Programmation Récursive

Une fonction *récursive* est une fonction définie à partir d'elle même. En prolog, on appelle *formulation récursive d'une relation* quand un but fait appel au but lui-même (entre autres buts), ou quand on utilise la relation dans sa propre définition. Logiquement, les formulations récursives sont correctes, compréhensibles et intuitivement évidentes.

A. Arithmétique

Les entiers naturels sont constitués à partir de deux constructions, le symbole constant 0 et la fonction successeur `s` d'arité[1] `1`.
Tous les entiers naturels sont alors donnés récursivement par 0, s(0), s(s(0)), s(s(s(0))), ... On adoptera la convention que $s^n(0)$ dénote l'entier n, à savoir n applications de la fonction successeur à 0.
Observez la proposition suivante :

le programme sur les entiers naturels est complet et correct relativement à l'ensemble des buts `entier_naturel(`s^i`(0))`, avec `i ≥ 0`.

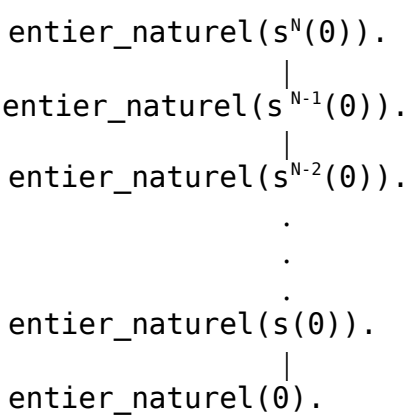
Rappel : *un programme logique est un ensemble d'axiomes ou de règles, définissant des relations entre objets. Un calcul d'un programme logique est une déduction de conséquences à partir du programme.*

On va montrer par un arbre de démonstration la *complétude* du programme : soit `N` un entier naturel. le but `entier_naturel(N)` est déductible du programme en donnant un arbre de démonstration explicite, ou bien N est 0 ou est de la forme $s^N(0)$.

Programme

```
entier_naturel(0).
entier_naturel(s(X)):- entier_naturel(X).
```

Démonstration



Les clauses se lisent ainsi :
La première '`0 est un entier naturel`'.
La deuxième clause se lit: `le successeur (s(X)) est un entier naturel si X est un entier naturel`.
La première clause est comme un test d'arrêt, nous pouvons dire que lorsque la première clause est

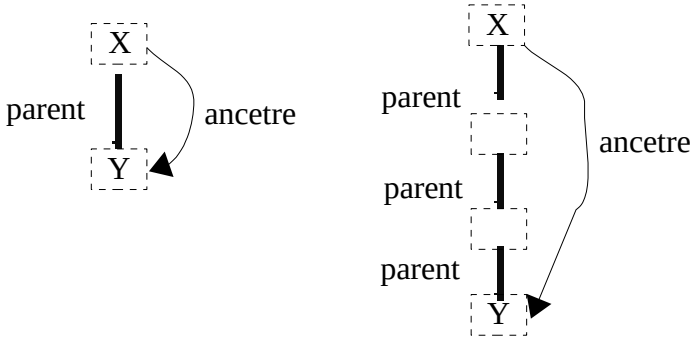
1 voir fichier glossaire.

vérifiée suite à la deuxième, alors l'interprète répond '`Yes`' à notre requête et il s'arrête.
En Prolog, les boucles existent sous cet aspect là. Dans la récursivité nous donnons le nombre de fois que nous voulons que la règle se vérifie et ensuite la première clause rencontrée et vérifiée alors on sort de la boucle et fournit le résultat.

Prenons l'exemple de la famille : nous définissons la relation parent/2 de la façon suivante :

```
parent(X, Y):- pere(X, Y);
               mere(X, Y).
```

qui signifie que X est parent de Y si et seulement si X est père de Y ou X est mère de Y.
A partir de cette relation nous définissons une nouvelle relation ancetre/2 qui sera vraie lorsque nous acceptons que tout parent X de Y est aussi ancetre X de Y. Regardez la figure qui suit :



La première règle est simple et l'on peut la formuler :
Pour tout X et Y,
X est un ancetre de Y si
X est parent de Y.

Donc en Prolog cela donne :

```
ancetre(X, Y) :-
     parent(X, Y).
```

La seconde règle est plus compliquée, car les parents intermédiaires peuvent poser quelques problèmes. Voyons comment peut-on définir la seconde règle :

```
ancetre(X, Y) :- parent(X, Y).

ancetre(X, Y) :- parent(X, Z), parent(Z, Y).

ancetre(X, Y) :- parent(X, Z1), parent(Z1, Z2), parent(Z2, Y).
```

Ceci donne un programme long et qui ne fonctionne que jusqu'à un certain point.
Nous allons revoir notre famille :
nous savons que `parent(jean, thomas)` est un fait. On peut conclure que `ancetre(jean, thomas)`. Ceci est un fait dérivé : il peut ne pas figurer explicitement dans notre programme, mais il peut être dérivé des autres règles et faits. Une telle inférence peut être écrite d'une manière plus compacte :

`parent(jean, thomas)` --> `ancetre(jean, thomas)`
On lit que `parent(jean, thomas)` implique `ancetre(jean, thomas)`. Dans cette famille on a `parent(thomas, sylvie)`. On peut alors déduire que `ancentre(jean, sylvie).`
Si l'on note le processus d'inférence en deux étapes on obtient :

`parent(jean, thomas)` → `ancetre(jean, thomas)`
`parent(thomas, sylvie)` et `ancetre(thomas, sylvie)` → `ancetre(jean, sylvie)`

Nous venons de montrer ce que peut être la suite d'étapes permettant de valider un but, que nous appellerons une *preuve*. Prolog obtient cette preuve dans l'ordre inverse de celui que l'on vient d'utiliser. Au lieu de partir des faits contenus dans le programme, l'interprète commence par les buts et un utilisant les règles, les remplace par de nouveaux buts, jusqu'à qu'il obtienne des faits élémentaires. Observez la question :

```
?- ancetre(jean, thomas).
```

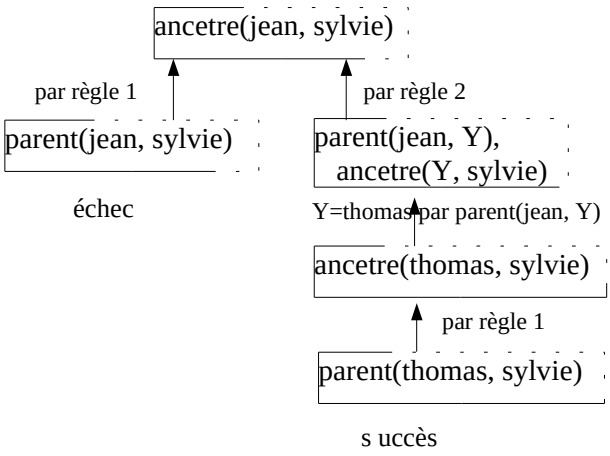
Prolog va tenter de valider ce but, et pour le faire, il va chercher une clause dans le programme dont ce but pourrait dériver. La règle : `ancetre(X, Y) :- parent(X, Y).` correspond au but `ancetre(jean, thomas)`, et les variables X et Y seront instanciées à X = jean et Y = thomas.

On peut donc écrire les règles suivantes pour ancetre/2 :

```
ancetre(X, Y) :-            % [2]voici la première règle
     parent(X, Y).

ancetre(X, Y) :-            % voici la deuxième règle, indirecte
     parent(X, Z),
     ancetre(Z, Y).
```

Voilà comment le fait ancetre(jean, sylvie) est réussi:



2 Les commentaires en Prolog peuvent s'écrire de deux façons : pour de courts commentaires on peut utiliser :
% ceci est un commentaire
ou bien pour les commentaires plus long :
/* voici un autre commentaire */

Retournons un peu aux entiers naturels que nous avons défini au début du chapitre.
Nous allons définir l'addition des entiers naturels comme on sait la faire : càd une opération fondamentale qui définit une relation entre deux entiers naturels et leur somme. Par complétude il faut entendre les règles qui permettent l'arrêt du programme.
Rappelons que pour les entiers naturels que nous avons défini le successeur de 0 est la suite naturelle de 0, et que le successeur de successeur de 0 est la suite naturelle de successeur de 0 et ainsi de suite : 0, s(0), s(s(0)), s(s(s(0))), s(s(s(s(0)))), etc.
Observez la question suivante :

```
?- addition(0, s(0), s(0)).                % consiste à vérifier que 0 + 1 = 1
Yes

?- addition(s(0), s(0), s(s(0))).          % consiste à vérifier que 1 + 1 = 2
Yes
```

On peut alors écrire les règles suivantes :

```
addition(0, X, X).      /*sachant que n'importe quel nombre additionné à 0 donne le nombre lui même */
addition(X, 0, X).

addition(s(X), Y, s(Z)) :- addition(X, Y, Z).
```

Nous avons la première règle qui décrit la relation : 0 + X = X
La deuxième qui décrit la même relation que la précédente mais la position de la variable change : X + 0 = X
Et la troisième règle qui décrit la relation : s(X) + Y = s(X + Y).

Un des avantages des programmes relationnels par rapport aux programmes fonctionnels consiste à des multiples usages que l'on peut faire du programme.

Le même programme est aussi facile à utiliser pour la soustraction observez les réponses aux questions qui suivent:

```
?- addition(X, s(0), s(s(s(s(0))))).

X= s(s(s(0)))
Yes


?- addition(X, s(0), s(s(0))).

X = s(0)
Yes


?- addition(X, s(0), s(0)).

X = 0
Yes
```

Une utilisation un peu particulière est celle où nous donnons seulement une donnée et deux variables, inconnues, regardez la question suivante :

```
?- addition(X, Y, s(s(0))).

X = 0
Y = s(s(0)) ;

X = s(s(0))
Y = 0 ;

X = s(0)
Y = s(0) ;

No
```

Cette utilisation plus originale que la précédente répond au fait qu'une question peut avoir des réponses multiples. Dans l'exemple qui précède la question est :

Y a-t-il des entiers naturel `X` et `Y` tels que leur somme soit égal à `s(s(0))` ?
Effectivement il y a plusieurs solutions, comme on peut le constater lors de la demande d'autres unifications possibles, pendant le back tracking.

Et dans notre programme on va définir la multiplication puisque nous avons déjà une règle définie celle de l'addition qui va être utilisée afin de définir la multiplication (additionner un nombre donné les entiers naturels entre eux).
Nous allons définir les règles de l'élément absorbant (le zéro) ainsi que l'élément neutre (le 1) séparément :

```
mult(0, X, 0):- entier_naturel(X).                    % élément absorbant
mult(X, 0, 0):- entier_naturel(X).
mult(s(0), X, X):- entier_naturel(X).                 % élément neutre
mult(X, s(0), X):- entier_naturel(X).
mult(s(X), Y, Z):- mult(X, Y, Z1), addition(Y, Z1, Z).
```

La première et la deuxième règle définissent l'élément absorbant de la multiplication qui est le zéro, la troisième et la quatrième règle définissent l'élément neutre de la multiplication qui est le 1 et la dernière règle définit la relation de la multiplication qui explicitement donne :
En multipliant un successeur de X avec un nombre Y j'obtiens Z si et seulement si je peux avoir Z1 en multipliant X avec Y et l'addition de Y avec le Z1 nous donne Z.

Exercices à faire :

1) Les entiers naturels sont munis d'un ordre naturel, essayez de définir la relation inférieur ou égal ainsi que la relation supérieur ou égal sachant que si l'on utilise une notation infixée comme on a l'habitude de le faire, cela donnerait les règles suivantes :

X <= Y
0 <= X :- entier_naturel(X).
s(X) <= s(Y) :- X <= Y.

Essayez de définir les mêmes règles avec une notation préfixée ≤(0, X).

2) Pourriez-vous expliquer les concepts introduits dans les trois premiers chapitres :

clause
fait, règle, question
tête d'une clause, corps d'une clause
récursivité
but
procédure
atome
variable
but satisfait, échec de but
retour en arrière (*backtracking*)
unification

****

A retenir :

En Prolog, un ensemble de clauses concernant une même relation s'appelle *procédure*.
On voit deux niveaux de signification d'un programme Prolog :

la signification *déclarative* et
la signification *procédurale*.

La signification déclarative concerne les relations définies par le programme. Elle détermine ce que le programme va produire, contrairement à la signification procédurale qui détermine **comment** les résultats seront obtenus.
D'un aspect pratique, les éléments déclaratifs d'un programme sont souvent plus compréhensibles que les détails procéduraux. Donc, comme les résultats dépendent de la partie déclarative, cela suffit pour écrire les programmes.
L'approche la plus efficace et la plus pratique sera déterminée par le problème.
Toutefois, on verra par la suite que pour les grands programmes l'aspect procédural détient une place importante pour le programmeur surtout pour des raisons d'efficacité lors de l'exécution, car les solutions déclaratives sont généralement les plus simples à mettre au point, mais souvent elles engendrent des programmes inefficaces.
Par exemple, lors de l'établissement d'une solution, il est pratique parfois de réduire le problèmes à un ou plusieurs sous-problèmes plus simples, il y a plusieurs méthodes qui permettent de trouver des sous-problèmes adéquats en Prolog, dont une la *méthode de la récursion*.
Nous verrons plusieurs exemples de programmes qui utilisent la récursion, au chapitre suivant qui traite les **listes**.