Programmation avec Maple

par Sadik BOUJAIDA

Instructions conditionnelles

Syntaxe :

if **cond1** *then* **instr1**
elif **cond2** *then* **instr2**
.
.
elif **condN** *then* **instrN**
else **instr(N+1)**
fi

condK sont des expressions maple quelconques dont l'évaluation doit donner une valeur logique, *true* ou *false*. **instrK** sont chacune un suite d'instructions Maple séparées par des symboles de terminaison ";" ou ":". Noter qu'il est inutile de placer des terminaisons avant les mot clés *elif*, *else* et *fi*. Dés qu'une condition **condK** se réalise l'instruction **instrK** est exécutée et on sort du bloc conditionnel **if ...fi**. Dans le cas où aucune des conditions **cond1,...,condN** ne se réalise c'est l'instruction **instr(N+1)** qui sera exécutée.
Une instruction conditionnelle doit comporter au moins le bloc *if* **cond1** *then* **instr1**, le reste n'est pas obligatoire.

Pour réaliser des tests, Maple offre :

- les opérateurs de comparaison =,<> (pour différent), <,<=,>,>=. qui permettent de comparer deux variables de type numérique.
- L'opérateur de comparaison =, qui teste l'égalité de deux objets Maple, qui peuvent être des nombres, des expressions algebriques, des séquences, des listes, des ensembles, des chaines de charactères... mais pas des tableaux, des table, ou des procédures.
- le mot clé *in* qui permet de tester si un élément **elt** est dans un objet **obj** selon la syntaxe **elt** *in* **obj**. **obj** doit être une liste ou un ensemble.
- La procédure *type* qui permet de controler le type de donnée d'une variable et retourne true ou false selon est ce que le typage est correct ou pas.

```
> if 3 in [k$k=1..10] then  "houra" fi;
                                                "houra"
> if 0 in [k$k=1..10] then  "houra" else "raté" fi;
```

"raté"

Une instruction de comparaison n'est pas évaluée en tant qu'expression logique en dehors du contexte d'une instruction conditionnelle (if mais aussi while). Si besoin on peut utiliser la procédure evalb pour forcer l'évaluation d'une telle instruction en dehors du dit contexte.

```
> 2>1;
```

$$1 < 2$$

```
> evalb(2>1);
```

$$true$$

```
> 1 in [2,4,6];
```

$$1 \in [2, 4, 6]$$

```
> evalb(1 in [2,4,6]);
```

$$false$$

```
> L:= {1,2,3};K:={3,1,2,1,3};
```

$$L := \{1, 2, 3\}$$
$$K := \{1, 2, 3\}$$

```
> evalb(L=K);
```

$$true$$

```
> A:=array(1..2,[1,1]);B:=array(1..2,[1,1]);
```

$$A := \begin{bmatrix} 1 & 1 \end{bmatrix}$$
$$B := \begin{bmatrix} 1 & 1 \end{bmatrix}$$

```
> evalb(eval(A)=eval(B));
```

$$false$$

```
> chaine1:="pour tester"; chaine2:="pour tester";
```

$$chaine1 := \text{"pour tester"}$$
$$chaine2 := \text{"pour tester"}$$

```
> evalb(chaine1=chaine2);
```

$$true$$

Les boucles :

Syntaxe d'une boucle for :

for **k** *from* **init** *to* **fin** *by* **pas** *do* **instr** *od*;

ou bien :

for **k** *in* **obj** *do* **instr** *od*;

Syntaxe d'une boucle while :

while **cond** *do* **instr** *od*;

Pour la boucle for : le compteur k prend son départ avec la valeur init et est incrémenté à chaque fois par la valeur pas, tant qu'on n'a pas atteint la valeur fin, a chaque valeur du compteur le suite d'instructions inst est executée. les valeurs init, pas et fin peuvent ne pas être des entiers mais n'importe quelle valeurs numériques. les clauses from, by et to sont optionnelles. La valeur par défaut du pas est 1, la valeur par défaut du départ est 1. Et pour le bloc to, il peut être omis à condition de prévoir une instruction qui permet d'arreter l'éxecution de la boucle: le mot clé break associé à une instruction if, qui permet

de sortir de la boucle si une ceratine condition se réalise.
La deuxième syntaxe est une variante de la première, obj est un objet Maple quelconque et k va prendre successivement la valeurs des opérandes de obj. Dans la pratique obj sera une liste, une chaine de charactère ...
Pour la boucle while : a les instructions instr seront exécutées tant que l'évaluation de l'expression logique cond retournera true comme valeur.
On peut déjà constater qu'une boucle for est en fait une boucle while, le test se basant sur une comparaison du compteur k avec la valeur d'arrêt fin.

```
> L:=[];for k in "Hello world" do L:=[op(L),k] od;
```

$$L := [\ ]$$
$$L := [\text{"H"}]$$
$$L := [\text{"H"}, \text{"e"}]$$
$$L := [\text{"H"}, \text{"e"}, \text{"l"}]$$
$$L := [\text{"H"}, \text{"e"}, \text{"l"}, \text{"l"}]$$
$$L := [\text{"H"}, \text{"e"}, \text{"l"}, \text{"l"}, \text{"o"}]$$
$$L := [\text{"H"}, \text{"e"}, \text{"l"}, \text{"l"}, \text{"o"}, \text{" "}]$$
$$L := [\text{"H"}, \text{"e"}, \text{"l"}, \text{"l"}, \text{"o"}, \text{" "}, \text{"w"}]$$
$$L := [\text{"H"}, \text{"e"}, \text{"l"}, \text{"l"}, \text{"o"}, \text{" "}, \text{"w"}, \text{"o"}]$$
$$L := [\text{"H"}, \text{"e"}, \text{"l"}, \text{"l"}, \text{"o"}, \text{" "}, \text{"w"}, \text{"o"}, \text{"r"}]$$
$$L := [\text{"H"}, \text{"e"}, \text{"l"}, \text{"l"}, \text{"o"}, \text{" "}, \text{"w"}, \text{"o"}, \text{"r"}, \text{"l"}]$$
$$L := [\text{"H"}, \text{"e"}, \text{"l"}, \text{"l"}, \text{"o"}, \text{" "}, \text{"w"}, \text{"o"}, \text{"r"}, \text{"l"}, \text{"d"}]$$

```
> printlevel:=2: # Voir explication plus en avant dans ce cours.
> for k in [1,2,0,7,-5,3,4,1] do if k>=0 then cat("la valeur de k est",k) else
   break fi od;
```

$$k := 1$$
"la valeur de k est1"
$$k := 2$$
"la valeur de k est2"
$$k := 0$$
"la valeur de k est0"
$$k := 7$$
"la valeur de k est7"
$$k := -5$$

La boucle s'arrete dés que la positivité de k tombe en défaut, grace à l'instruction break.
La même chose peut être faite en mixant dans une même boucle les mots clés for et while, noter qu'il ne s'agit pas du tout de deux boucles imbriquées, mais d'une seule boucle.

```
> printlevel:=1:
> for k in [1,2,0,7,-5,3,4,1] while k>=0 do cat("la valeur de k est",k) od;
```

"la valeur de k est1"
"la valeur de k est2"
"la valeur de k est0"
"la valeur de k est7"

Les procédures :

Généralités

Les procédures Maple sont des blocs de codes, comportant plusieurs instructions, auquel on peut donner un nom (par affectation). On exécute le bloc en faisant appel au nom

qu'on lui affecté.
Une procédure peut avoir 0 ou plusieurs argument, et peut retourner ou pas une valeur. La declaration des types des arguments, quoique possible n'est pas obligatoire dans Maple, non plus le type de la valeur qu'elle retourne (si elle retourne une valeur).
Les procédures sont à proprement parler des programmes Maple. Dans Maple on ne fait pas de différence entre procédures et fonctions (comme le fait PASCAL), et le passage des arguments se fait par valeurs et non par références (commme le fait le langage C): le nom d'un argument ne peut pas recevoir une affectation dans le corps de la procédure.
Il y'a un mécanisme de variables locales et globales qui définit la portée d'une variable qui a été définie dans le corps d'une procédure. La récursivité est gérée de façon naturelle et sans aucun effort particulier de la part du programmeur.

Pour des procédures simple, on dispose de l'opérateur ->, qui permet de définir une procédure à la manière d'une fonction mathématique. Les procédures plus complexes se verront plutôt construire grace au constructeur *proc*.
La séquence des arguments d'une procédure est stockée localement dans la variable de nom **args,** et le nombre de ses arguments dans la variable nargs.
La valeur retournée par une procédure est le résultat de la dernière instruction exécutée. En cas de besoin de retourner une valeur donnée sans exécuter le suite des instructions dans une procédure on utilise la fonction RETURN, qui retourne son argument et provoque la sortie de la procédure.

```
> f:=x->x^2;
```

$$f := x \rightarrow x^2$$

```
> f(2);f(a);
```

$$4$$
$$a^2$$

```
> g:=()->args; #g retourne simplement la séquence des arguments qu'on lui
  fournit
```

$$g := () \rightarrow args$$

```
> g(1,2,a,x);
```

$$1, 2, a, x$$

Une définition récursive de la fonction factorielle :

```
> fact:=proc(n) ;
  if n=0 then RETURN(1) else n*fact(n-1) fi;
  end proc;
```

$$fact := \mathbf{proc}(n)\ \mathbf{if}\ n = 0\ \mathbf{then}\ RETURN(1)\ \mathbf{else}\ n * fact(n-1)\ \mathbf{end\ if}\ \mathbf{end\ proc}$$

```
> fact(7);
```

$$5040$$

Cette définition est loin d'être parfaite. Rien n'empèche de passer un entier négatif en argument à la procédure fact (ou encore pire un nom de variable sans valeur), auquel cas l'appel récursif de fact bouclera à l'infini. D'où l'importance du typage des arguments d'une procédure.

```
> fact:=proc(n::nonnegint) ;
  if n=0 then RETURN(1) else n*fact(n-1) fi;
```

```
  end proc;
```

$$fact := \mathbf{proc}(n::nonnegint)\ \mathbf{if}\ n = 0\ \mathbf{then}\ RETURN(1)\ \mathbf{else}\ n * fact(n-1)\ \mathbf{end\ if}\ \mathbf{end\ proc}$$

```
> fact(-7); # provoque une erreur.
Error, invalid input: fact expects its 1st argument, n, to be of type nonnegint, but
received -7
```

Par la même occasion, on peut en fait déclarer aussi le type de la valeur retournée par une procédure avec la synstaxe :

proc(seq_arg)::type_sortie;

Définition récursive de la procédure pgcd qui retourne le PGCD de deux entiers

```
> pgcd:=proc(a::integer,b::integer)::inetger;
  if a=0 then RETURN(b) else pgcd(b mod a, a) fi
  end proc;
```

$$pgcd := \mathbf{proc}(a::integer, b::integer)::inetger,$$
$$\quad \mathbf{if}\ a = 0\ \mathbf{then}\ RETURN(b)\ \mathbf{else}\ pgcd(mod(b, a), a)\ \mathbf{end\ if}$$
$$\mathbf{end\ proc}$$

```
> pgcd(-15,12);
```

$$3$$

```
> pgcd(7,x);
Error, invalid input: pgcd expects its 2nd argument, b, to be of type integer, but
received x
```

Autre déffaillance (de l'avis de l'auteur), c'est que une procédure n'a aucun contrôle sur le nombre d'arguments qu'on lui passe. Une procédure qui a par exemple été définie avec deux arguments, s'executera lors d'un appel avec les deux premiers éléments de la séquence des arguments qu'on lui passe en ignorant les autres. Bien sur Maple provoquera une erreur si on passe à la même procédure un seul argument.

```
> pgcd(12,15,7);
```

$$3$$

```
> pgcd(4);
Error, invalid input: pgcd uses a 2nd argument, b (of type integer), which is missing
```

On peut quand même contoler explicitement le nombre d'arguments. C'est l'occasion d'introduire la fonction de gestion des erreurs ERROR. Cette fonction permet sur la base d'un test de provoquer une exception, et l'arrêt immediat de l'éxécution d'une procédure si une certaine condition n'est pas remplie.

```
> pgcd:=proc(a::integer,b::integer)::integer;
  if nargs > 2
       then ERROR("pgcd prend exactement  deux arguments")
  elif a=0
       then RETURN(b)
  else
       pgcd(b mod a, a)
  fi
  end proc;
```

$$pgcd := \mathbf{proc}(a::integer, b::integer)::integer,$$
$$\quad \mathbf{if}\ 2 < nargs\ \mathbf{then}$$
$$\qquad ERROR(\text{"pgcd prend exactement deux arguments"})$$

```
    elif a = 0 then
        RETURN(b)
    else
        pgcd(mod(b, a), a)
    end if
end proc
```

N.B : Un manque trés génant dans l'interface de Maple est l'absence totale de fonctionnalité d'indentation su code, pour une meilleur lisibité. Le débutant qui commençera l'apprentissage de la programmation dans Maple, héritera d'une très mauvaise habitude.

```
> pgcd(12,15,17);
Error, (in pgcd) pgcd prend exactement  deux arguments
> :=
```

Variables locales, variables globales

Une variable qui n'est pas un argument, et qui a été introduite dans le corps d'une procédure peut être déclarée comme local ou global. En cas d'absence de déclaration, par défaut toute variable est locale. Si une variable est déclarée comme locale, elle n'a d'existence que le temps d'execution de la procédure lors d'un appel, aprés quoi le nom est libéré. Une variable déclarée comme globale continue d'exister aprés l'éxecution de la procédure. Un programme qui contient plusieurs procédure peut profiter des variables globales, puisque elles pouront être utilisées et changées dans les différentes procédures.

```
> test_local:=proc()
  local a; global b;
  a:=1;b:=1;
  end proc;
```

test_local := **proc**( ) **local** *a*; **global** *b*; *a* := 1; *b* := 1 **end proc**

```
> test_local();
```

1

```
> a;
```

a

```
> b;
```

1

Options Maple pour les procédures

Une procédure peut être définie avec certaines option prédefinie dans Maple. La déclaration d'une option doit être faite immédiatement après la declaration des variables locales et globales (si présentes) en utilisant le mot clé *option*.

Les options les plus interessantes disponibles sont *remember* et *trace*.

L'option remember permet d'associer à une procédure une table dite table remember, ou seront stockées toutes les valeurs calculées aprés chaque appel de la procédure. On peut accéder à cette table comme la quatrième opérande de la procédure.

Cette option et surtout utile avec les procédures récursives, puisque elle évite de chaque fois déscendre jusqu'à la valeur de sortie précisée dans le corps de la procédure. En contre partie de l'espace mémoire sera occupé par la table.
Pour tester cette fonctionalité on va définir deux procédures récursives, l'une va utiliser l'option remember l'autre non. On va utiliser la commande time() qui affiche le temps avec un formatage en float, pour chronometrer l'éxécution des deux procédures.

```
> restart:
> fact:=proc(n::nonnegint)
  if n=0 then return 1 else n*fact(n-1) fi;
  end proc;
```

$$fact := \mathbf{proc}(n::nonnegint)\ \mathbf{if}\ n = 0\ \mathbf{then}\ \mathbf{return}\ 1\ \mathbf{else}\ n * fact(n-1)\ \mathbf{end\ if}\ \mathbf{end\ proc}$$

```
> fact_rem:=proc(n::nonnegint)
  option remember;
  if n=0 then return 1 else n*fact(n-1) fi;
  end proc;
```

$$fact\_rem := \mathbf{proc}(n::nonnegint)$$
$$\quad \mathbf{option}\ remember;$$
$$\quad \mathbf{if}\ n = 0\ \mathbf{then}\ \mathbf{return}\ 1\ \mathbf{else}\ n * fact(n-1)\ \mathbf{end\ if}$$
$$\mathbf{end\ proc}$$

```
> op(4,eval(fact_rem)); # table remember encore vide
```

On remplit un peu la table de fact_rem

```
> for k in [5,15,25,50,75] do fact_rem(k) od;
```

$$120$$
$$1307674368000$$
$$1551121004333098598400000 0$$
$$3041409320171337804361260816606476884437764156896051200000000000 0$$
$$2480914081139539809194647711659403366092624388657012283779589451265584267757\2$$
$$8674094438154240000000000000000000$$

```
> op(4,eval(fact_rem));
```

$$table([5 = 120, 15 = 1307674368000, 75$$
$$= 2480914081139539809194647711659403366092624388657012283779589451265584267757\2$$
$$8674094438154240000000000000000000, 25 = 1551121004333098598400000 0, 50$$
$$= 3041409320171337804361260816606476884437764156896051200000000000 0])$$

On chronogmètre maintenant

```
> t:=time():fact(1000):time()-t;s:=time():fact_rem(1000):time()-s;
```

$$0.008$$
$$0.004$$

L'option trace : cette option permet de faire entrer une procédure dans un état de débogage. La définition d'une procédure, peut être correcte au niveau de la syntaxe. Mais cela n'empeche pas que des erreurs peuvent survenir lor de l'appel de la procédure. Erreur qu'il est parfois difficile de corriger puisque on sait pas au niveau de quelle instruction elles surviennent. Avec l'option trace, Maple affichera tous les détails des calculs effectuée lors d'un appel de la procédure.

```
> pgcd:=proc(a::integer,b::integer)
  option trace;
```

```
  if a=0 then b else pgcd(b mod a,a) fi;
  end proc;
```

$$pgcd := \mathbf{proc}(a::integer, b::integer)\ \mathbf{option}\ trace;\ \mathbf{if}\ a = 0\ \mathbf{then}\ b\ \mathbf{else}\ pgcd(mod(b,a),a)\ \mathbf{end\ if}\ \mathbf{end\ proc}$$

```
> pgcd(30,66);
{--> enter pgcd, args = 30, 66
{--> enter pgcd, args = 6, 30
{--> enter pgcd, args = 0, 6
```

$$6$$

```
<-- exit pgcd (now in pgcd) = 6}
```

$$6$$

```
<-- exit pgcd (now in pgcd) = 6}
```

$$6$$

```
<-- exit pgcd (now at top level) = 6}
```

$$6$$

Il est à noter qu'une procédure qui a été définie grace à l'opérateur -> est en fait une procédure avec l'option arrow. Entre autre options il y'a l'option builtin propre à des procédures Maple qui sont fait écrite en langage machine (le langage C pour la plupart).

Evaluation et appels d'une procédure

Les procédures comme pour les tableaux et les tables sont évaluées jusqu'à leurs derniers noms. Donner un deuxième nom à une procédure qui a été déjà définie fera que les deux noms pointeront en fait vers le même objet.
On peut effectuer certains operations sur les procédures. Les opérations permises ne seront limitées que par le type des valeurs retournées par les procédures (comme en mathématiques).

```
> restart;f:=x->x^2;g:=x->x^(1/2);h:=f+g;
```

$$f := x \to x^2$$

$$g := x \to \sqrt{x}$$

$$h := f + g$$

```
> h(x);
```

$$x^2 + \sqrt{x}$$

Une opération notable est la composition qui se fait avec l'opérateur @.

```
> l:=g@f;
```

$$l := g@f$$

```
> l(x);l(2);l(-1);
```

$$\sqrt{x^2}$$

$$\sqrt{4}$$

$$1$$

Un nom f suivi d'une parenthèse (est de facto considére par Maple comme le nom d'une procédure, même si cette dernière n'a pas été définie ou lorsque ce "nom" est une valeur constante, dans ce dernier cas la procédure est la fonction constante qui retourne cette valeur.

```
> restart;
```

```
> f(5);
```

$$f(5)$$

```
> 1(5);
```

$$1$$

Une expression de la forme **f(x)** qui ne peut être évaluée en autre chose (à cause de x ou à cause de f) admet un type Maple spécial : function

```
> whattype(f(x));whattype(f(1));whattype(1(x));whattype(sin(1));
```

$$\mathit{function}$$
$$\mathit{function}$$
$$\mathit{integer}$$
$$\mathit{function}$$

Maintenant du à leurs mode d'évaluation on ne peut afficher le corps d'une procédure en faisant simplement appel à son nom, on doit passer par la fonction d'évaluation *eval* qui retourne la valeur de la procédure ou print qui se contente d'afficher le corps de la procédur

```
> func:=proc() print("j'affiche mes arguments"); args end proc;
```

$$\mathit{func} := \mathbf{proc}(\,)\ \mathit{print}(\text{"j'affiche mes arguments"});\ \mathit{args}\ \mathbf{end\ proc}$$

```
> func;
```

$$\mathit{func}$$

```
> eval(func);
```

$$\mathbf{proc}(\,)\ \mathit{print}(\text{"j'affiche mes arguments"});\ \mathit{args}\ \mathbf{end\ proc}$$

```
> print(func);
```

$$\mathbf{proc}(\,)\ \mathit{print}(\text{"j'affiche mes arguments"});\ \mathit{args}\ \mathbf{end\ proc}$$

Attention print ne retourne aucune valeur et une variable auquelle on aura affecté le résultat d'une commande *print* sera *NULL*.

```
> gunc:=eval(func);hunc:=print(func);
```

$$\mathit{gunc} := \mathbf{proc}(\,)\ \mathit{print}(\text{"j'affiche mes arguments"});\ \mathit{args}\ \mathbf{end\ proc}$$
$$\mathbf{proc}(\,)\ \mathit{print}(\text{"j'affiche mes arguments"});\ \mathit{args}\ \mathbf{end\ proc}$$
$$\mathit{hunc} :=$$

```
> gunc(a,b);
```

$$\text{"j'affiche mes arguments"}$$
$$a, b$$

```
> hunc(a,b);
```

Niveaux d'exécution et verbosité de l'interface

La variable d'environnement *printlevel*

Chaque instruction à un niveau d'exécution qui décide du détail des informations affichées lors de son exécution. Le niveau d'exécution par défaut de Maple est 1. Les boucles et les instructions conditionnelles ont un niveau égal à 1 et les procédures 5.
Ce qui fait que lors de l'exécution de boucles imbriquée seuls les résultats de la boucles extérieure sont affichés. De même lors de l'appel d'une procédure seule la valeur finale est retourné sans aucun détail sur son exécution.
On régle le niveau d'exécution de Maple grace à la variable d'environnement *printlevel*.

```
> printlevel; # La valeur par défaut.
```

$$1$$

```
> for k to 9 do if (k mod 2)=0 then "paire" else "impaire" fi od ; # Nada,
  silence complet.
> printlevel:=2;
```

$$printlevel := 2$$

```
> for k to 9 do if (k mod 2)=0 then "paire" else "impaire" fi od ;
```

$$\begin{gathered}\text{"impaire"}\\ \text{"paire"}\\ \text{"impaire"}\\ \text{"paire"}\\ \text{"impaire"}\\ \text{"paire"}\\ \text{"impaire"}\\ \text{"paire"}\\ \text{"impaire"}\end{gathered}$$

```
> f:=proc(x) local y; y:=x; to 5 do y:=x*y od end proc;
```

$$f := \mathbf{proc}(x)\ \mathbf{local}\ y;\ y := x;\ \mathbf{to}\ 5\ \mathbf{do}\ y := x*y\ \mathbf{end\ do}\ \mathbf{end\ proc}$$

```
> f(a); # resultat uniquement
```

$$a^6$$

```
> printlevel:=5;
```

$$printlevel := 5$$

```
> f(a); # juste de quoi attiser notre curiosité, la boucle n'est pas touchée.
{--> enter f, args = a
```

$$y := a$$

```
<-- exit f (now at top level) = a^6}
```

$$a^6$$

```
> printlevel:=6; # et la boucle aussi
```

$$printlevel := 6$$

```
> f(a);
{--> enter f, args = a
```

$$\begin{gathered}y := a\\ y := a^2\\ y := a^3\\ y := a^4\\ y := a^5\\ y := a^6\end{gathered}$$

```
<-- exit f (now at top level) = a^6}
```

$$a^6$$

```
> printlevel:=1000; # Si vous êtes voyeur.
```

$$printlevel := 1000$$

```
> printlevel:=1; # retour à la normale;
```

$$printlevel := 1$$

```
> restart;
```

Interface et verboseproc

```
> f:=proc(x) local y; y:=x; to 5 do y:=x*y od end proc;
```

$$f := \mathbf{proc}(x)\ \mathbf{local}\ y;\ y := x;\ \mathbf{to}\ 5\ \mathbf{do}\ y := x*y\ \mathbf{end\ do}\ \mathbf{end\ proc}$$

```
> print(f); # pas de problème ici, on peut tout se dire.
```

$$\mathbf{proc}(x)\ \mathbf{local}\ y;\ y := x;\ \mathbf{to}\ 5\ \mathbf{do}\ y := x*y\ \mathbf{end\ do}\ \mathbf{end\ proc}$$

```
> print(igcd); # Attention secret d'état
```

$$\mathbf{proc}(\,)\ \mathbf{option}\ builtin = igcd;\ \mathbf{end\ proc}$$

De toute façon avec l'option builtin on ne peut avoir mieux, la procédure n'est pas écrite en langage Maple. Par contre

```
> print(ifactor);
```

C'est là que Maple nous offre une fonction specialisée dans le rélage des paramètres de l'interface, l'un de ces parametres et le degré de verbosité. Cette fonction est *interface*, et le paramètre qui nous ineteresse ici est *verboseproc*. Il est reglé par défaut sur la valeur 1. Pour avoir plus de détail sur les procédures Maple (qui sont souvent écrite en langage Maple, une lecture très instructive, la meilleure des écoles), il faut le mettre sur 2.

```
> interface(verboseproc=2);
```

$$1$$

```
> print(ifactor);
```

```
proc(n)
    option remember, system,
    Copyright (c) 1991 by the University of Waterloo. All rights reserved.;
    local sol, r, t1;
    global ifactor/bottom;
    if nargs < 1 then
        error "argument required"
    elif 1 < nargs and not type(args[2], 'name') then
        error "second argument must be a name"
    end if;
    if type(n, 'integer') then
        if 0 < n then sol := 1; r := n elif n < 0 then sol := -1; r := -n else return 0 end if
    elif type(n, 'fraction') then
        return procname(op(1, n), args[2 .. -1]) / procname(op(2, n), args[2 .. -1])
    elif type(n, {'`*`', 'set', 'list', 'relation'}) then
        return map(procname, n, args[2 .. -1])
    elif type(n, `^`) and type(op(2, n), 'integer') then
        return procname(op(1, n), args[2 .. -1])^op(2, n)
    elif type(n, ''('integer')) then
        return procname(op(1, n), args[2 .. -1])
    else
        error "invalid arguments"
    end if;
    if assigned(ifactor/from_signature[r]) then return ifactor/from_signature[r] end if;
    t1 := [ifactor/SmallFactors(r, 9)];
    sol := sol*t1[1];
    while 2 < nops(t1) do t1 := [ifactor/SmallFactors(t1[2], 9)]; sol := sol*t1[1] end do;
    r := t1[2];
    t1 := igcd(r,
    156875691403555582337133430528216188287394693912194151633377671884410910641\
    707088142206722656172041499289847016341552341593310179755914122207219762871\
    400408798550366225682947994604979120310160477968695489598895507697442783391\
    093147166979806891516680143894488662727890442827481214859668827865642378141\
    133808158952216599780042393649160279542709681003135175834842304947903880691\
    929390853898035416645532901518070999905642723904715497637027249406145833711\
    604895769322921453869610428677266739849701864340825108110781609586648109771\
    108388041433729417710970141607587435904413742090162284440150597621590351531\
    858203419696922860233784418251767712992128403730272471240756320716529825731\
    752095949840723303930826519812228461613036786877);
    while t1 <> 1 do sol := sol*ifactor/ifact1st(t1); r := iquo(r, t1); t1 := igcd(t1, r) end do;
    if r <> 1 then
        if nargs = 1 then
```

```
            if _Env_ifactor_easy = true and 30 < length(r) then
                ifactor/bottom := ifactor/easy; t1 := ifactor/ifact0th(r)
            else
                ifactor/bottom := ifactor/mpqsmixed; t1 := ifactor/ifact0th(r)
            end if
        else
            ifactor/bottom := cat(ifactor/, args[2]); t1 := ifactor/ifact0th(r, args[3..nargs])
        end if;
        if t1 <> FAIL then sol*t1 else FAIL end if
    else
        sol
    end if
end proc
```

Entrées sorties :

Manipulation de dossiers

currentdir permet d'afficher le dossier de travail courrant mais aussi de changer de dossier de travail. La syntaxe est :

- **currentdir()** : se contente d'afficher le nom du doddier courant
- **currentdir("CheminDossier")** : déplace le dossier de travail vers le dossier CheminDir, le dossier doit exister et son nom doit être fourni en tant que chaine de charactère. La commande retourne le dossier du dossier courant. Une fois fixé le dossier de travail, on peut se referer aux fichiers qui s'y trouvent par leurs chemins relatifs, ie par leur noms sans arborescence.

```
> currentdir();
                                    "/home/sadik"
> currentdir("/media/data/Music");
                                "/media/data/Music"
> currentdir();
                                "/media/data/Music"
```

mkdir permet de créer un dossier.

```
> mkdir("/home/sadik/MapleWork");
> currentdir("/home/sadik/MapleWork");
                                "/media/data/Music"
> currentdir();system
                            "/home/sadik/MapleWork"
```

rmdir("CheminDossier") va effacer le dossier CheminDossier.

Appel de commandes systèmes

system permet de passer des commandes au système hôte.

- **system("command") : command** est une commande qui doit être reconnu par le système et qui s'exécute en mode texte. Maple retourne le code de sortie de la

commande, qui est (sous les sytèmes de type Unix, y compris MacOS X) 0 en cas de succée de la commande, une autre valeur en cas d'echec.

- **system[launch]("program") :** Va demander au système de lancer le programme **program**, la valeur retournée est le PID (Process Identificator) du programme en cas de succé du lancement, silence radio en cas d'echec.
- **!command arguments** : La commande **command** avec se arguments est relayée au système, le résultat s'affiche dans une fenêtre Popup de Maple (ne marche pas sous Microsoft Windows).

```
> system("pwd"); # dossier courant.
```

$$0$$

```
> system[launch]("totem"); # Le lecteur vidéo par défaut du bureau gnome.
```

$$5316$$

```
> !ls /media/data;
```

$$0$$

```
> !ping 212.217.0.1 # histore de réveiller MTelecom
```

$$-1$$

Opérations de lecture/écriture dans un fichier

De loin la fonction la plus interessante presentée dans ce paragraphe. Elle offre l'opportunité de récupérer ou d'ecrire des données dans un fichier.

```
> restart;
> currentdir();
```

$$\text{"/media/data/Formation/FormInfo"}$$

```
> currentdir("/home/sadik/MapleWork");
```

$$\text{"/media/data/Formation/FormInfo"}$$

```
> file:=fopen("DataTest.txt",WRITE);
```

$$file := 0$$