Fonctions et procédures

En *Maple*, la forme la plus générale d'un programme est appelée procédure. Mais il est parfois possible de recourir à une syntaxe abrégée.

I. Fonctions simples

On pourra utiliser la syntaxe abrégée pour une fonction pouvant prendre plusieurs arguments, mais retournant un seul résultat :

nom **:=** {**(**}*argument*{*s*}**)**} **- >** *expression* **;**

(Ce type de définition ne se prête pas à l'utilisation de variables locales.)

Rappelons qu'il est facultatif de nommer une fonction. *Maple* permet les fonctions "anonymes".

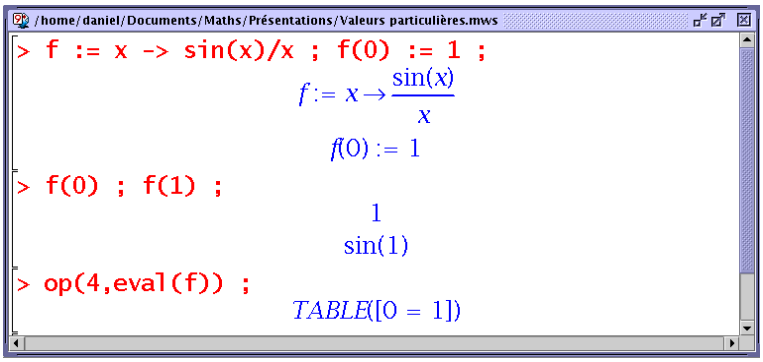
Valeurs particulières

Supposons qu'on veuille définir la fonction $f : x \to \sin(x)/x$ en la prolongeant[1] avec $f(0) = 1$.

- Un test peut régler le problème : **f := x -> if x=0 then 1 else sin(x)/x end ;**
- Mais il est aussi possible de poser la définition générale, puis de rajouter *ensuite* les valeurs particulières :
  f := x -> sin(x)/x ; f(0) := 1 ;

Soulignons l'importance de placer celles-ci *après* la définition générale. En effet, *Maple* crée une *table de valeurs particulières* qui est réinitialisée à chaque nouvelle définition de la fonction.

Cette table est accessible pour vérification par **op(4,eval(f))** ([2]) :



On constate la mise en mémoire de "$f(0) = 1$" dans l'exemple ci-dessus.

Maple utilise des tables de valeurs particulières pour bon nombre de ses fonctions intégrées.

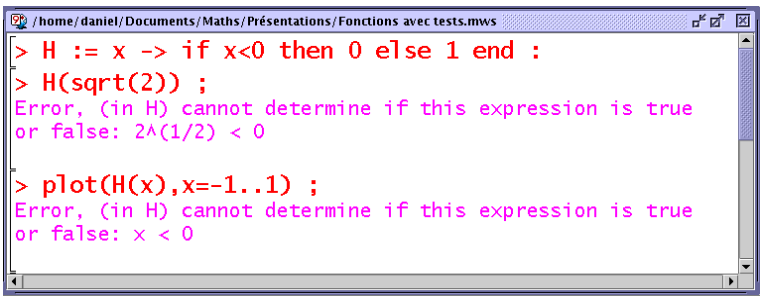
Fonctions nécessitant des tests

Si une fonction nécessite des comparaisons, on ne sera pas surpris de rencontrer des problèmes avec les valeurs *non numeric* :

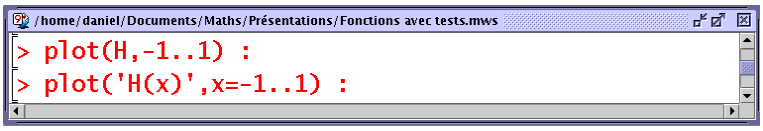
---

1 par continuité, dans ce cas

2 **eval(f)**, car sinon **f** ne serait pas remplacée par son contenu, cf. le poly. "Évaluation".

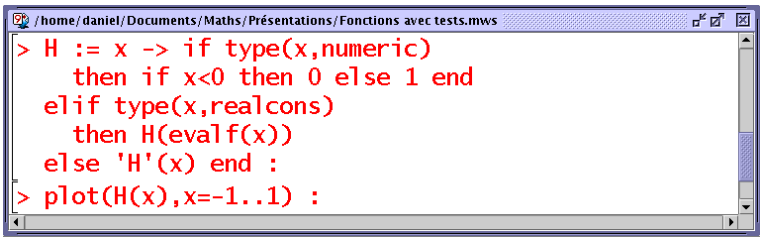


Des solutions possibles consistent à utiliser la syntaxe des fonctions (et non des expressions) ou à différer l'évaluation :



Mais l'idéal serait de prendre en compte *dès la définition de la fonction* les problèmes éventuels.

Voici la même fonction dans une programmation plus "propre" :



Les tests de type sont utiles dans ces situations.

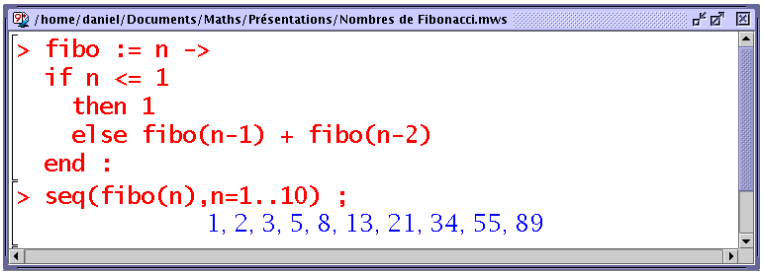
Récursivité

On peut être surpris, dans l'exemple précédent, de rencontrer dans la définition de **H** des appels à **H** elle-même. La définition demeure correcte, parce que les valeurs auxquelles on fait appel ont déjà été définies plus haut.

Cette possibilité pour une fonction de s'appeler elle-même s'appelle la *récursivité*. Elle est permise par *Maple*[3].

On y aura recours lorsque la définition non récurrente de la fonction n'apparaît pas clairement.

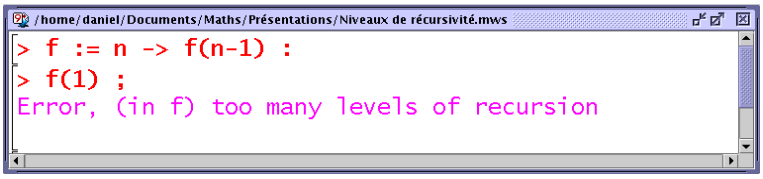
Un autre exemple avec les nombres de Fibonacci :



Cet exemple profiterait avantageusement de l'option **remember**.

Lorsqu'on utilise la récursivité, il faut s'assurer que les itérations successives vont bien conduire à un ensemble de valeurs définies *explicitement*. Si cela ne devait pas être le cas, *Maple* déclencherait un mécanisme de protection pour éviter d'entrer dans une boucle sans fin :

3 comme par presque tous les langages de programmation



II. Procédures

Rappelons la syntaxe générale :

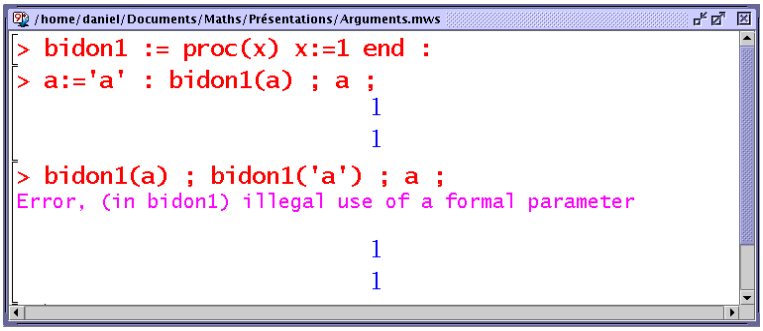
nom **:= proc(***paramètre*{*s*}**)**
{ **local** *séquence de variables locales* **;** }
{ **global** *séquence de variables globales* **;** }
{ **options** *séquence d'options* **;** }
instruction1 { **;**
...
instructionN }
end ;

Détaillons maintenant les rôles des différents éléments qui interviennent.

Paramètres (ou arguments)

Il ne faut pas les confondre avec des variables.

Leur contenu (valeur) est récupéré au début de la procédure[4].
Toute tentative de les utiliser en tant que variables se solderait tôt ou tard par une erreur :



Dans l'exemple ci-dessus, la procédure (incorrecte) **bidon1** tente d'utiliser son argument comme une variable. La première tentative (avec une variable **a** vide) semble fonctionner (elle équivaut à "**a := 1**"). Mais la deuxième revient à faire "**1 := 1**", ce qui déclenche un message d'erreur péremptoire[5].

Bien sûr, les tables, tableaux etc... ne sont évalués qu'au "dernier nom". Ceci peut permettre à une procédure mal écrite de faire illusion avec un argument de ce type. Mais il suffirait que la procédure soit appelée avec un argument explicite (*i.e.* qui ne soit pas un nom de variable) pour retomber sur l'erreur précédente.

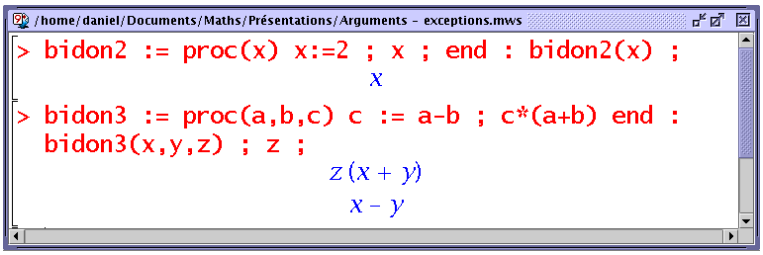
Évaluation des arguments

Notons une particularité : les arguments d'une procédure sont évalués complètement *au*

4 transmission par valeur
5 *Maple* ne badine pas avec la loi... logiciel propriétaire oblige !

début seulement, plus ensuite !

L'exemple suivant illustre ce comportement, qu'on peut facilement contourner si nécessaire avec une instruction **eval** judicieusement placée.

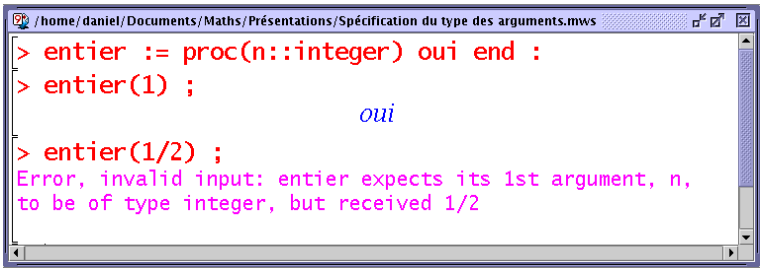


Spécification du type des arguments

On peut confier à *Maple* la vérification du type du ou des arguments d'une procédure, selon la syntaxe :

nom **:= proc(***paramètre* **: :** *type_permis* **)**
...

Une procédure ainsi mise en place n'accepte que des arguments du type défini. Si ce n'est pas le cas, elles s'arrête avec un message d'erreur :



C'est mieux que rien, mais il est bien sûr plus souple et plus élégant de déplacer cette gestion dans la procédure elle-même (avec des tests de type appropriés).

Accès aux arguments (et à leur nombre)

Lorsqu'une procédure démarre, deux variables locales sont créées pour recevoir les arguments d'appel (notamment lorsque ceux-ci sont explicites) :

- **nargs** est le *nombre* des arguments ;
- **args** est la *séquence* des arguments.

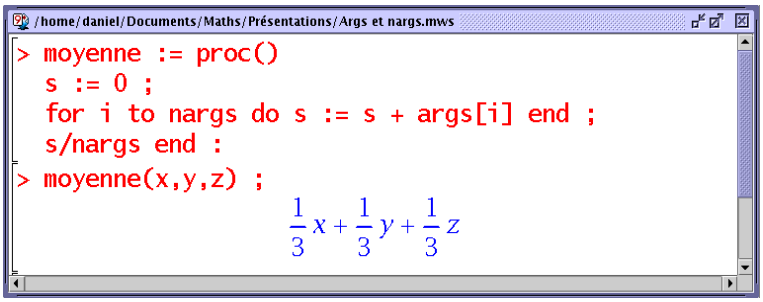
Cette dernière est soumise aux mêmes règles d'accès qu'une séquence quelconque : **args[i]** est le $i^{ème}$ argument (le premier est **args[1]**, le dernier **args[-1]** ou **args[nargs]** ([6])).

Ces deux variables fournissent le seul moyen de définir une procédure *dont le nombre d'arguments n'est pas fixé d'avance*. Pour réaliser cela, il faut déclarer la procédure *sans aucun argument*. L'accès aux arguments passe entièrement par les deux variables précédentes.

Par exemple, la procédure ci-dessous calcule la moyenne de ses arguments, quel que soit leur

6 Encore un bon exercice de prononciation.

nombre[7] :

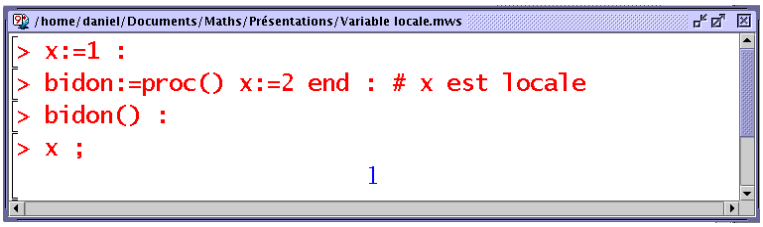


Variables locales

Une *variable locale* est définie et utilisée à l'intérieur d'une procédure. Elle n'existe que lorsque la procédure est active. Pendant ce temps, *elle annule et remplace toute variable homonyme qui existerait à l'extérieur de la procédure*.

Une éventuelle telle variable (dite *globale*) est ainsi protégée de toute modification par la procédure. En contrepartie, son contenu est *totalement inaccessible à l'intérieur de celle-ci*.

Reprenons l'exemple déjà rencontré :



Le " **x** " hors de la procédure, *global*, n'est pas modifié par l'instruction " **x := 2** " : elle ne porte que sur le **x** *local* de la procédure.

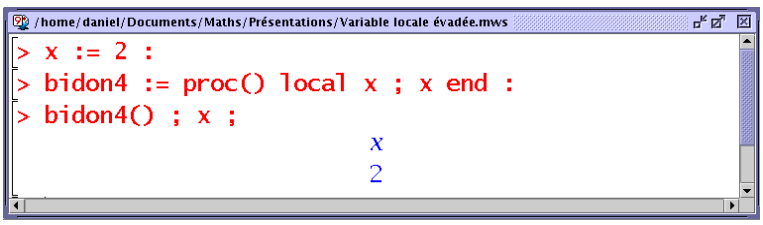
Rappelons que *Maple* considère automatiquement[8] locales certaines variables :

- celles qui apparaissent au membre de gauche d'une affectation : **x :=** ... -- c'est le cas du " **x** " dans l'exemple précédent ;
- les compteurs des boucles **for**.

On peut facilement contourner ce mécanisme en déclarant explicitement comme locale ou globale une variable. Il suffit de placer une ligne **local x** ou **global x** au début de la procédure.

On réservera les variables globales aux procédures qui doivent échanger ou partager des valeurs avec l'utilisateur (ou d'autres procédures). On en trouvera un exemple un peu plus loin.

D'un autre côté, une procédure particulièrement mal écrite peut laisser échapper une variable locale :



7 À condition bien sûr qu'il soit non nul.
8 *Maple* génère alors un message d'avertissement ("*warning*").

Ici, la variable locale " **x** " s'est échappée dans la nature. Il n'y a plus aucun moyen d'accéder à son contenu[9].

Options

Certaines options sont réservées par *Maple*. On a déjà cité l'option **builtin** qui identifie les fonctions internes de *Maple*, bloquant toute évaluation.

Nous allons détailler une option qui peut rendre de grands services, mais qui doit être utilisée avec discernement.

L'option remember

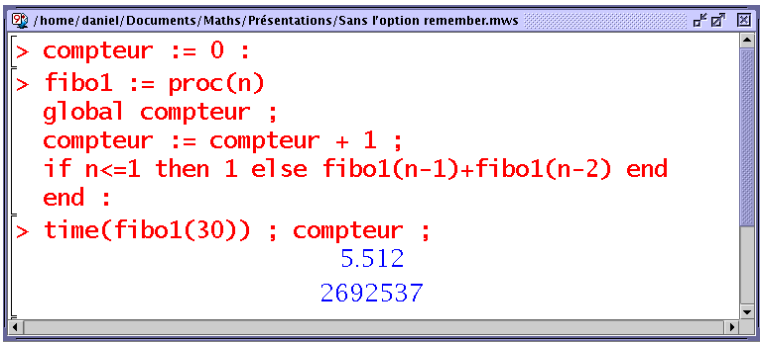
Cette option force *Maple* à "se souvenir" des valeurs déjà calculées. Ceci est réalisé en ajoutant *chaque nouvelle valeur calculée* à la table des valeurs particulières.

On conçoit donc que le coût en mémoire de cette option soit particulièrement élevé.

Elle est spécialement recommandable pour des procédures récursives qui retombent toujours sur le même ensemble de valeurs.

Les nombres de Fibonacci nous en fournissent une illustration :

- **sans l'option remember** :



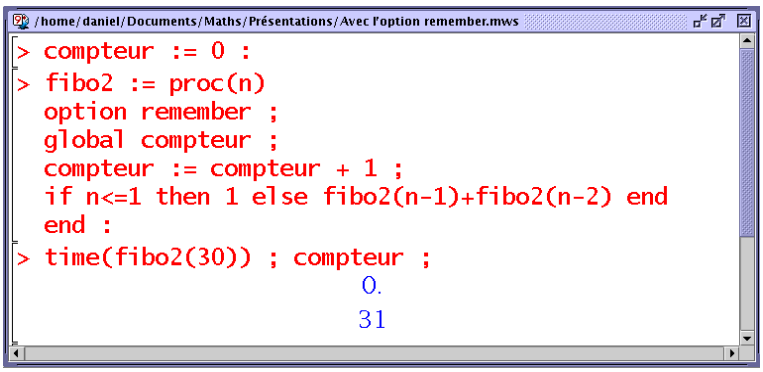
On comprend le temps nécessaire en voyant le nombre de calculs : 2 692 537 appels à la fonction **fibo1** ! En effet, pour calculer *fibo1*(30), il a fallu calculer *fibo1*(28) et *fibo1*(29), mais ce dernier a recalculé *fibo1*(28), et les 4 calculeront *fibo1*(27) etc. (Comptez !)

Une autre remarque : compteur doit impérativement être global pour deux raisons. Si ce n'était pas le cas,

- un nouveau compteur (local) serait initié à chaque appel récursif de **fibo1** ;
- celui-ci n'étant pas initialisé, la ligne **compteur := compteur + 1** déclencherait immédiatement une erreur d'affectation récursive.

- **avec l'option remember** :

9 C'est fini pour elle...



La seule différence est la présence de la ligne **option remember**. Petite cause, grands effets : la procédure effectue maintenant (en un temps non mesurable[10]) seulement 31 calculs : pas un de plus que nécessaire (*fibo2*(*n*) pour *n* allant de 0 à 30).

Répétons-le, on aurait tort d'utiliser aveuglément cette option pour toute procédure récursive.

Par exemple, une suite récurrente $u_{n+1} = f(u_n)$ se programme naturellement récursivement, mais ne revient typiquement jamais sur des valeurs déjà rencontrées. Dans un tel cas, utiliser l'option **remember** ne serait qu'un pur gaspillage de mémoire.

10 Nous n'avons pas réussi à trouver une valeur de *n* donnant à la fois un temps d'exécution raisonnable sans **remember** et significatif avec, *Maple* étant plus rapide sous Linux que sous Windows (à matériel égal) d'un petit facteur... 25.