4.3. Récursivité enveloppée versus terminale *(VIDEO-PPT 3.2)*

a) Récursivité enveloppée

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

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

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

D'où le programme en *Prolog* :

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

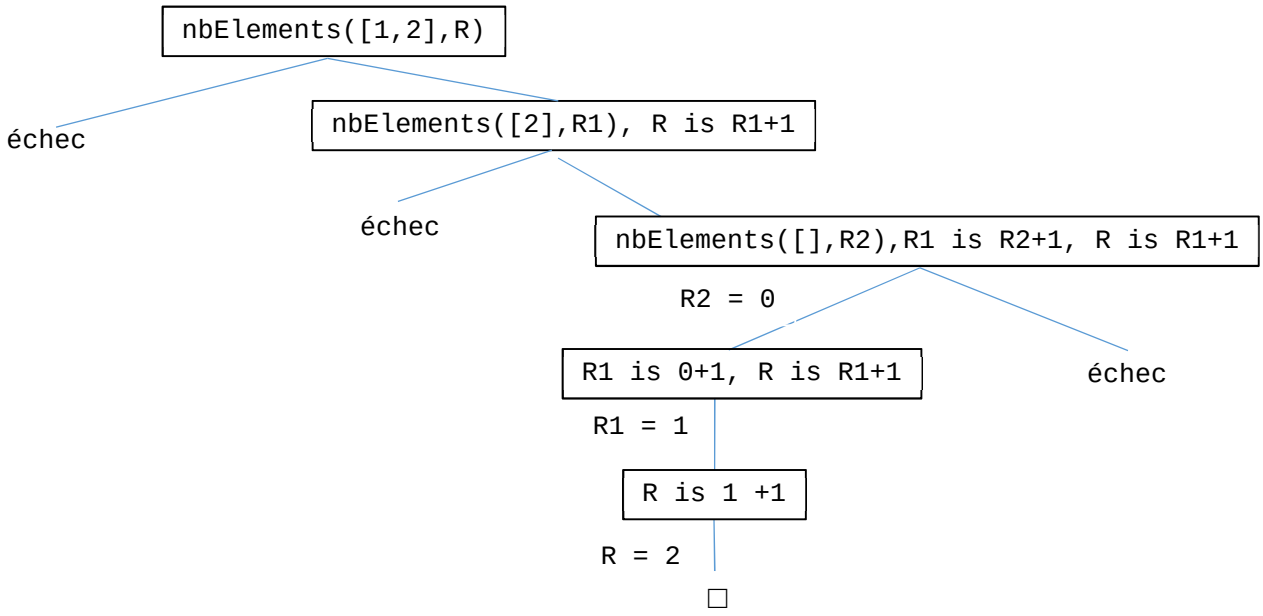
En *Prolog* le pseudo-prédicat `X is` *expr* unifie `X` à l'évaluation de l'expression *expr*.

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

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

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

L'arbre de recherche de la question `nbElement([1,2], R)` est :



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

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

b) Récursivité terminale

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

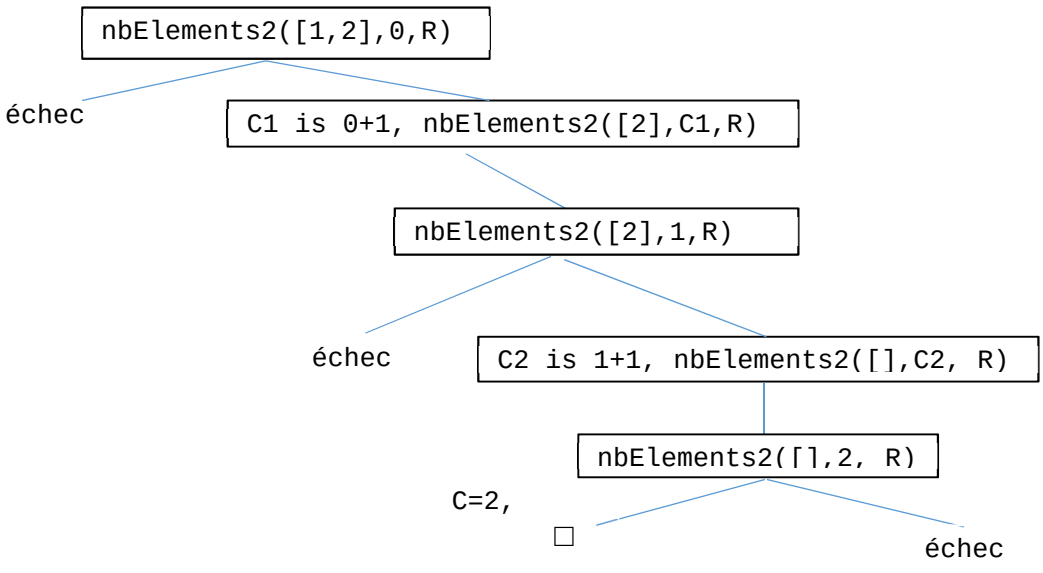
Un invariant de boucle donne le sens de `C` et `P`. À une étape du processus de comptage `P` désigne la liste des éléments restant à compter, `C` désigne le nombre d'éléments qui ont déjà été comptés. Au démarrage du calcul `P` désigne toute la liste et `C=0`, car rien n'a été compté. Bien entendu, quand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

D'où :

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

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

L'arbre de recherche de cette question est :



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

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

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

Exemple hors-Prolog (qui ne permet pas l'itération) :
Pour une fonction récursive terminale dont :

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

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

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

qui se réécrit en itératif :

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

➢ *Mini-TP : factorielle*

1. *En utilisant le schéma de récursivité terminale suivant :*

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

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

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

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