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.

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 :

$$\begin{array}{lll} \text{fact} : & N & N \\ & 0 & 1 \\ & n & n \times (n-1)! \end{array}$$

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

Programme 1 : 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 10307.

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 par la suite 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.

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 2 : 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.

3. Valeur approchée

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

3.1. Première approximation

Ainsi, on peut utiliser la formule de James Sterling : $! \approx \sqrt{2} \; -$

Programme 3 : 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.

3.2 Calcul logarithmique

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

$$\log(!) = \log\left(\sqrt{2\pi} \times \frac{/}{\quad}\right) = \frac{1}{2}\log(\sqrt{2\pi}) + !+ \left(\frac{1}{2}\right)'' \times \log(-) \times \log$$

Programme 4 : 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… ($3000! = 4{,}149\ldots\times 10^{9130}$)

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 : $! \approx \sqrt{2\pi n} + \frac{}{} + \frac{}{} - \frac{}{} \times$ –

Programme 5 : 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, 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
```