Chapitre VII

PROCÉDURES

1 Le jeu des procédures

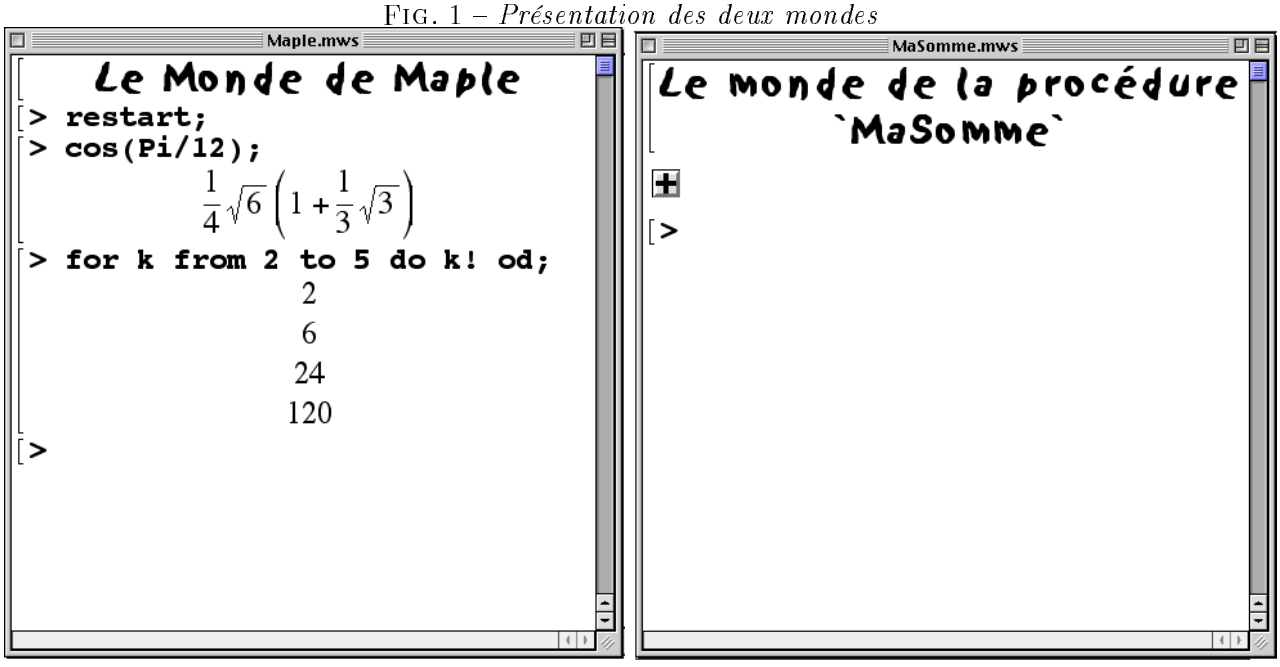
1.1 Présentation

TOUT SE PASSE DANS UN UNIVERS où deux mondes se cotoient pacifiquement. Il y a le monde auquel nous sommes maintenant habitué, le monde de `Maple`. Et il y a un monde voisin, le monde de la procédure `MaSomme`.

Les habitants du monde de `Maple` travaillent durement, et doivent parfois faire des tâches répétitives. Les habitants du monde de la procédures sont de petits individus ne se plaignant jamais de la quantité de travail à effectuer.

Les deux mondes sont bien distincts. Aucun habitant du monde de la procédure n'irait s'installer dans le monde de `Maple`, et l'inverse est formellement interdit. Mais les deux mondes, même s'ils sont cloisonnés, ne sont pas complètement indépendants. Les échanges sont possibles, mais sont très codifiés. Le principe de ces échanges est toujours le même : les habitants du monde de `Maple` demandent aux habitants du monde de la procédure d'effectuer un travail bien précis. C'est d'ailleurs le seul travail que les habitants du monde de la procédure sont capables d'effectuer.

FIG. 1 – *Présentation des deux mondes*



On ne peut pas transgresser le code d'échange, qui se résume pour nous aux règles suivantes :

- Pour le monde de `Maple`
  - On peut faire appel au monde de la procédure en tapant son nom.
  - On peut (on doit) fournir au monde de la procédure des valeurs, grace aux arguments de la procédure.
- Pour le monde de la procédure `MaSomme`
  - Dans le monde de la procédure, on est au service des habitants du monde de `Maple`.
  - On fournit au monde de `Maple`, à chaque fois qu'il fait appel à nous, la valeur de la dernière expression évaluée.
  - On peut appeler les autres mondes de procédures à l'aide, en tapant leurs noms.

– Notre travail est complètement invisible pour les habitants du monde de `Maple`. Nous ne pourrons en tirer aucune gloire.

Le but du jeu est de définir correctement le travail des habitants du monde de la procédure, pour qu'ils effectuent le travail que leur confient les habitants du monde de `Maple`. Et plus généralement, c'est de créer de nouveaux mondes de procédures, auxquels nous demanderons d'effectuer des tâches.

1.2 Premier exemple

Définissons la procédure `MaSomme`, dont le travail est de calculer pour un n donné la somme $\sum_{i=0}^{n} \frac{1}{i!}$, sans utiliser la fonction `sum`.

FIG. 2 – *Définition de la procédure*

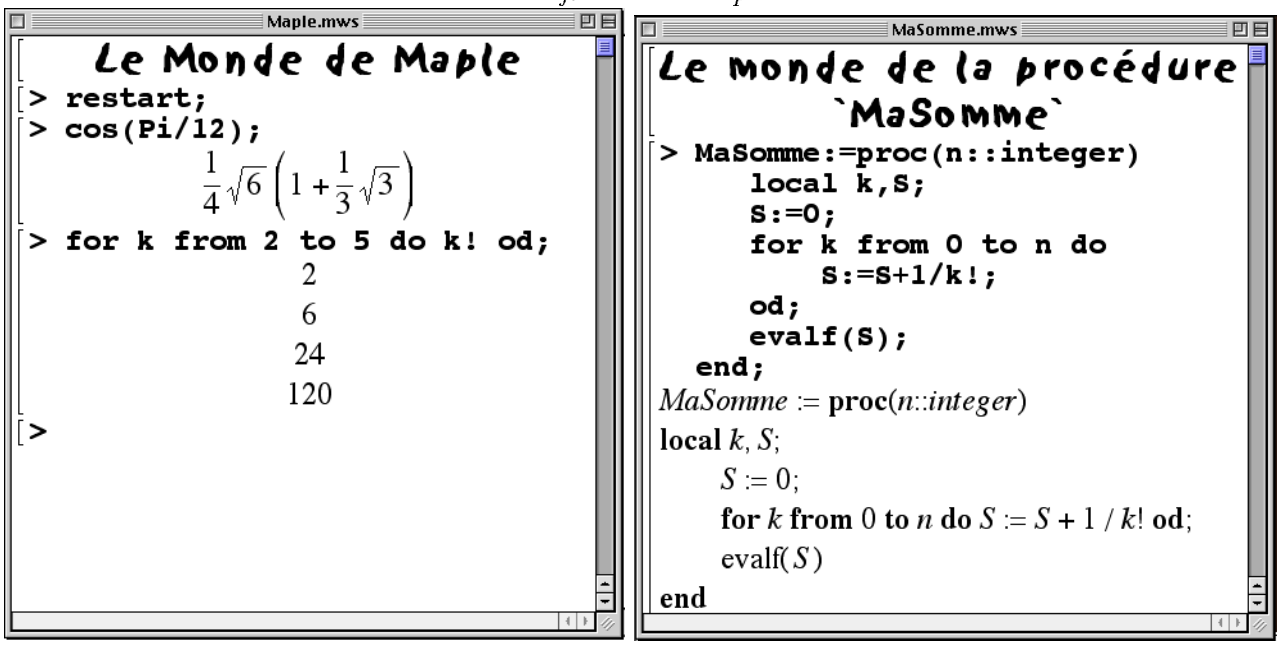
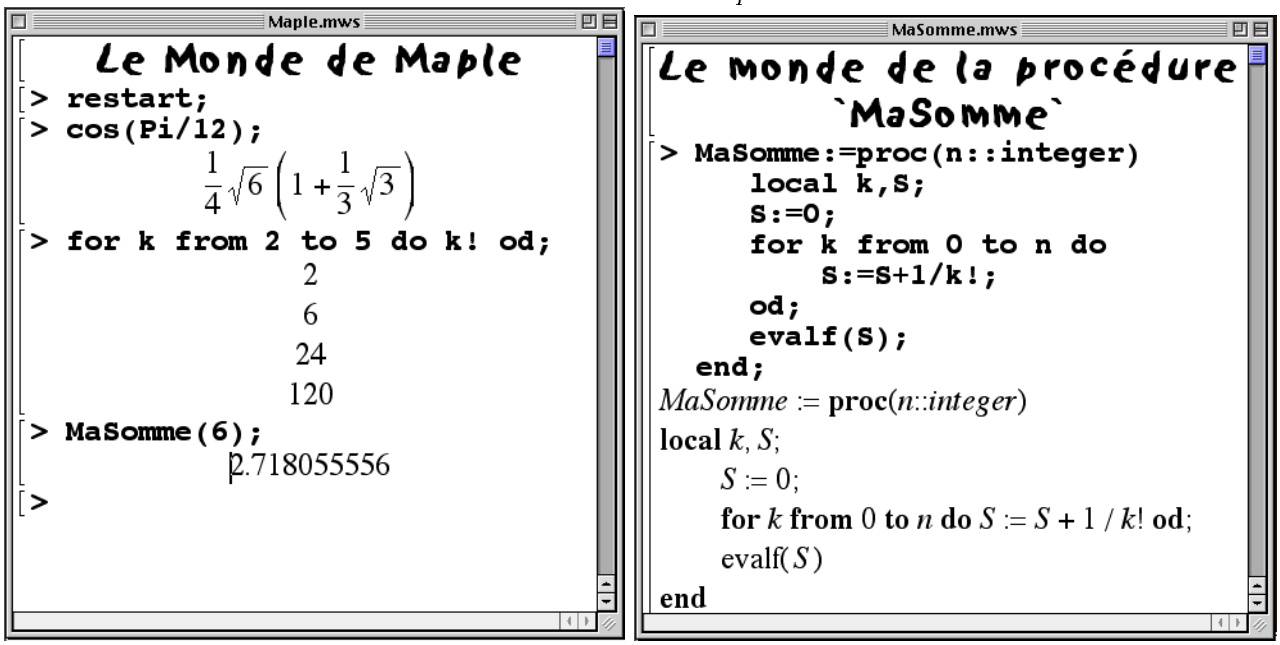


FIG. 3 – *Utilisation de la procédure*



2 Syntaxe, définitions

Retrouvons dans ce paragraphe un vocabulaire plus conventionnel. Écrire une **procédure**, c'est définir une nouvelle commande, une nouvelle fonction Maple. Pour être utilisable, une procédure est toujours affectée à une variable. Elle est de type `procedure` et suit la syntaxe suivante :

Séquence de début	`>`*Nom* `:=proc(`*arg1* `::` *type1*, ... ,*argn* `::` *typen* `)`
	`local` *i,j* `;`
	`(global` *a,b,c* `;) # à éviter`
	`(option remember;) # optionnel`

Corps de la procédure	*Instructions*
Séquence de fin	`end;`

Nous allons étudier les différents éléments qui composent la définition d'une procédure.

2.1 Arguments

Les **arguments** (parfois appelés **paramètres**) d'une procédure sont évalués au moment de l'appel et ne pourront pas être modifiés au cours de l'exécution de la procédure. On peut retarder leur évaluation en utilisant des apostrophes (`' '`). On effectue un contrôle de type de l'argument par l'utilisation du double deux points (`::`).

2.2 Variables

Les **variables locales** d'une procédure seront déclarées à l'intérieur de la procédure. Ces variables servent uniquement pendant l'exécution de la procédure et n'existeront plus une fois la procédure exécutée. Une variable locale peut porter le même nom qu'une variable existant en dehors de la procédure : Maple ne fait pas la confusion entre les deux, et la valeur de la variable extérieure n'est pas modifiée à l'issue de la procédure.

Toutes les variables doivent être déclarées au début de la procédure. Nous n'utiliserons que des variables locales et des paramètres, et éviterons d'utiliser des variables globales, hormis les constantes prédéfinies (`Pi ...`) et les autres procédures.

2.3 Remarques

On prendra soin de toujours écrire une procédure sur plusieurs lignes, le retour à la ligne sans exécution se faisant en frappant simultanément MAJ+ENTRÉE. La définition d'une procédure commence à `proc` et se termine à `end`. Oublier mot `end` provoquerait une erreur, et mettre un point-virgule (`;`) après `proc` n'aurait pas de sens.

Une procédure renvoie toujours la dernière expression évaluée. Rien d'autre. On verra sur des exemples des techniques pour qu'une procédure renvoie plusieurs expressions.

3 Exemples

3.1 Calcul des coefficients binomiaux

```
>binom:=proc(n::posint,k::posint)
   if k>n then 0
   else n!/(k!*(n-k)!)
   fi;
  end;
>binom(3,6);
>binom(6,3);
```

3.2 Procédures itératives: exemple de factorielle

```
>mafactorielle1:=proc(n::posint)
   local f,k;
   f:=1;
   for k from 1 to n do
       f:=f*k;
   od;
  end;
>mafactorielle1(10);
```

3.3 Procédures itératives: exemple d'une suite définie par récurrence

On considère la suite définie par $\begin{cases} u_0 = 1 \\ u_{n+1} = \frac{6u_n+5}{u_n+2} \quad \forall n \in \mathbb{N} \end{cases}$

Compléter la procédure ci-dessous pour qu'elle permette de calculer les termes de la suite.

```
>u:=proc(n::posint)
   local

   for k from 1 to n do
       resultat:=(6*resultat+5)/(resultat+2);
   od;
   resultat;
  end;
>u(50);
```

3.4 Procédures récursives: exemple de factorielle

On appelle procédure récursive une procédure qui s'appelle elle-même.

```
>mafactorielle2:=proc(n::posint)
   option remember;
   if n=1 then 1
         else n*mafactorielle2(n-1);
   fi;
  end;
>mafactorielle2(10);
```

3.5 Procédures récursives: exemple d'une suite définie par récurrence

On reprend la suite précédente.
Compléter la procédure ci-dessous pour qu'elle permette de calculer les termes de la suite.

```
>u2:=proc(n::posint)
   option remember;
   if n=1 then
         else
   fi;
  end;
>u2(50);
```

4 Exercices

VII.1 Écrire une procédure `menu` d'un argument `jour` de type `string` qui donne le menu du jour.

VII.2 On dit qu'un entier n est **parfait** si et seulement si il est égal à la somme de tous ses diviseurs stricts. Par exemple 6 et 28 sont parfaits.

1. Écrire une procédure `estparfait` d'un argument de type `integer` qui détermine si un nombre est parfait ou non. Le résultat sera de type booléen.
2. Écrire une procédure `listeparfait` d'un argument de type `integer` qui utilise la procédure précédente et dresse la liste de tous les nombres parfaits plus petit que l'argument.
3. Utiliser la procédure `listeparfait` pour déterminer tous les entiers parfaits plus petits que 500.

VII.3 Algorithme de Syracuse

L'algorithme de Syracuse est le suivant : Étant donné un entier n, on remplace n par $\frac{n}{2}$ lorsque n est pair et par $3n+1$ lorsque n est impair. On réitère alors le procédé jusqu'à obtenir 1 [1].
Écrire une procédure `Syracuse` d'un argument entier `n`, qui donne le nombre d'étapes nécessaires pour obtenir 1.

VII.4 Les petits devant et les grands derrière

Écrire une procédure qui associe à toute liste de nombres réels la liste où les nombres ont été classés par ordre croissant.

VII.5 Suite de Fibonacci

Écrire deux procédures, l'une itérative, l'autre récursive, permettant de calculer le $n^{\text{ème}}$ terme de la suite définie par :

$$\begin{cases} u_0 = u_1 = 1 \\ u_{n+2} = u_{n+1} + u_n \quad \forall n \in \mathbb{N} \end{cases}$$

VII.6 Polnômes de Chebychev

Soit $T_n \in \mathbb{R}_n[X]$ défini par :

$$T_0 = 1, T_1 = X \text{ et } \forall n \geqslant 2, T_n = 2XT_{n-1} - T_{n-2}$$

1. En utilisant un procédé itératif, calculer T_{11}.
2. Écrire une procédure récursive permettant de calculer $T(n)$ pour d'assez grandes valeurs de n (attention cependant à ne pas surestimer la capacité des machines !).
3. Mettre en évidence l'intérêt de l'option `remember`.

VII.7 Écrire une procédure permettant de calculer les coefficients binomiaux par une méthode récursive.

VII.8 Entiers amiables

Soit a un entier naturel. On appelle **diviseur propre de** a tout diviseur de a différent de a.
Deux entiers sont dits **amiables** si et seulement si chacun d'eux est égal à la somme des diviseurs propres de l'autre.
Écrire une procédure permettant de déterminer tous les couples d'entiers amiables inférieurs ou égaux à 1500.

1. Il n'a pas été démontré que l'on aboutit toujours à 1, mais on n'a jamais trouvé de contre-exemple.