (succès), l'utilisateur peut demander une autre solution. Alors PROLOG remonte au dernier choix qu'il a du faire et applique la règle suivante s'il y en a encore une, sinon il remonte au choix précédent et s'il n'y a plus aucun choix possible il retourne `no`.

Ce fonctionnement en niveaux avec parfois plusieurs choix possibles à certains niveaux permet de qualifier l'exécution d'un programme PROLOG de parcours préfixe d'arbre. En effet, il est tout à fait possible, comme nous allons le voir, de représenté une telle exécution par un arbre.

#### *III.2.2.2. Exemples avec des opérations en unaire*

Pour la première série d'exemples du fonctionnement de PROLOG, nous allons utiliser une version modifiée du programme de définition du système unaire page 22.

**Programme 16 : Définition du système unaire avec répétitions**

```
% Définition des entiers
:-op(100,fy,s).
/* F1 */ entier(0).
/* R1 */ entier(s X):-entier(X).

% Addition
/* R2 */ add(X,0,X):-entier(X).
/* R3 */ add(0,X,X):-entier(X).                    /* Repetition maladroite */
/* R4 */ add(X,s Y,s Z):-add(X,Y,Z).
/* R5 */ add(s X,Y,s Z):-add(X,Y,Z).               /* Repetition maladroite */

% Multiplication
/* R2 */ mult(0,X,0).
/* R3 */ mult(X,0,0).                              /* Repetition maladroite */
/* R6 */ mult(X,s Y,Z):-mult(X,Y,T),add(X,T,Z).
```

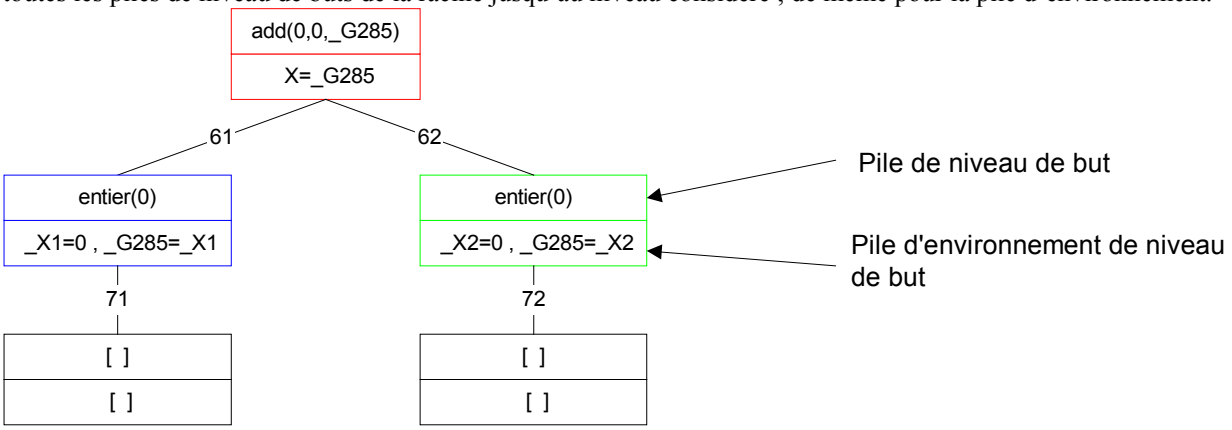
Dans ce programme ont été introduit quelques répétitions de sorte que PROLOG nous donne plusieurs réponses (identiques) à nos requêtes.

**Ex** : 0+0=0

```
[debug]  ?- add(0,0,X).
 T Call: (6) add(0, 0, _G285)
 T Call: (7) entier(0)
 T Exit: (7) entier(0)
 T Exit: (6) add(0, 0, 0)
X = 0 ;
 T Fail: (7) entier(0)
 T Redo: (6) add(0, 0, _G285)
```

```
 T Call: (7) entier(0)
 T Exit: (7) entier(0)
 T Exit: (6) add(0, 0, 0)
X = 0 ;
 T Fail: (7) entier(0)
 T Redo: (6) add(0, 0, _G285)
 T Fail: (6) add(0, 0, _G285)
No
```

PROLOG trouve deux fois la solution `0`. Essayons de comprendre ce qu'il fait.
Pour ça traçons l'arbre d'exécution. Chaque nœud de celui-ci correspond à une pile de niveau de buts et une pile d'environnement de niveau de buts. Pour connaître le contenu de la pile de buts à un moment donné, il suffit d'empiler toutes les piles de niveau de buts de la racine jusqu'au niveau considéré ; de même pour la pile d'environnement.



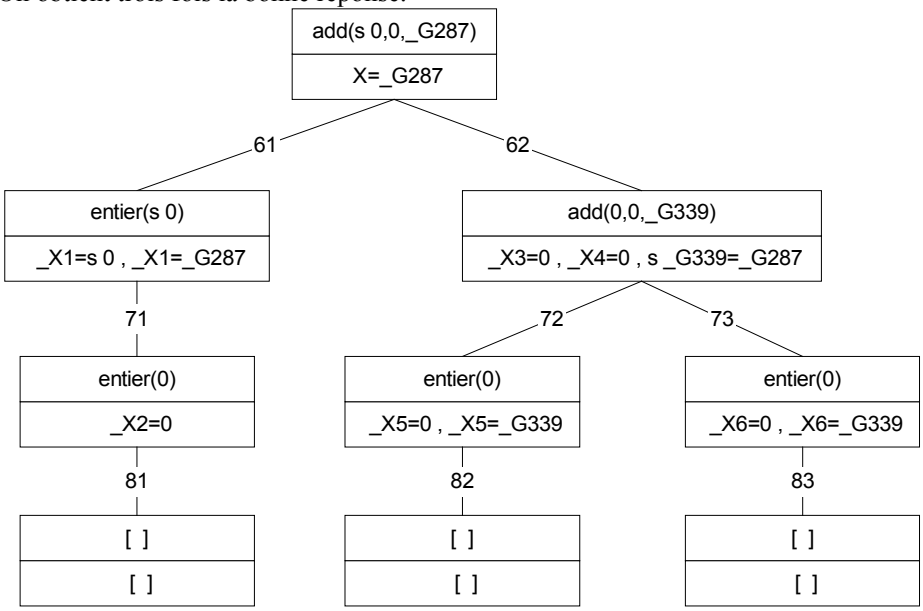

Regardons les étapes (dans l'ordre) de l'exécution du programme qui ont mené à un succès (seules celles-ci sont indiquées dans le traçage de PROLOG car seules celles-ci sont intéressantes) :
Transition 61 :
PROLOG unifie le membre gauche de la règle R2 `add(_X1,0,_X1):-entier(_X1).` (il utilise toujours de nouvelles variables) avec le but courant `add(0, 0, _G285)`.
L'unification réussit (`_X1=0` et `_G285=_X1`).
Il passe alors à un autre niveau de but ayant dans la pile de niveau de buts `entier(0)` et dans la pile d'environnement de niveau de buts le résultat de l'unification.
Transition 71 :
PROLOG unifie le but courant `entier(0)` avec le fait F1 `entier(0).`.
Un fait n'ayant pas de membre droit, le niveau de but suivant possède une pile de niveau de buts vide. Ce qui signifie qu'il a atteint une feuille de l'arbre, c'est à dire que la démonstration est finie.
Il remonte alors jusqu'à la racine pour trouver la solution de la requête : `X=_G285=_X1=0`
Transition 62 :
L'utilisateur demande un autre résultat. PROLOG revient donc au niveau de buts où il a fait sa dernière unification (représenté dans l'arbre ci-dessus par le nœud bleu) et cherche à unifier ce but avec un autre fait / règle mais cela n'est plus possible (⇒`T Fail: (7) entier(0)` dans la trace). Il remonte donc d'un niveau (revient à la racine en rouge) et tente une nouvelle unification. Cette fois-ci il y en a une et il unifie le but courant `add(0, 0, _G285)` avec le membre gauche de la règle R3 `add(0,_X2,_X2):-entier(_X2).`
Transition 72 :
Identique à la transition 71.
Fin :
L'utilisateur redemande encore un autre résultat. PROLOG revient au niveau de buts où il a fait sa dernière unification (colorié en vert). Cependant, il n'est plus possible d'unifier `entier(0)` avec aucune autre règle ou fait. Alors il remonte d'un niveau et se retrouve dans la racine (rouge), mais `add(0, 0, _G285)` n'est également plus unifiable. Donc ce but n'est pas démontrable. Donc il est faux. Alors PROLOG retourne `no`.

**Ex** : 1+0=1

```
[debug]  ?- add(s 0,0,X).
 T Call: (6) add(s 0, 0, _G287)
 T Call: (7) entier(s 0)
 T Call: (8) entier(0)
 T Exit: (8) entier(0)
 T Exit: (7) entier(s 0)
 T Exit: (6) add(s 0, 0, s 0)
X = s 0 ;
 T Fail: (8) entier(0)
 T Fail: (7) entier(s 0)
```

```
 T Redo: (6) add(s 0, 0, _G287)
 T Call: (7) add(0, 0, _G339)
 T Call: (8) entier(0)
 T Exit: (8) entier(0)
 T Exit: (7) add(0, 0, 0)
 T Exit: (6) add(s 0, 0, s 0)
X = s 0 ;
 T Fail: (8) entier(0)
 T Redo: (7) add(0, 0, _G339)
 T Call: (8) entier(0)
 T Exit: (8) entier(0)
 T Exit: (7) add(0, 0, 0)
 T Exit: (6) add(s 0, 0, s 0)
X = s 0 ;
 T Fail: (8) entier(0)
 T Redo: (7) add(0, 0, _G339)
 T Fail: (7) add(0, 0, _G339)
 T Fail: (6) add(s 0, 0, _G287)
No
```

On obtient trois fois la bonne réponse.




Transition 61 :
règle : `add(_X1,0,_X1):-entier(_X1).`
unification : `add(_X1,0,_X1)=add(s 0,0,_G287)`
résultat : `_X1=s 0` et `_X1=_G287`
Transition 71 :
règle : `entier(s _X2):-entier(_X2).`
unification : `entier(s _X2)= entier(s 0)`
résultat : `_X2=0`
Transition 81 :
fait : `entier(0).`
succès : `X=_G287=_X1=0`
Transition 62 :
règle : `add(s _X3,_X4,s _G339):-add(_X3,_X4,_G339).`
unification : `add(s _X3,_X4,s _G339)=add(s 0,0,_G287)`
résultat : `_X3=0` , `_X4=0` et `s_G339=_G287`
Transition 72 :
règle : `add(_X5,0,_X5):-entier(_X5).`
unification : `add(_X5,0,_X5)= add(0,0,_G339)`
résultat : `_X5=0` et `_X5=_G339`
Transition 82 :
fait : `entier(0).`
succès : `X = _G287 = s _G339 = s _X5 = s 0`
Transition 73 :
règle : `add(0,_X6,_X6):-entier(_X6).`

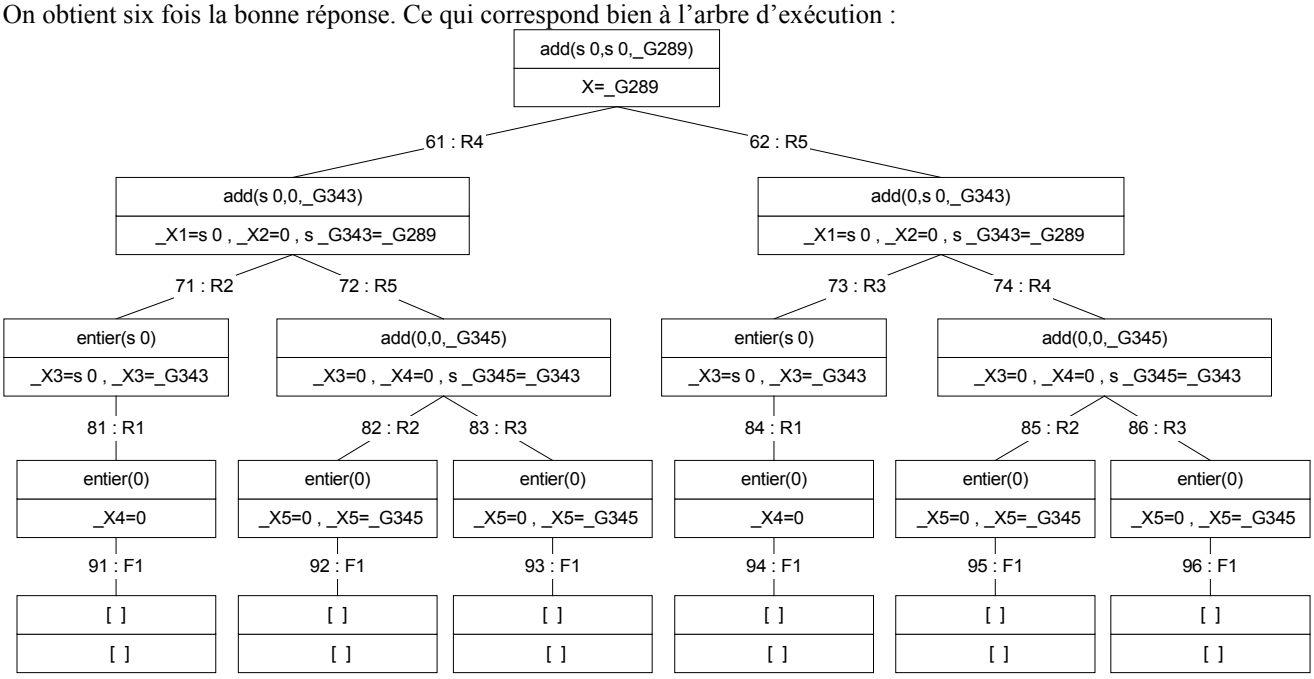
unification : `add(0,_X6,_X6)= add(0,0,_G339)`
résultat : `_X6=0` et `_X6=_G339`
Transition 83 :
fait : `entier(0).`
succès : `X = _G287 = s _G339 = s _X6 = s 0`

**Ex** : 1+1=2

```
[debug]  ?- add(s 0,s 0,X).
 T Call: (6) add(s 0, s 0, _G289)
 T Call: (7) add(s 0, 0, _G343)
 T Call: (8) entier(s 0)
 T Call: (9) entier(0)
 T Exit: (9) entier(0)
 T Exit: (8) entier(s 0)
 T Exit: (7) add(s 0, 0, s 0)
 T Exit: (6) add(s 0, s 0, s s 0)
X = s s 0 ;
 T Fail: (9) entier(0)
 T Fail: (8) entier(s 0)
 T Redo: (7) add(s 0, 0, _G343)
 T Call: (8) add(0, 0, _G345)
 T Call: (9) entier(0)
 T Exit: (9) entier(0)
 T Exit: (8) add(0, 0, 0)
 T Exit: (7) add(s 0, 0, s 0)
 T Exit: (6) add(s 0, s 0, s s 0)
X = s s 0 ;
 T Fail: (9) entier(0)
 T Redo: (8) add(0, 0, _G345)
 T Call: (9) entier(0)
 T Exit: (9) entier(0)
 T Exit: (8) add(0, 0, 0)
 T Exit: (7) add(s 0, 0, s 0)
 T Exit: (6) add(s 0, s 0, s s 0)
X = s s 0 ;
 T Fail: (9) entier(0)
 T Redo: (8) add(0, 0, _G345)
 T Fail: (8) add(0, 0, _G345)
 T Fail: (7) add(s 0, 0, _G343)
 T Redo: (6) add(s 0, s 0, _G289)
 T Call: (7) add(0, s 0, _G343)
 T Call: (8) entier(s 0)
 T Call: (9) entier(0)
 T Exit: (9) entier(0)
 T Exit: (8) entier(s 0)
 T Exit: (7) add(0, s 0, s 0)
 T Exit: (6) add(s 0, s 0, s s 0)
X = s s 0 ;
 T Fail: (9) entier(0)
 T Fail: (8) entier(s 0)
 T Redo: (7) add(0, s 0, _G343)
 T Call: (8) add(0, 0, _G345)
 T Call: (9) entier(0)
 T Exit: (9) entier(0)
 T Exit: (8) add(0, 0, 0)
 T Exit: (7) add(0, s 0, s 0)
 T Exit: (6) add(s 0, s 0, s s 0)
X = s s 0 ;
 T Fail: (9) entier(0)
 T Redo: (8) add(0, 0, _G345)
 T Call: (9) entier(0)
 T Exit: (9) entier(0)
 T Exit: (8) add(0, 0, 0)
 T Exit: (7) add(0, s 0, s 0)
 T Exit: (6) add(s 0, s 0, s s 0)
```

```
X = s s 0 ;
 T Fail: (9) entier(0)
 T Redo: (8) add(0, 0, _G345)
 T Fail: (8) add(0, 0, _G345)
 T Fail: (7) add(0, s 0, _G343)
 T Fail: (6) add(s 0, s 0, _G289)
No
```

Rem : le traçage du prédicat entier ayant peu d'intérêt, il ne sera plus fait par la suite.
On obtient six fois la bonne réponse. Ce qui correspond bien à l'arbre d'exécution :




Dans les transitions, on a noté la règle ou le fait avec lequel il y a eu unification pour passer à un nouveau niveau de buts. Par exemple, pour la transition 11, on a utilisé la règle : `add(_X1,s _X2,s _G343):-add(_X1,_X2,_G343).` (R4) ce qui a donné lieu à l'unification : `add(_X1,s _X2,s _G343)=add(s 0,s 0,_G289)` dont le résultat est : `_X1=s 0` , `_X2=0` et `s _G343=_G289`

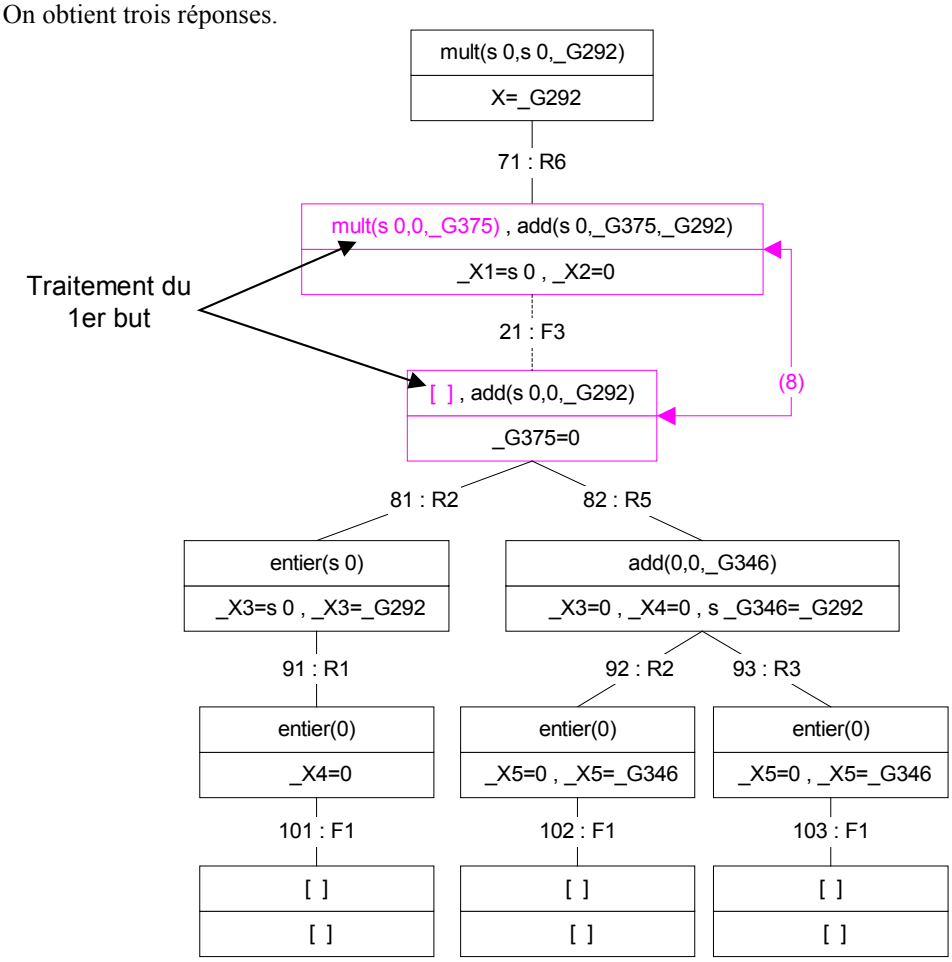
Jusqu'ici, nous n'avons vu que des piles de niveau de buts internes (c'est à dire qui ne sont pas des feuilles) ayant un seul élément. Cela vient du fait que nous n'avons employé que des règles dont le membre droit est réduit à une seule formule atomique et que toutes nos requêtes étaient aussi restreintes à une seule formule atomique. Cependant, ceci n'a rien d'obligatoire.

**Ex** : 1×1=1

```
[debug]  ?- mult(s 0,s 0,X).
 T Call: (7) mult(s 0, s 0, _G292)
 T Call: (8) mult(s 0, 0, _G375)
 T Exit: (8) mult(s 0, 0, 0)
 T Call: (8) add(s 0, 0, _G292)
 T Exit: (8) add(s 0, 0, s 0)
 T Exit: (7) mult(s 0, s 0, s 0)
X = s 0 ;
 T Redo: (8) add(s 0, 0, _G292)
 T Call: (9) add(0, 0, _G346)
 T Exit: (9) add(0, 0, 0)
 T Exit: (8) add(s 0, 0, s 0)
 T Exit: (7) mult(s 0, s 0, s 0)
X = s 0 ;
 T Redo: (9) add(0, 0, _G346)
 T Exit: (9) add(0, 0, 0)
 T Exit: (8) add(s 0, 0, s 0)
 T Exit: (7) mult(s 0, s 0, s 0)
X = s 0 ;
 T Redo: (9) add(0, 0, _G346)
 T Fail: (9) add(0, 0, _G346)
 T Fail: (8) add(s 0, 0, _G292)
 T Redo: (8) mult(s 0, 0, _G375)
 T Fail: (8) mult(s 0, 0, _G375)
```

```
 T Fail: (7) mult(s 0, s 0, _G292)
No
```

On obtient trois réponses.



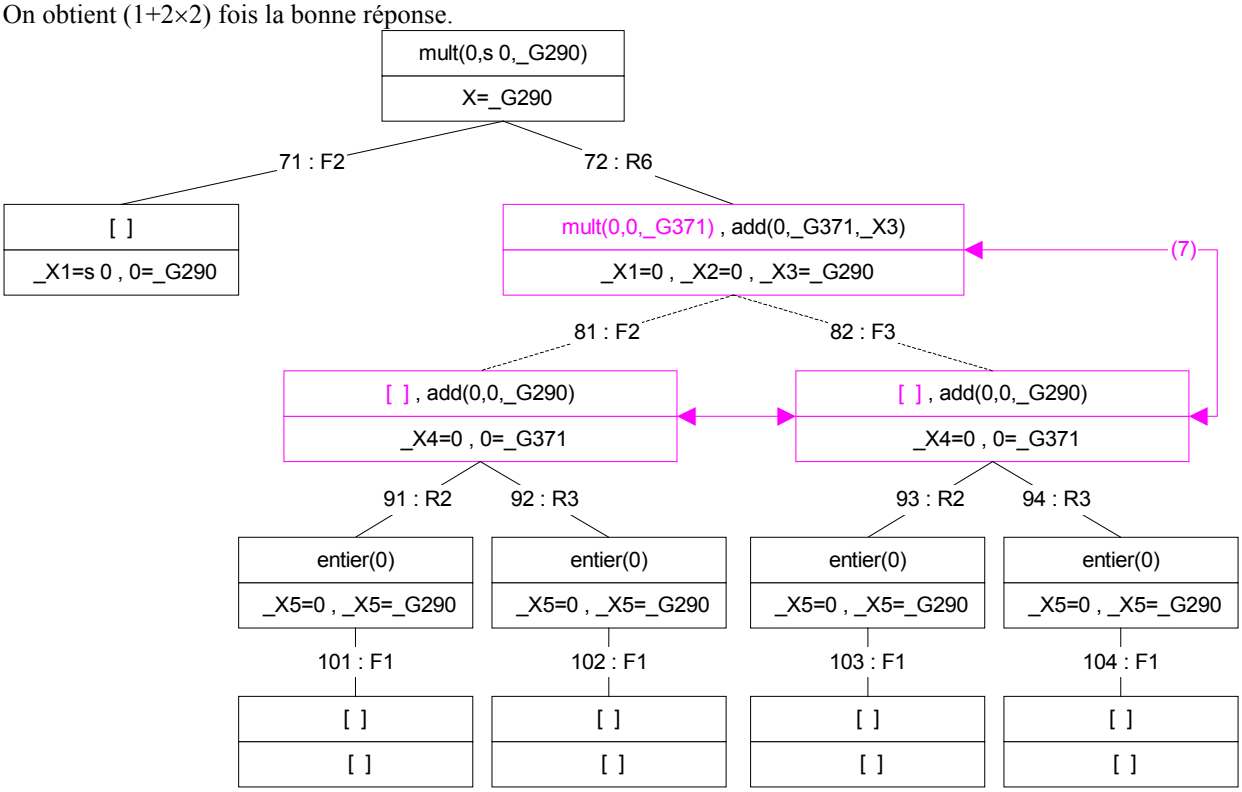

Le fait qu'une la pile de niveau de buts soit constituée de plus d'un but pose un problème dans la représentation arborescente de l'exécution du programme. En effet, jusqu'ici une transition représentait une unification et un changement de niveau de buts ; mais lorsqu'une pile de niveau de buts contient plusieurs buts, il existe toujours au moins une unification qui ramène la démonstration au niveau de buts qui contenait plusieurs but. C'est pour cela qu'en mode traçage, on voit que les appels de `mult(s 0, 0, _G375)` et `add(s 0, 0, _G292)` sont tous les deux faits au même niveau (8). Cela vient du fait que pour traiter tous les buts de ce niveau PROLOG essaie de les démontrer les uns après les autres et lorsqu'il a fait la preuve d'un but il revient à ce niveau pour passer au but suivant.
Pour lever toute ambiguïté, nous représenterons en pointillé les transitions représentant une unification qui ne mène pas à un nouveau niveau de buts. De plus, nous relierons les nœuds de l'arbre faisant parti du même niveau de buts (cela se révèlera plus utile dans les exemples suivants que dans celui-ci). Enfin, le but traité sera mis en couleur.

**Ex** : 0×1=0

```
[debug]  ?- mult(0,s 0,X).
 T Call: (7) mult(0, s 0, _G290)
 T Exit: (7) mult(0, s 0, 0)
X = 0 ;
 T Redo: (7) mult(0, s 0, _G290)
 T Call: (8) mult(0, 0, _G371)
 T Exit: (8) mult(0, 0, 0)
 T Call: (8) add(0, 0, _G290)
 T Exit: (8) add(0, 0, 0)
 T Exit: (7) mult(0, s 0, 0)
X = 0 ;
 T Redo: (8) add(0, 0, _G290)
 T Exit: (8) add(0, 0, 0)
 T Exit: (7) mult(0, s 0, 0)
X = 0 ;
 T Redo: (8) add(0, 0, _G290)
 T Fail: (8) add(0, 0, _G290)
 T Redo: (8) mult(0, 0, _G371)
 T Exit: (8) mult(0, 0, 0)
 T Call: (8) add(0, 0, _G290)
```

```
 T Exit: (8) add(0, 0, 0)
 T Exit: (7) mult(0, s 0, 0)
X = 0 ;
 T Redo: (8) add(0, 0, _G290)
 T Exit: (8) add(0, 0, 0)
 T Exit: (7) mult(0, s 0, 0)
X = 0 ;
 T Redo: (8) add(0, 0, _G290)
 T Fail: (8) add(0, 0, _G290)
 T Redo: (8) mult(0, 0, _G371)
 T Fail: (8) mult(0, 0, _G371)
 T Fail: (7) mult(0, s 0, _G290)
No
```

On obtient (1+2×2) fois la bonne réponse.



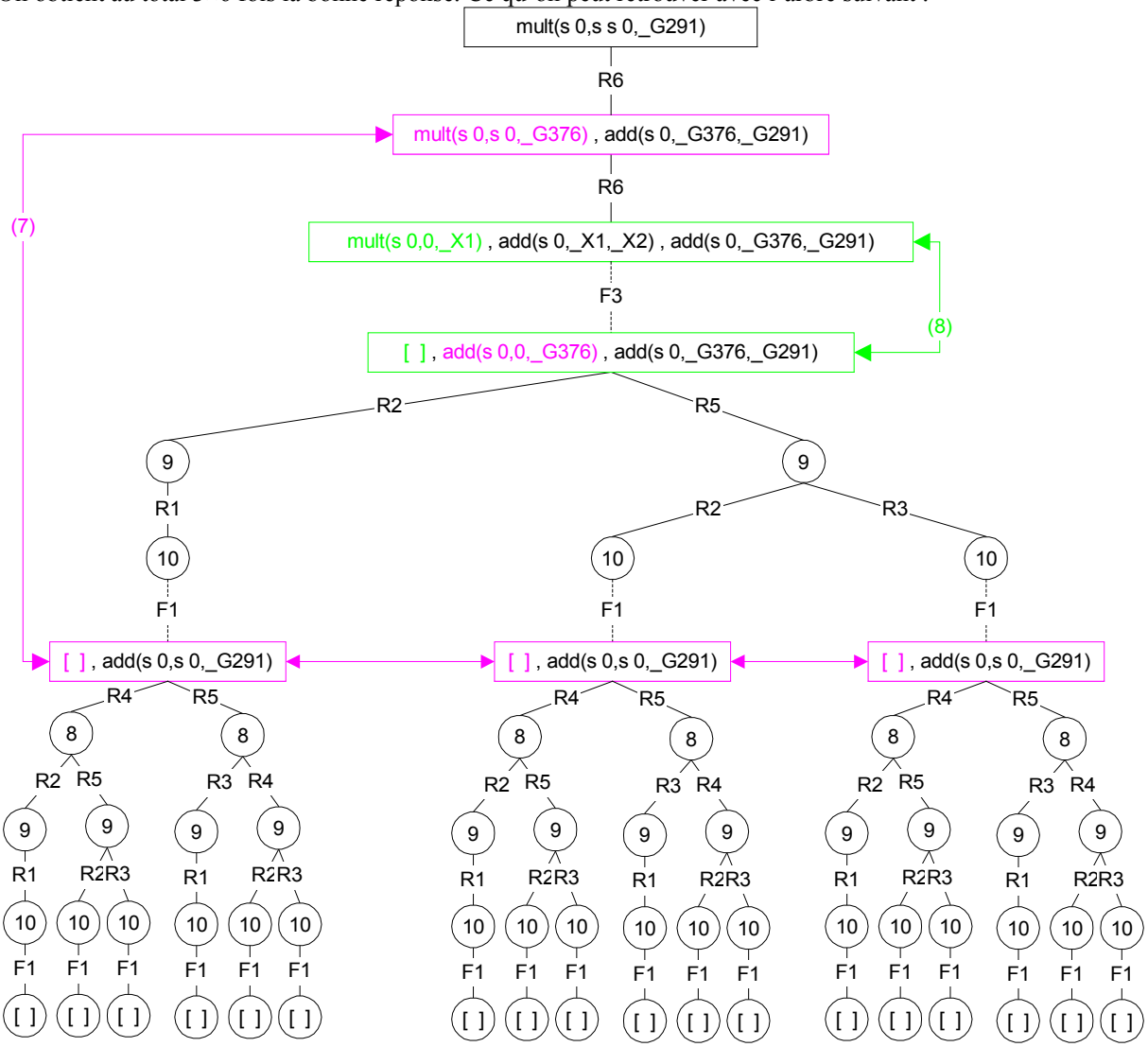

Il existe deux façons de prouver `mult(0, 0, _G371)`. Donc `add(0,_G371,_X3)` peut être appelé avec deux séries d'arguments différentes (même si ici les deux séries sont identiques).

**Ex** : 1×2=2

```
[debug]  ?- mult(s 0,s s 0,X).
 T Call: (6) mult(s 0, s s 0, _G291)
 T Call: (7) mult(s 0, s 0, _G376)
 T Call: (8) mult(s 0, 0, _G376)
 T Exit: (8) mult(s 0, 0, 0)
 T Call: (8) add(s 0, 0, _G376)
 T Call: (9) entier(s 0)
 T Call: (10) entier(0)
 T Exit: (10) entier(0)
 T Exit: (9) entier(s 0)
 T Exit: (8) add(s 0, 0, s 0)
 T Exit: (7) mult(s 0, s 0, s 0)
 T Call: (7) add(s 0, s 0, _G291)
 T Call: (8) add(s 0, 0, _G347)
 T Call: (9) entier(s 0)
 T Call: (10) entier(0)
 T Exit: (10) entier(0)
 T Exit: (9) entier(s 0)
 T Exit: (8) add(s 0, 0, s 0)
 T Exit: (7) add(s 0, s 0, s s 0)
 T Exit: (6) mult(s 0, s s 0, s s 0)
X = s s 0 ;
```

. . .

On obtient au total 3×6 fois la bonne réponse. Ce qu'on peut retrouver avec l'arbre suivant :




L'arbre à été simplifié pour ne montrer que les piles de buts ayant plusieurs buts. Dans les autres nœuds de l'arbre n'est indiqué que le niveau de buts du nœud.

On observe très bien dans cet exemple le 'retour en arrière' de certaines transitions.

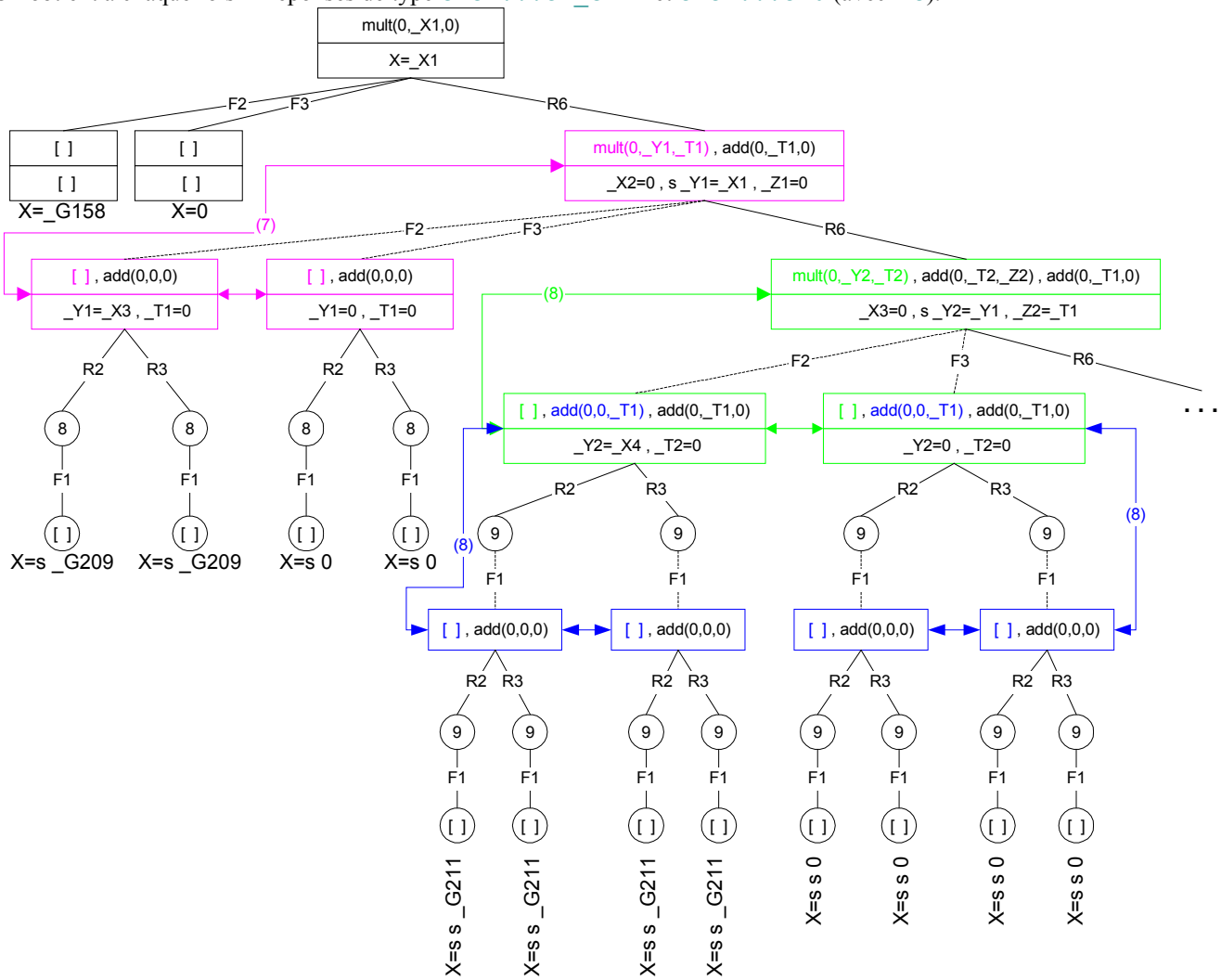
Pour terminer avec cet exemple, observons un cas de boucle infinie.

**Ex** : Enumérer tous les entiers X tel que X×0=0

```
?- mult(0,X,0).
X = _G158 ;
X = 0 ;
X = s _G209 ;
X = s _G209 ;
X = s 0 ;
X = s 0 ;
X = s s _G211 ;
X = s s _G211 ;
X = s s _G211 ;
X = s s _G211 ;
X = s s 0 ;
X = s s 0 ;
X = s s 0 ;
X = s s 0 ;
X = s s s _G213 ;
X = s s s _G213 ;
X = s s s _G213 ;
X = s s s _G213 ;
X = s s s _G213 ;
X = s s s _G213 ;
```

```
X = s s s _G213 ;
X = s s s _G213 ;
X = s s s 0
...
```

On obtient à chaque fois $2^n$ réponses de type `s s ...s _G2xx` et `s s ...s 0` (avec n `s`).




S'il y a à chaque fois deux types de réponses (celles qui se terminent par `0` et celles qui se terminent par `_Gxxx`) cela est dû aux faits F2 et F3 qui sont deux façons possibles de prouver les buts `mult(0,X,Y)`.
Ensuite, si les réponses `s s ...s 0` (avec n `s`) sont répétées n fois, c'est parce que chaque but `add(0,X,Y)` multiplie par deux les possibilités de démonstration ; or à chaque fois qu'on applique la règle R6, on introduit un but de ce type supplémentaire dans la pile de niveau de buts.

#### *III.2.2.3. Exemples avec des listes*

##### *III.2.2.3.1. Propriétés des listes*

En PROLOG, les listes sont prédéfinies. Elles sont notées entre `[` et `]`. Leurs éléments sont séparés par des `,`.

**Ex** : `[]` est la liste vide
`[a,b,c]` est une liste non vide
`[[a,b,c],12 ,X,[f,g]]` est une autre liste contenant des sous-listes et une variable.

Elles sont toujours constituées d'une tête et d'une queue. La tête est toujours un et un seul élément (qui peut éventuellement être une autre liste). La queue est toujours une liste (éventuellement vide).

Le symbole `|` permet d'unir ou séparer ces deux parties.

**Ex** : Concaténation d'un élément en tête d'une liste :
`[a|[c,d,e]] is [a,b,c,d,e]`

**Ex** : Extraction de la tête ou de la queue d'une liste :

```
?- [a,b,c,d,e]=[H|T].
H = a
```

```
T = [b, c, d, e]
Yes
```

Cas d'une liste où la tête est une autre liste :

```
?- [[a,b,c,d],e]=[H|T].
H = [a, b, c, d]
T = [e]
Yes
```

Cas d'une queue vide :

```
?- [a]=[H|T].
H = a
T = []
Yes
```

Il est possible grâce à l'unification d'extraire d'une liste n'importe quel élément. Cependant, toutes les unifications ne sont pas forcément valides.

**Ex** : Cas d'unifications impossibles :

```
?- []=[H|T].
No
```

La liste vide n'a pas de tête.

```
?- [[a,b],[c,d,e]]=[[H1|T1]|[H2|[H3|T3]]].
No
```

Essayons de faire l'unification à la main pour comprendre où est l'erreur.

$$1)\begin{cases}[a,b]=[H1|T1] \\ [[c,d,e]]=[H2|[H3|T3]]\end{cases} \qquad 2)\begin{cases}H1=a \\ T1=[b] \\ H2=[c,d,e] \\ []=[H3|T3]\end{cases}$$

3) Nous avons vu juste au-dessus que la liste vide n'avait pas de tête donc l'unification `[]=[H3|T3]` n'a pas de solution.

**Ex** : Recherche d'éléments dans une liste

La liste contenant la liste vide possède une tête et une queue :

```
?- [[]]=[Tete|Queue].
Tete = []
Queue = []
Yes
```

La queue est toujours incorporée dans une liste :

```
?- [0,[a,b,c]]=[Tete|Queue].
Tete = 0
Queue = [[a, b, c]]
Yes
```

La queue est donc ici une liste ayant pour seul élément une autre liste à trois éléments. On peut récupérer cette sous-liste par la requête suivante :

```
?- [0,[a,b,c]]=[Tete|[SousListe|Fin]].
Tete = 0
SousListe = [a, b, c]
Fin = []
Yes
```

Récupération de tous les éléments d'une queue :

```
?- [[a,b,c],[d,e,f],g]=[Tete|[[D|[E|[F|T1]]]|[G|T2]]].
Tete = [a, b, c]
D = d
E = e
F = f
T1 = []
G = g
T2 = []
Yes
```

#### *III.2.2.3.2. Exemples*

Afin de pouvoir observer le fonctionnement de PROLOG avec les listes, nous allons utiliser un programme qui permet de vérifier l'appartenance d'un élément à une liste.

**Programme 17 : Appartenance à une liste (1ère version)**

```
/* F1 */ belong(X,[X|L]).
/* R1 */ belong(X,[T|L]):-belong(X,L).
```

**Rem** : il existe déjà un prédicat prédéfini en PROLOG qui permet cela : `member`.

Dans un premier temps, ce programme permet effectivement de vérifier si un élément appartient à une liste.

**Ex** : Etre ou ne pas être dans la liste…

```
?- belong(a,[q,d,a,f,d]).
Yes
?- belong(a,[q,d,f,d]).
No
```

**Ex** : Nombre d'occurrences d'un élément.

```
?- belong(X,[1,2,3,2,3,3]),X=1.
X = 1 ;
No
?- belong(X,[1,2,3,2,3,3]),X=3.
X = 3 ;
X = 3 ;
X = 3 ;
No
?- belong(X,[1,2,3,2,3,3]),X=0.
No
```

On obtient autant de réponses qu'il y a d'occurrences de l'élément recherché.

On peut également demander une énumération des éléments de la liste.

**Ex** :

```
?- belong(X,[e,n,u,m,e,r,a,t,i,o,n]).
X = e ;
X = n ;
X = u ;
X = m ;
X = e ;
X = r ;
X = a ;
X = t ;
X = i ;
X = o ;
X = n ;
No
```

Enfin, il est également possible de construire des listes particulières.

**Ex** : Construction de toutes les listes contenant un 'a' :

```
?- belong(a,L).
L = [a|_G248] ;
L = [_G248, a|_G252] ;
L = [_G248, _G251, a|_G255] ;
L = [_G248, _G251, _G254, a|_G258] ;
L = [_G248, _G251, _G254, _G257, a|_G261] ;
L = [_G248, _G251, _G254, _G257, _G260, a|_G264] ;
L = [_G248, _G251, _G254, _G257, _G260, _G263, a|_G267]
...
```

PROLOG énonce toutes les solutions qu'il peut démontrer. Or il y en a une infinité, donc il ne s'arrête jamais.

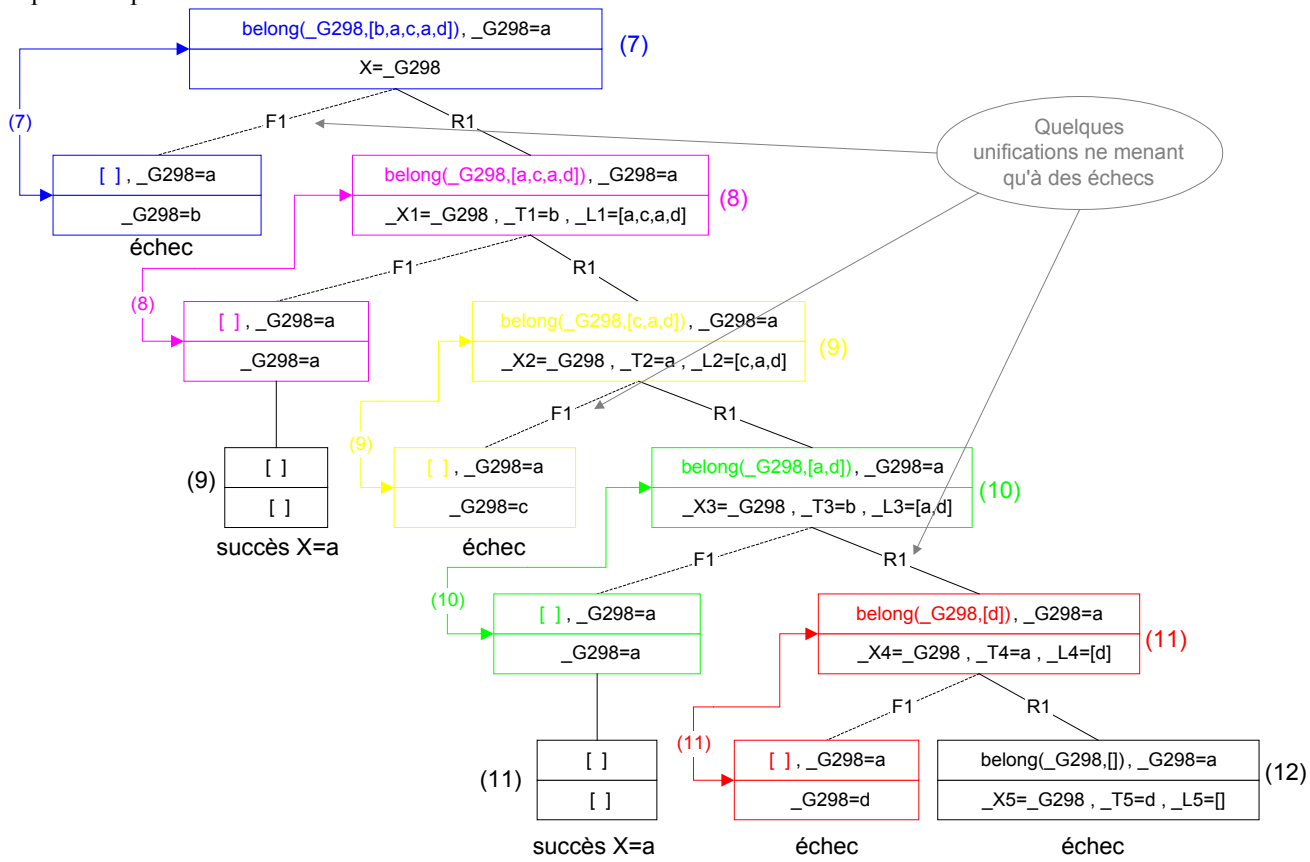
Maintenant que nous avons vu ce que pouvait faire ce programme, regardons son exécution plus en détail.

**Ex** : Nombre d'occurrences d'un élément.

```
[debug]  ?- belong(X,[b,a,c,a,d]),X=a.
 T Call: (7) belong(_G298, [b, a, c, a, d])
 T Exit: (7) belong(b, [b, a, c, a, d])
 T Redo: (7) belong(_G298, [b, a, c, a, d])
 T Call: (8) belong(_G298, [a, c, a, d])
 T Exit: (8) belong(a, [a, c, a, d])
 T Exit: (7) belong(a, [b, a, c, a, d])
X = a ;
 T Redo: (8) belong(_G298, [a, c, a, d])
 T Call: (9) belong(_G298, [c, a, d])
```

```
 T Exit: (9) belong(c, [c, a, d])
 T Exit: (8) belong(c, [a, c, a, d])
 T Exit: (7) belong(c, [b, a, c, a, d])
 T Redo: (9) belong(_G298, [c, a, d])
 T Call: (10) belong(_G298, [a, d])
 T Exit: (10) belong(a, [a, d])
 T Exit: (9) belong(a, [c, a, d])
 T Exit: (8) belong(a, [a, c, a, d])
 T Exit: (7) belong(a, [b, a, c, a, d])
X = a ;
 T Redo: (10) belong(_G298, [a, d])
 T Call: (11) belong(_G298, [d])
 T Exit: (11) belong(d, [d])
 T Exit: (10) belong(d, [a, d])
 T Exit: (9) belong(d, [c, a, d])
 T Exit: (8) belong(d, [a, c, a, d])
 T Exit: (7) belong(d, [b, a, c, a, d])
 T Redo: (11) belong(_G298, [d])
 T Call: (12) belong(_G298, [])
 T Fail: (12) belong(_G298, [])
 T Fail: (11) belong(_G298, [d])
 T Fail: (10) belong(_G298, [a, d])
 T Fail: (9) belong(_G298, [c, a, d])
 T Fail: (8) belong(_G298, [a, c, a, d])
 T Fail: (7) belong(_G298, [b, a, c, a, d])
No
```

Ce qui correspond à l'arbre d'exécution :




Sur cet arbre, on peut observer deux nouveaux points qu'on n'avait pas remarqués avec les exemples du calcul unaire. Tout d'abord, même si on en avait parlé, on n'avait pas vu de cas de requête à plusieurs buts. Cependant le traitement d'une requête ayant plusieurs buts n'apporte aucune difficulté supplémentaire par rapport à un niveau de but ayant plusieurs buts comme on en a déjà vu dans plusieurs exemples.

Ensuite, jusqu'ici toutes les unifications que nous faisions menaient à au moins un succès. Cet exemple nous montre que cela n'est pas une obligation. Donc :

démonstration réussie ⇒ unification réussie

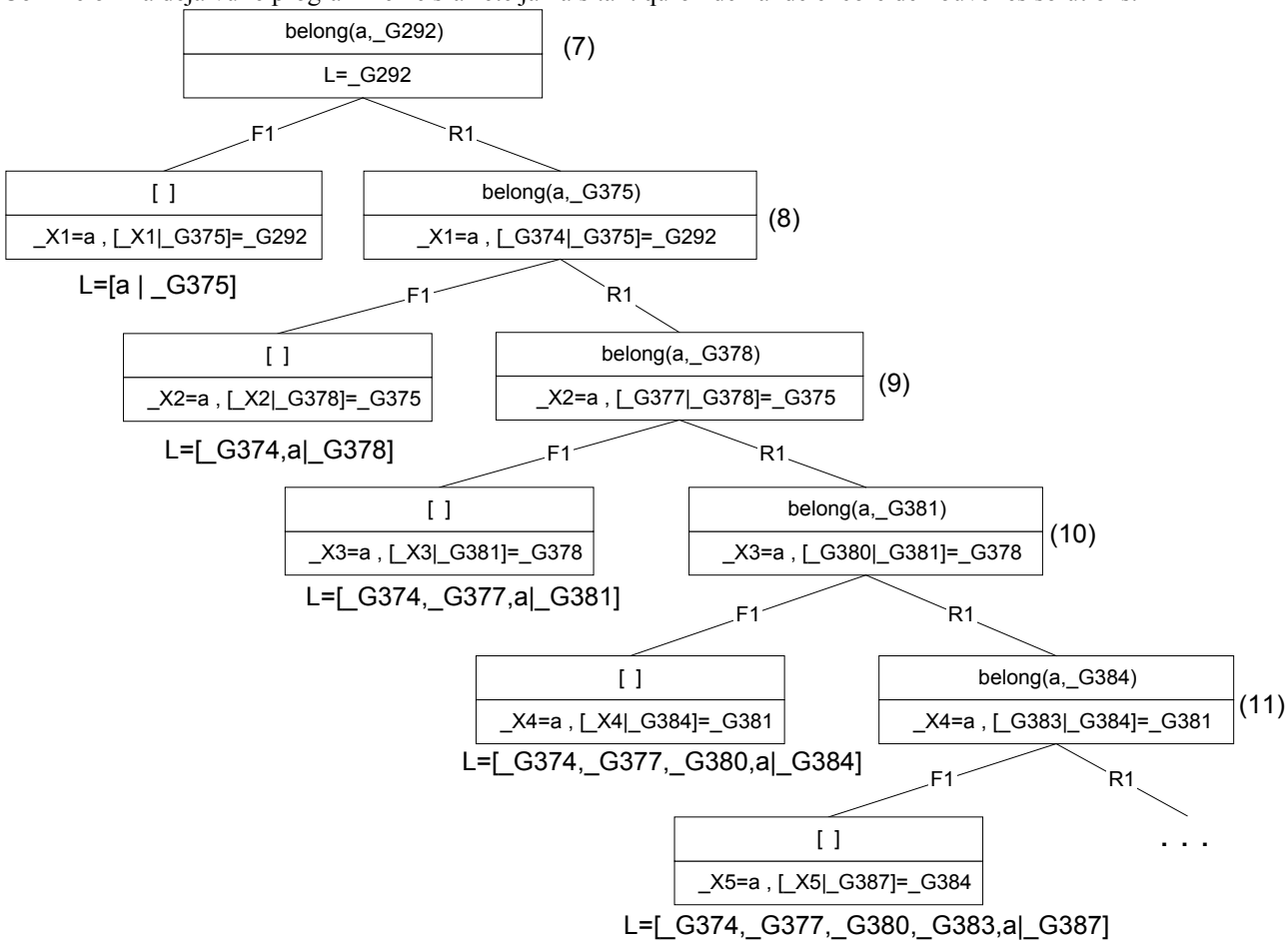
ou encore : unification impossible ⇒ échec à la démonstration

mais l'inverse n'est pas vrai.

**Ex** : Construction de toutes les listes contenant un ‘a’ :

```
[debug]  ?- belong(a,L).
T Call:  (  7) belong(a, _G292)
T Exit:  (  7) belong(a, [a|_G375])
L = [a|_G375] ;
T Redo:  (  7) belong(a, _G292)
T Call:  (  8) belong(a, _G375)
T Exit:  (  8) belong(a, [a|_G378])
T Exit:  (  7) belong(a, [_G374, a|_G378])
L = [_G374, a|_G378] ;
T Redo:  (  8) belong(a, _G375)
T Call:  (  9) belong(a, _G378)
T Exit:  (  9) belong(a, [a|_G381])
T Exit:  (  8) belong(a, [_G377, a|_G381])
T Exit:  (  7) belong(a, [_G374, _G377, a|_G381])
L = [_G374, _G377, a|_G381] ;
T Redo:  (  9) belong(a, _G378)
T Call:  ( 10) belong(a, _G381)
T Exit:  ( 10) belong(a, [a|_G384])
T Exit:  (  9) belong(a, [_G380, a|_G384])
T Exit:  (  8) belong(a, [_G377, _G380, a|_G384])
T Exit:  (  7) belong(a, [_G374, _G377, _G380, a|_G384])
L = [_G374, _G377, _G380, a|_G384] ;
T Redo:  ( 10) belong(a, _G381)
T Call:  ( 11) belong(a, _G384)
T Exit:  ( 11) belong(a, [a|_G387])
T Exit:  ( 10) belong(a, [_G383, a|_G387])
T Exit:  (  9) belong(a, [_G380, _G383, a|_G387])
T Exit:  (  8) belong(a, [_G377, _G380, _G383, a|_G387])
T Exit:  (  7) belong(a, [_G374, _G377, _G380, _G383, a|_G387])
L = [_G374, _G377, _G380, _G383, a|_G387] ;
...
```

Comme on l’a déjà vu le programme ne s’arrête jamais tant qu’on demande encore de nouvelles solutions.



Dans le précédent exemple, le programme boucle à l'infini tant que l'utilisateur lui demande des solutions parce qu'il présente une branche infinie à droite. Cependant, il est très facile de transformer cette récursivité droite en récursivité gauche et donc de créer une branche infinie gauche sur laquelle le programme va boucler sans même que l'utilisateur lui demande d'autres solutions puisque de toute façon, il ne donne aucune solution.

Pour cela, il suffit juste d'inverser l'ordre de lecture du fait F1 et de la règle R1.

**Programme 18 : Appartenance à une liste (2ème version)**

```
/* R1 */ belong(X,[T|L]):-belong(X,L).
/* F1 */ belong(X,[X|L]).
```

Vérifions d'abord que cette deuxième version du programme retourne toujours les mêmes résultats dans les cas classiques.

**Ex** : Etre ou ne pas être dans la liste…

```
?- belong(a,[q,d,a,f,d]).
Yes
?- belong(a,[q,d,f,d]).
No
```

**Ex** : Nombre d'occurrences d'un élément.

```
?- belong(X,[1,2,3,2,3,3]),X=1.
X = 1 ;
No
?- belong(X,[1,2,3,2,3,3]),X=3.
X = 3 ;
X = 3 ;
X = 3 ;
No
?- belong(X,[1,2,3,2,3,3]),X=0.
No
```

On obtient autant de réponses qu'il y a d'occurrences de l'élément recherché.

**Ex** : Enumération

```
?- belong(X,[e,n,u,m,e,r,a,t,i,o,n]).
X = n ;
X = o ;
X = i ;
X = t ;
X = a ;
X = r ;
X = e ;
X = m ;
X = u ;
X = n ;
X = e ;
No
```

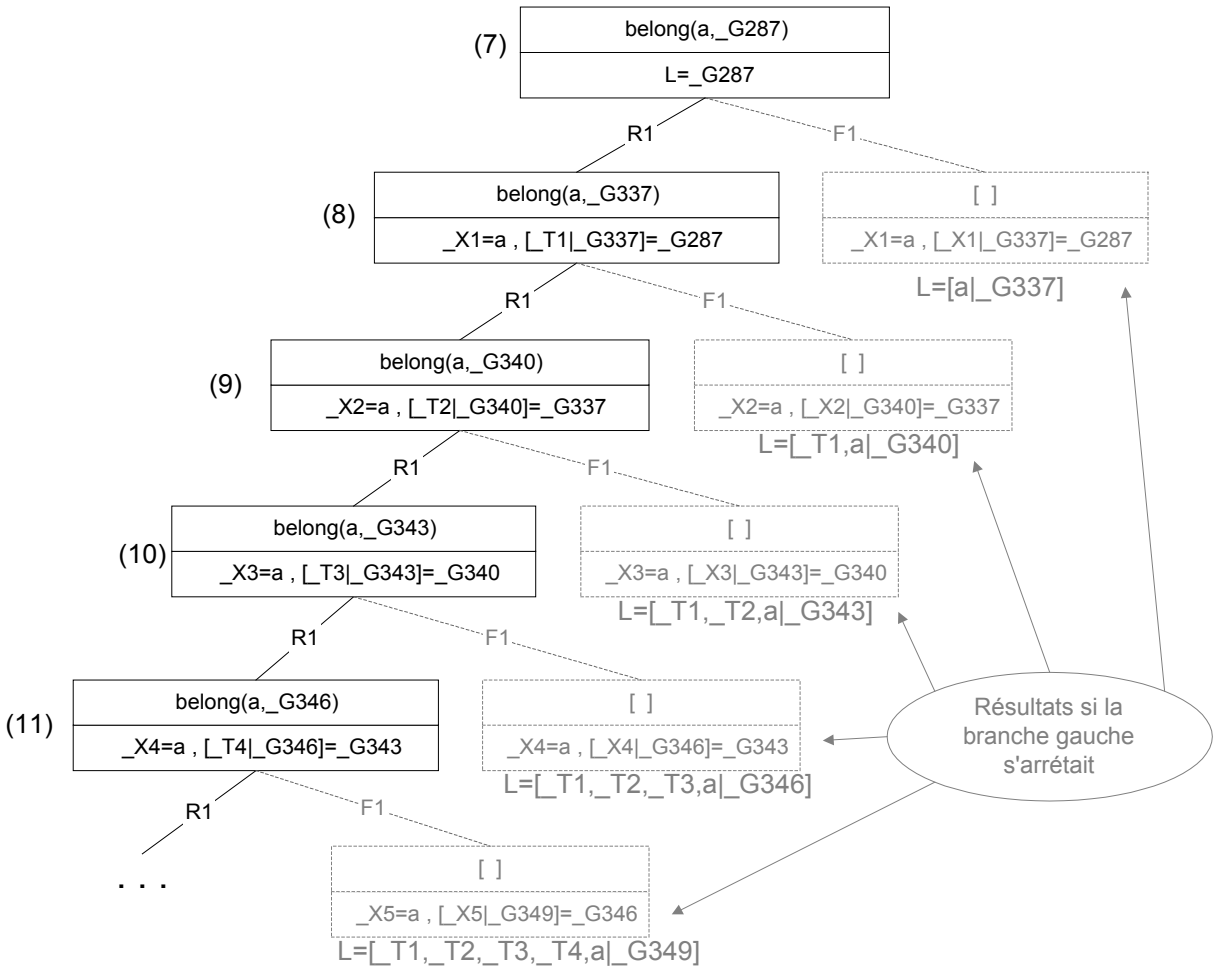
L'inversion de la règle R1 et du fait F1, modifie l'ordre dans lequel les éléments de la liste sont énumérés. Il semble bien (nous allons le vérifier dans l'exemple suivant) que nous ayons réussi à transformer la récursivité droite en récursivité gauche. En effet, si l'énumération commence à partir de la fin de la liste, c'est que PROLOG n'a pas trouvé de preuve avant d'atteindre la fin de la liste c'est à dire pas avant que la règle R1 ne soit plus applicable.

**Ex** : Construction de toutes les listes contenant un 'a' :

```
[debug]  ?- belong(a,L).
 T Call: (7) belong(a, _G287)
 T Call: (8) belong(a, _G337)
 T Call: (9) belong(a, _G340)
 T Call: (10) belong(a, _G343)
 T Call: (11) belong(a, _G346)
 T Call: (12) belong(a, _G349)
 T Call: (13) belong(a, _G352)
...
```

Nous obtenons comme voulu une boucle infinie.

En effet, on voit très bien, dans l'arbre ci-dessous, la récursivité gauche qui engendre une branche infinie gauche provocant la non-terminaison du programme car PROLOG continue tant qu'il ne rencontre pas de preuve (pile de niveau de but vide) et qu'il peut appliquer une règle (ici R1).




On peut voir que si la branche gauche s'arrêtait, tout comme pour l'énumération, les solutions seraient données à l'envers par rapport au premier programme puisqu'elles ne seraient pas données en 'descendant', mais en 'remontant' après avoir atteint le bout de la branche gauche.

Donc s'il n'est pas toujours possible d'éviter les récursivités qui bouclent, il vaut mieux que la récursivité soit à droite. De fait, une branche infinie à droite n'est pas trop gênante puisque l'utilisateur garde la main pour l'interrompre (il suffit qu'il ne demande plus d'autres solutions) et qu'il obtient des résultats (s'il y en a).

# IV. Conclusion

Tout d'abord insistons sur le fait que PROLOG est un langage de programmation à part entière, tout comme le C, Java, Caml, Scheme… Cependant, il se distingue de ces derniers par son mode de programmation. Il s'agit en effet d'un langage déclaratif, de haut niveau, interactif et particulièrement performant pour résoudre tous les problèmes ayant une relation avec la logique : systèmes experts, langage naturel, aide à la décision, représentation de connaissances...

Comme nous l'avons vu, PROLOG est étonnamment puissant puisqu'il suffit bien souvent de très peu de lignes de code pour mettre en place des programmes intéressants. Il est capable à partir de quelques faits et règles, décrits d'une façon très mathématique, d'établir une stratégie de réflexion pour répondre aux requêtes de l'utilisateur. Pourtant, il n'est pas non plus exempt de tout défaut. Ainsi, cette prétendue stratégie se ramène souvent à une simple énumération des cas possibles. De plus, il lui arrive assez fréquemment de se perdre dans des boucles infinies.