Maple expliqué

Francis Sergeraert

Chapitre 1

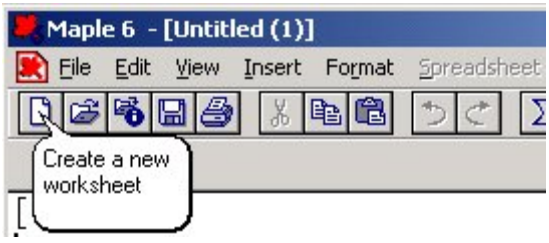
Premières sessions Maple

1.1 À propos du système d'exploitation utilisé.

Le logiciel Maple est utilisable sur tout poste de travail Windows, Mac ou Unix, après une *installation* du logiciel, très simple, surtout sous Windows et Mac. Une fois le logiciel installé, les procédures d'utilisation sont les mêmes, à des détails minimes près, qui ne concernent *jamais* l'essentiel de ce qui est fait avec ce logiciel. Les micro-différences sont dues aux traditions plus ou moins bien définies pour chaque environnement, et aussi aux ressources disponibles dans chaque système d'exploitation. Pour ne pas alourdir inutilement la présentation, on ne considèrera ici que le cas de *Maple utilisé sous Windows*.

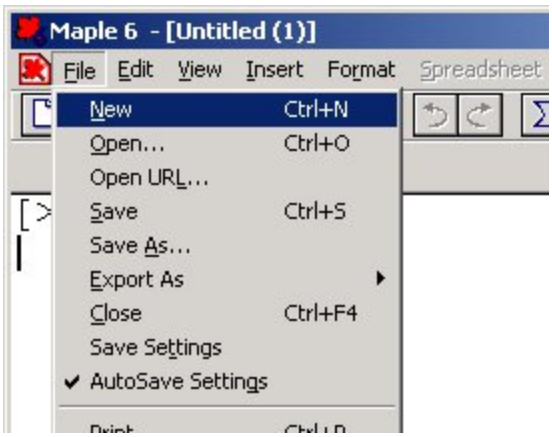
Pour rassurer le lecteur angoissé, examinons immédiatement une telle différence afin d'en mettre en évidence la futilité[1]. Fréquemment l'utilisateur doit ouvrir une nouvelle fenêtre, il faut dire une nouvelle *feuille de travail* (worksheet). L'utilisateur Windows a trois méthodes équivalentes pour le faire :

1. Il peut cliquer sur le premier bouton de la barre d'outils, dont l'icône représente une feuille vierge écornée en haut à droite. La feuille vierge souhaitée s'ouvre aussitôt.



2. Il peut dérouler le menu **File**, à l'aide de la souris ou s'il préfère à l'aide de la touche **Alt-F** ; ensuite il sélectionne l'option **New** de ce menu, à nouveau à l'aide de la souris ou bien de la touche clavier **N**. La nouvelle feuille de travail est alors ouverte.

[1] Le lecteur débutant complet n'est pas concerné par le reste de cette section.



3. L'indication **Ctrl-N** sur la ligne **New** du menu informe l'utilisateur qu'il aurait obtenu exactement le même résultat avec la touche **Ctrl-N**. C'est, indépendamment du logiciel utilisé, le standard Windows en l'occurence ; c'est facile à retenir et c'est de loin la méthode la plus rapide.

Pour une session Maple sous Unix, le processus est exactement le même à ceci près que la troisième possibilité est exclue. L'utilisateur Unix prend donc vite l'habitude d'utiliser la première méthode ci-dessus, en utilisant le bouton «page vierge».

D'une façon générale les différences entre Maple-Windows, Maple-Mac et Maple-Unix ne concernent que des points très accessoires de gestion d'édition, de gestion de fenêtres et de fichiers. Presque toujours les processus particuliers à tel système d'exploitation respectent les habitudes de ces systèmes ou encore, cas de Maple-Unix, les traditions Emacs, considérées comme les plus naturelles pour les universitaires Unix. Par exemple, sous Maple-Unix, la touche **Ctrl-N**, au lieu d'ouvrir une fenêtre, descend le curseur d'une ligne sur la feuille de travail active, conformément à la tradition Emacs.

On ne peut pas ne pas signaler que l'ensemble des raccourcis claviers standard de Windows, bien conçu, rend l'ergonomie Maple sensiblement meilleure sous Windows que sous Unix. On observe un autre inconvénient sous Maple-Unix : il est relativement fréquent que certains raccourcis claviers en principe valides sous Unix deviennent soudain inopérants, sans raison connue. Le phénomène a été constaté sous des systèmes Unix différents, en particulier Linux. Espérons que quand le présent manuel paraîtra, ce problème sera «fixé», selon l'anglicisme à la mode.

Mais pour tous les points importants, notamment les points mathématiques, les trois environnements sont rigoureusement identiques et la question du système d'exploitation utilisé est dès lors sans objet.

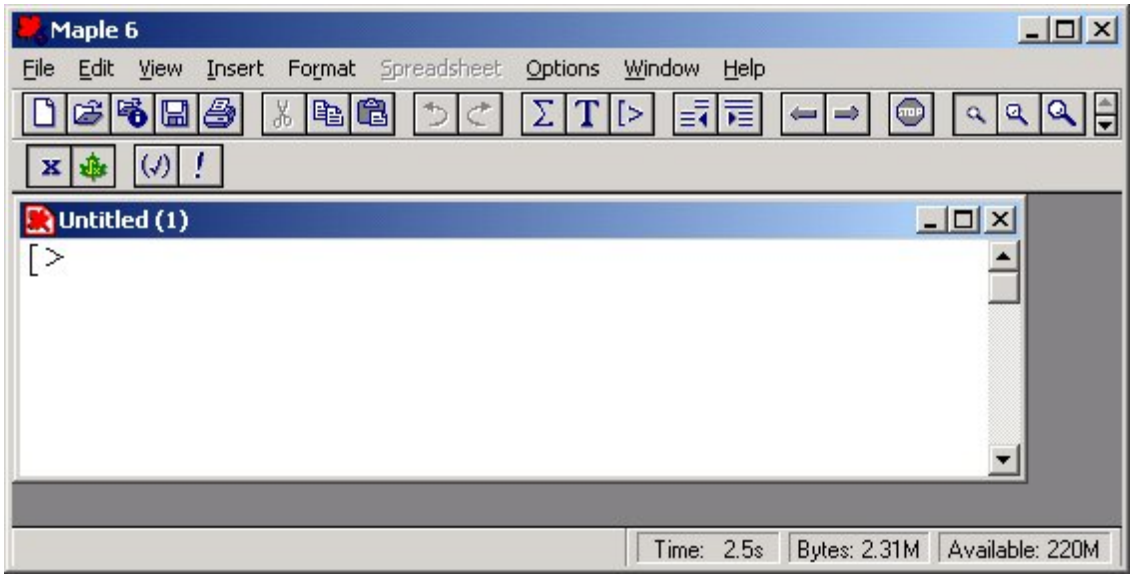
1.2 Description de la fenêtre Maple.

On peut lancer une *session* Maple de plusieurs façons :

1. En cliquant sur l'*icône* appropriée dans un dossier ou sur le bureau Windows ;

2. En déroulant les menus accessibles à partir du bouton **Démarrer** jusqu'à atteindre l'option citant «Maple 6»[2] ;
3. En entrant la *commande*[3] appropriée dans une fenêtre de dialogue avec le système, par exemple la fenêtre «Exécuter».
4. En utilisant un *raccourci* clavier, selon la méthode banale sous Windows ; c'est cette méthode qui est la plus commode si on envisage un usage relativement fréquent de Maple.

Une *fenêtre* Maple est alors ouverte par le système où l'utilisateur va pouvoir dialoguer avec le programme Maple. Cette fenêtre est organisée comme suit.



Le haut de la fenêtre est un bandeau contenant le titre :

Maple 6

Les *releases* (versions) actuellement en usage sont généralement les releases 4, 5, 6 ou 7, la dernière étant la plus récente[4]. Les différences entre les releases 4 et 5 sont assez significatives[5], la release 4 devant devenir rapidement obsolète.

Du côté droit du même bandeau, on trouve les mini-boutons usuels Windows, à savoir, de gauche à droite :

1. Le bouton _ servant à faire disparaître provisoirement la fenêtre de l'écran, en ne conservant que le bouton barre des tâches pour une restauration ultérieure ;

[2] Ou l'indication correspondante pour une version différente, plus ancienne ou plus récente.

[3] Probablement «`C:\Program Files\Maple 6\BIN.WNT\wmaple.exe`», berk !

[4] Ce manuel a été préparé avec la Release 6 ; pendant sa préparation, la release 7 est parue ; les différences avec la release 6 sont, pour ce qui concerne notre sujet, négligeables.

[5] Ces différences ne sont pas anodines ; sur beaucoup de questions la release 5 est beaucoup plus commode que la release 4. En particulier la release 5 permet la *programmation fonctionnelle*, outil capital pour beaucoup de questions, qu'on examinera en temps utile, voir LL.

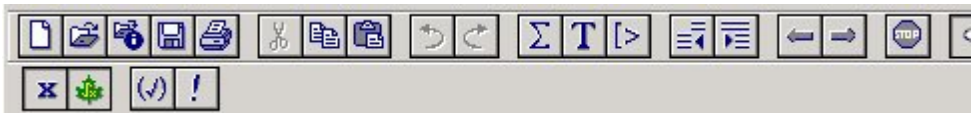
2. Le bouton  servant à agrandir la fenêtre en plein écran ou au contraire  à la réduire en écran partiel ;
3. Le bouton  terminant définitivement la session Maple.

L'usage de ces boutons est banal et il n'y a pas lieu d'en parler. Si la fenêtre est en écran partiel, sa taille et sa position peuvent être modifiées par les procédures habituelles.

Sous le bandeau titre, on trouve le *bandeau des menus* :

Le nombre et la nature des menus dépendent du contexte mais, en cours de travail ordinaire, on voit les menus **File**, **Edit**, **View**, **Insert**, **Format**, **Options**, **Windows** et **Help**. Les deux menus **File** et **Help** sont toujours présents. On déroule un menu avec la souris en cliquant sur son entête ou en entrant au clavier la combinaison **Alt- ?** appropriée, par exemple **Alt-F** pour le menu **File**. Quand le menu souhaité est déroulé, on active l'option souhaitée par un clic de souris ; on peut aussi, après avoir sélectionné l'option souhaitée à l'aide de la touche directionnelle «↓», activer cette option avec la touche <**Entrée**>[6]. Certaines options ouvrent à leur tour un sous-menu. Comme déjà expliqué, de nombreuses options de menus ont des *raccourcis claviers* indiqués en face de l'option correspondante sur le menu, permettant d'obtenir le même résultat souvent plus facilement.

On trouve ensuite, sous le bandeau des menus, la *barre d'outils* :



C'est un ensemble de boutons-icônes disposés sur deux lignes, représentant de façon imagée telle ou telle action qu'on peut obtenir aussi bien en général par menu déroulant ou par raccourci-clavier, mais pour donner une chance supplémentaire à l'utilisateur de trouver ce qu'il cherche, les actions les plus importantes ont presque toutes un équivalent bouton. L'action correspondante est activée en cliquant sur le bouton approprié, quand il existe. La deuxième ligne de la barre d'outils dépend du contexte.

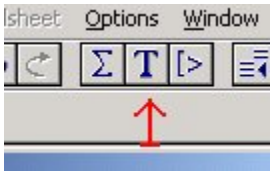
Certains clics intempestifs ou accidentels mettent fréquemment en sérieuse difficulté les utilisateurs débutants, notamment quand le mode **Texte** est activé inopinément par le bouton  et qu'on ne sait pas le gérer.

Conseil 1 — *Ne cliquez pas sur les boutons de la barre d'outils tant que vous n'avez pas raisonnablement compris leur usage. Pour ce faire, faites en d'abord* au besoin *l'expérience* séparément *dans une micro-session Maple dédiée uniquement à cette question. Sinon, abstenez-vous prudemment.*

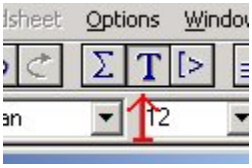
[6] <**Return**> sur les claviers anglais.

Il peut vous être utile dans ce registre d'activer l'option **Balloon Help** du menu **Help** ; si ceci a été fait, le passage du pointeur souris sur un bouton provoque l'affichage d'une bulle explicative sur la nature du bouton. On avait montré un exemple page 1. Cette explication n'est pas toujours suffisante pour le débutant.

Certains boutons ont deux positions ; l'une *off* (bouton en attente) indique que l'option correspondante est pour le moment inactive, l'autre *on* (bouton «poussé») indique au contraire que l'option est active. Par exemple le mode **Text** est inactif si le bouton  est *off* :



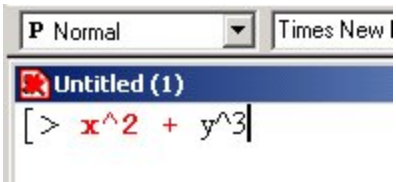
Au contraire le mode **Text** est actif si le bouton  est *on* :



Noter que la ligne inférieure de la barre d'outils, la partie dite *dépendante du contexte*, dépend de l'option en cours.

Si vous êtes utilisateur débutant, évitez soigneusement d'utiliser le mode **Text**, servant à préparer des commentaires sophistiqués pour les calculs prévus. Le chapitre **LU** est entièrement consacré à ces questions, assez techniques, mais puissantes en moyens d'expression. Dans la première phase où seuls les «calculs» Maple vous intéressent, le mode **Text** ne vous concerne pas.

Il peut arriver que vous activiez *par mégarde* le mode **Text** ! Vous vous en apercevez par le fait que les caractères entrés au clavier apparaissent mystérieusement noirs sur l'écran :



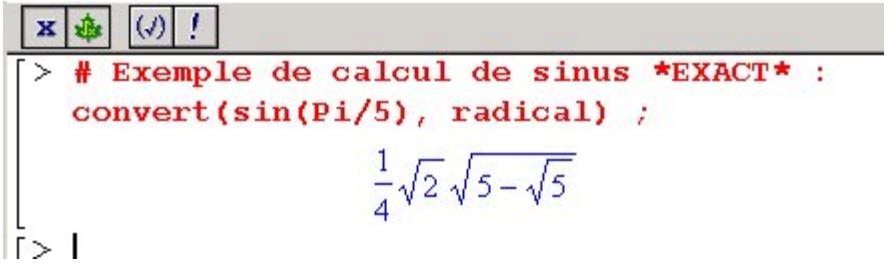
Le plus simple est alors d'utiliser, plusieurs fois au besoin, la touche **Ctrl-Z** (Undo Typing) jusqu'à revenir au mode standard. Autre solution, on balaie la zone étrange avec la souris *en débordant un peu sur la zone standard* :

puis on utilise la touche <**Suppr**> pour la faire disparaître.

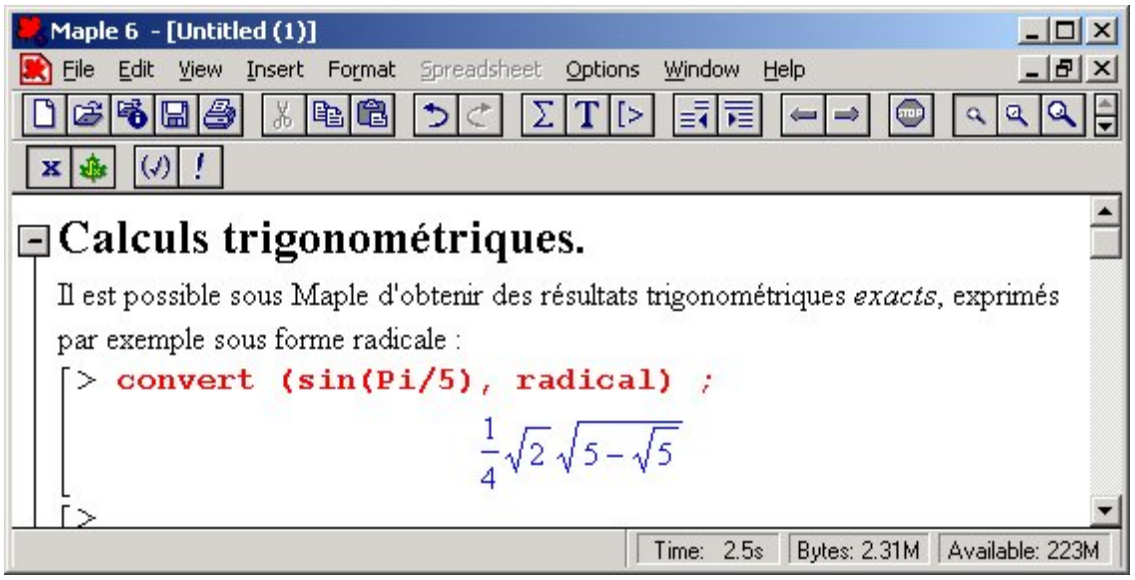
D'où l'importance du conseil suivant.

Conseil 2 — *Pendant la phase de première initiation à Maple, n'utilisez pas le mode* ***Text****. Vous pourrez insérer, si vous y tenez absolument, des commentaires dans vos commandes Maple par le procédé de tous les langages de programmation : sur toute ligne de commande, le caractère* #, sans changer de «mode», *indique à Maple que le texte qui le suit sur sa ligne est un commentaire. Ceci suffit largement pendant l'initiation.*

Exemple de simple commentaire :

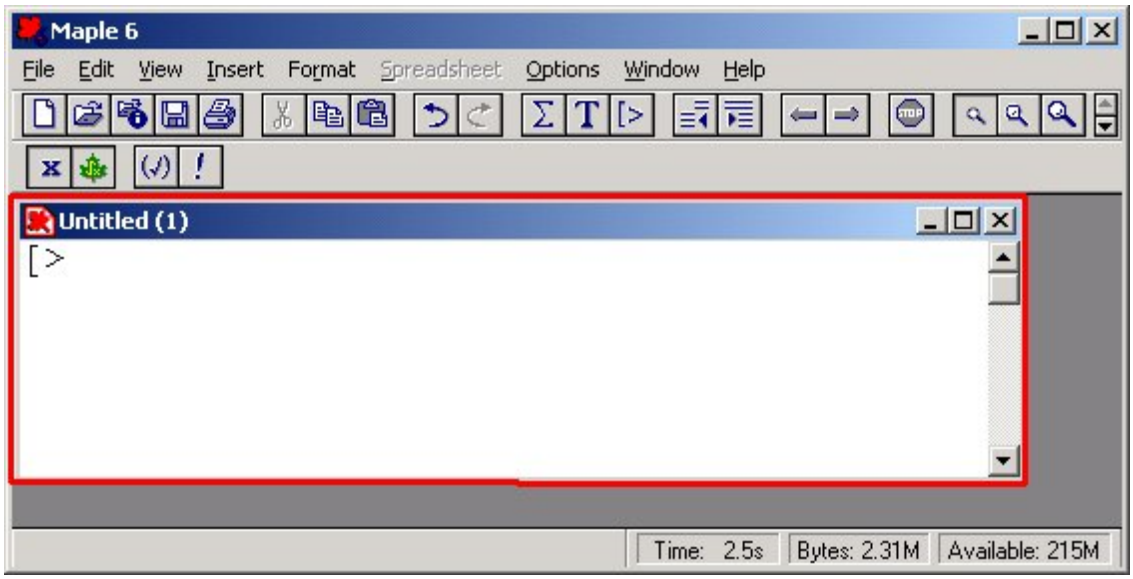


Exemple de commentaire sophistiqué :

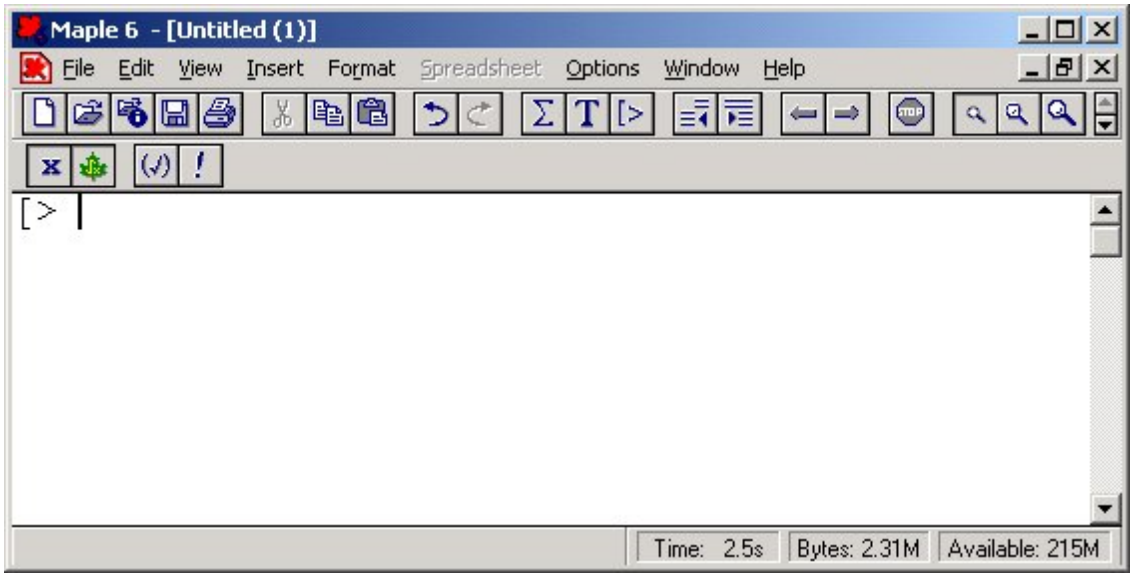


Selon le contexte, d'autres mini-fenêtres s'ouvrent dans l'entête de la fenêtre Maple, par exemple pour choisir une police et sa taille quand on est en mode **Texte**, pour modifier l'angle de vue d'un graphique 3D. Ces questions seront examinées en temps utile.

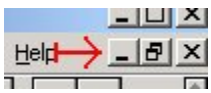
La partie principale de la fenêtre Maple est la *feuille de travail* (worksheet) qu'on a encadrée ici en rouge :



Une autre version de la disposition générale peut être la suivante, où la feuille de travail est *plein écran* par rapport à la fenêtre Maple :



Il est important de bien comprendre dans un tel cas que les boutons Windows *de la feuille de travail* sont affichés sur la partie droite du bandeau des menus, juste au-dessous des boutons analogues pour la fenêtre Maple *entière*, méthode générale Windows très commode à l'usage mais qui surprend toujours les débutants. On les indique ici à l'aide de notre flèche rouge :



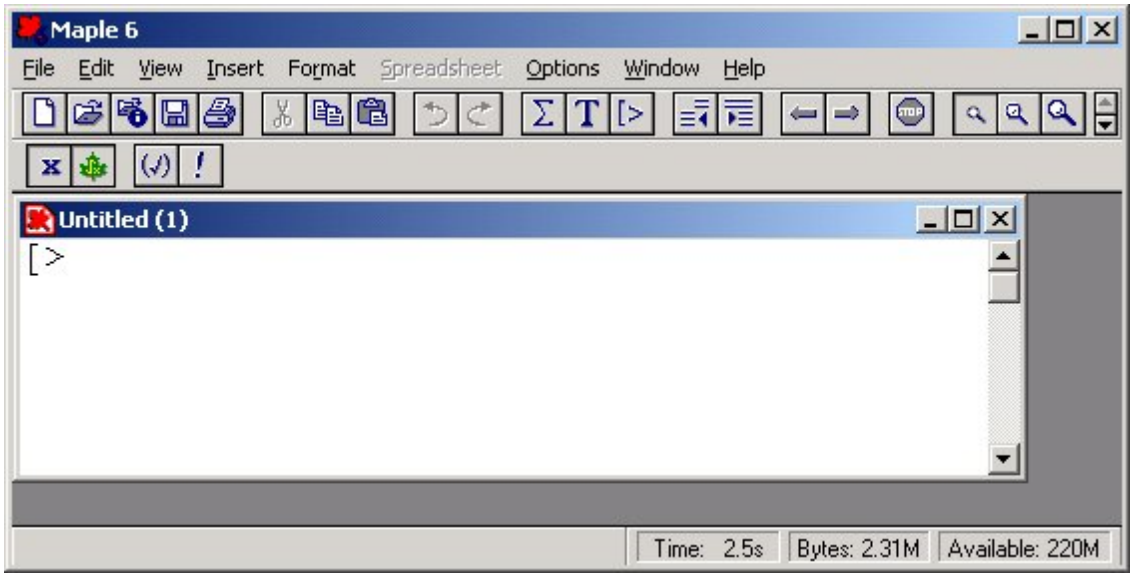
On passe de la version écran partiel à la version plein écran de la *feuille de travail* en cliquant sur *son* bouton Windows et inversement on revient à l'affichage écran partiel en cliquant sur le bouton . On procède de même pour la fenêtre *entière* Maple par rapport à l'écran du moniteur en cliquant sur les boutons analogues *de la fenêtre Maple* cette fois, à l'extrême nord-est de la fenêtre.

Le bandeau inférieur de la fenêtre Maple donne à droite un certain nombre d'indications, sur le temps CPU déjà consommé, sur l'espace mémoire déjà mobilisé, sur l'espace disque disponible. Sauf calculs d'envergure, ces indications sont de peu d'utilité.

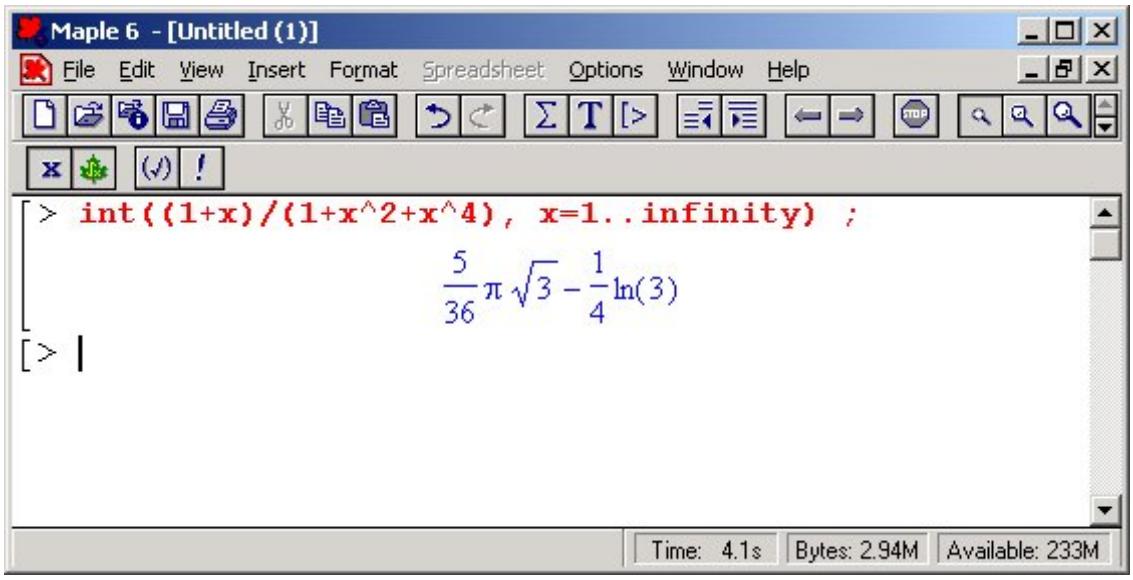
1.3 Première session.

Le programme Maple permet comme la quasi-totalité des logiciels modernes de mener de front plusieurs travaux, notamment par l'intermédiaire d'un système de *multi-fenêtrage*. Maple suit les traditions Windows maintenant bien établies. On en évoque ici quelques aspects, juste pour indiquer au lecteur les grandes lignes des disponiblités, assez secondaires par rapport à notre sujet. L'expérience montre que les préférences des utilisateurs à ce propos sont très variables : chacun son style.

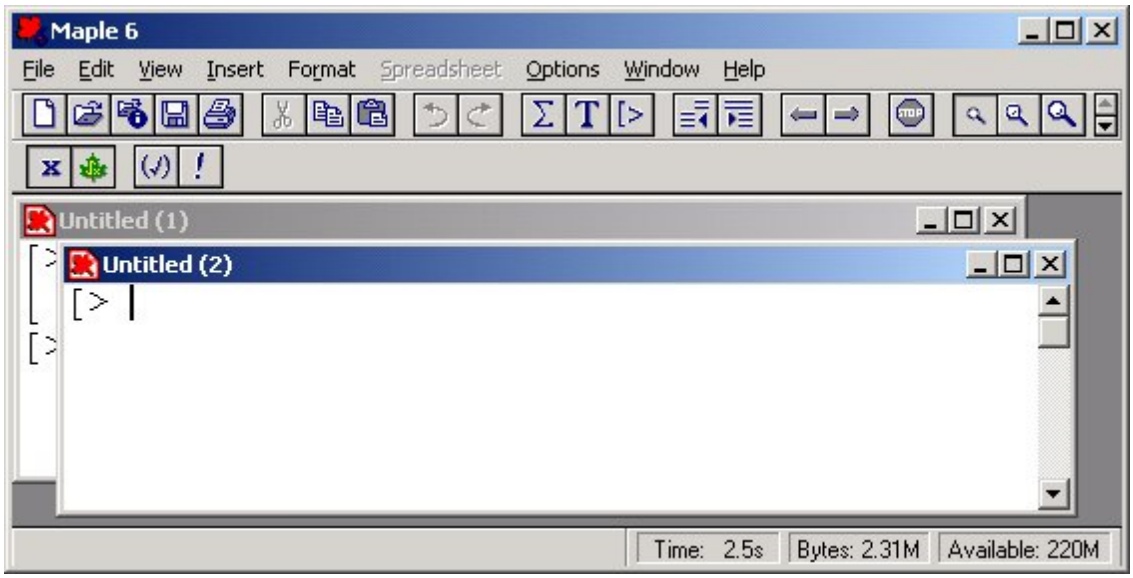
Considérons la situation initiale standard :



Si l'utilisateur souhaite connaître une valeur *numérique* de $\sin(\pi/5)$, il entre au clavier «`evalf(sin(Pi/5))` ;», puis la touche <**Entrée**> ; ne pas oublier la majuscule `Pi` et le point-virgule ' ;' terminal avant la touche <**Entrée**>. Il obtient le résultat suivant :

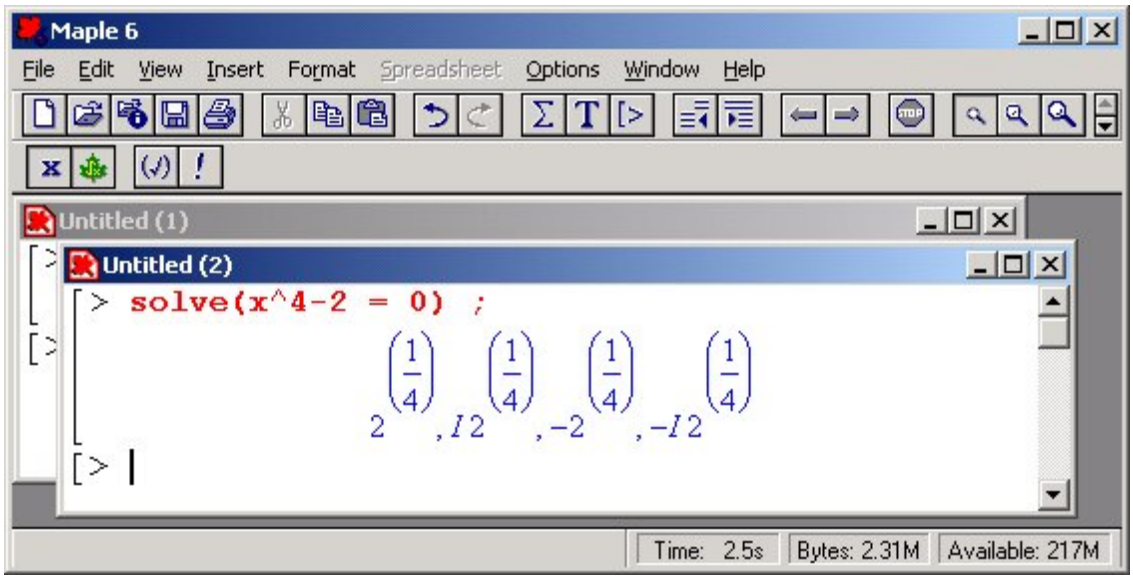


Il peut alors décider d'ouvrir une autre fenêtre avec la touche[7] **Ctrl-N** :

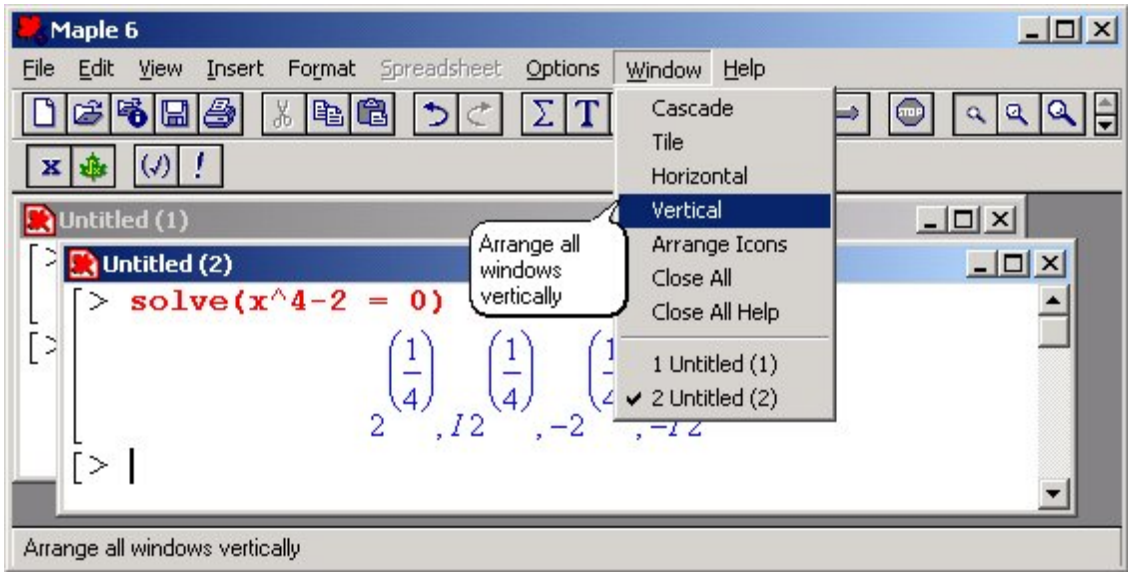


puis dans cette nouvelle fenêtre demander de résoudre l'équation $x^4 - 2 = 0$:

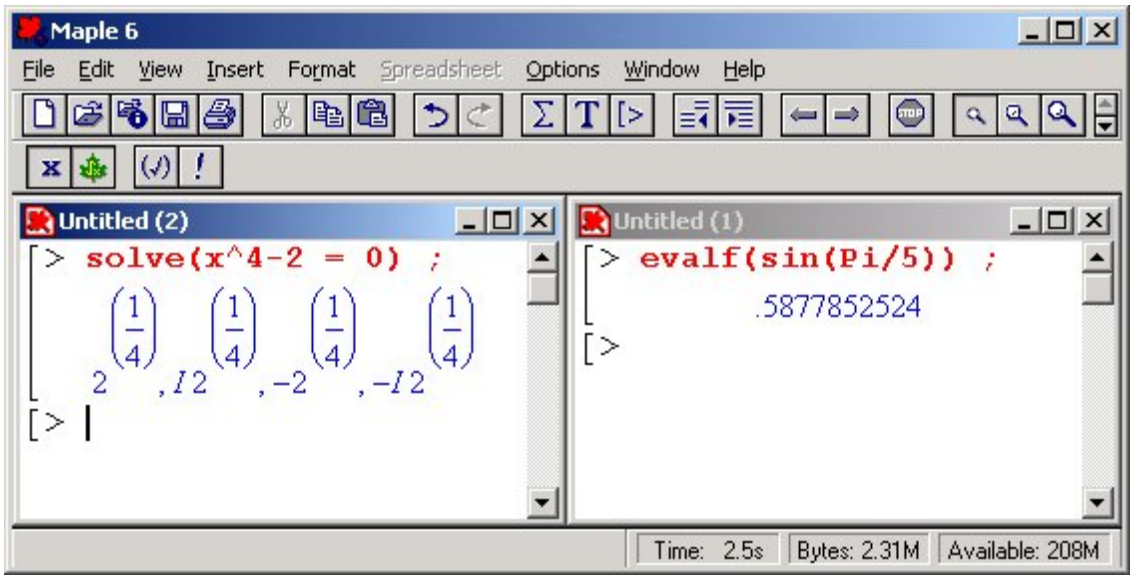
[7] Voir Section 1.1.



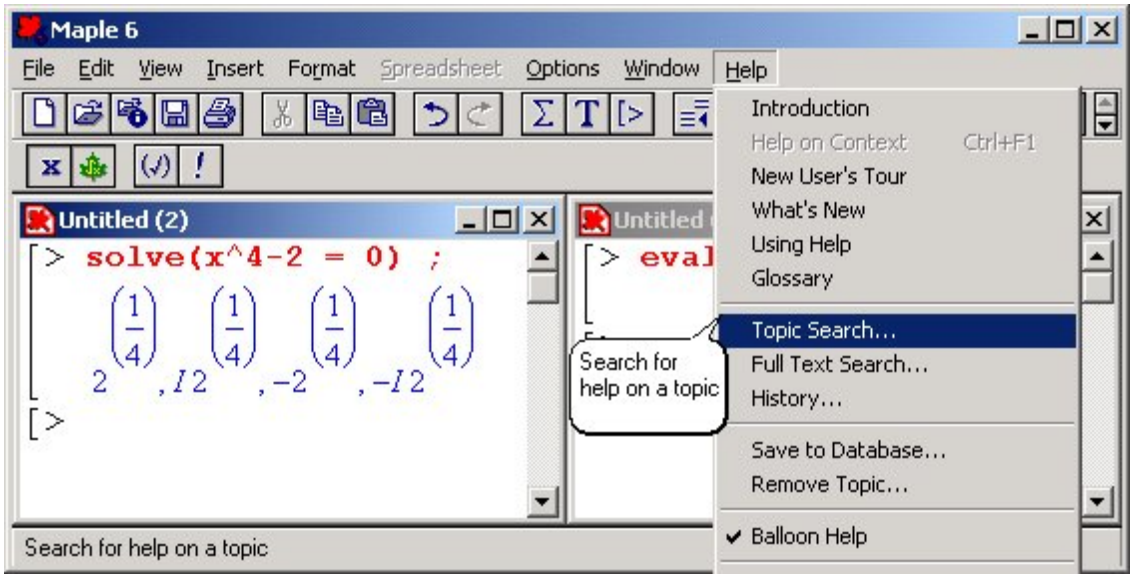
On voit que la deuxième fenêtre est active (foreground) alors que la première est passive (background) ; noter en particulier les couleurs des bandeaux des fenêtres selon leur état. On peut néanmoins préférer voir simultanément les deux fenêtres ; ceci est obtenu en demandant l'option **Vertical** du menu **Window** :



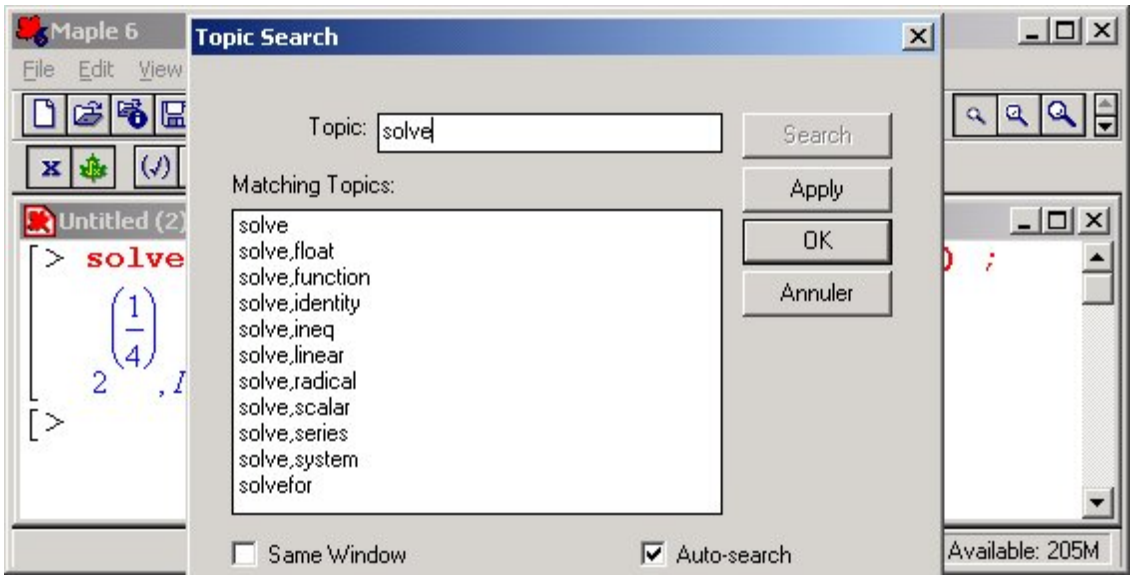
On voit que Maple répartit exactement, verticalement, l'espace disponible entre les fenêtres ouvertes.



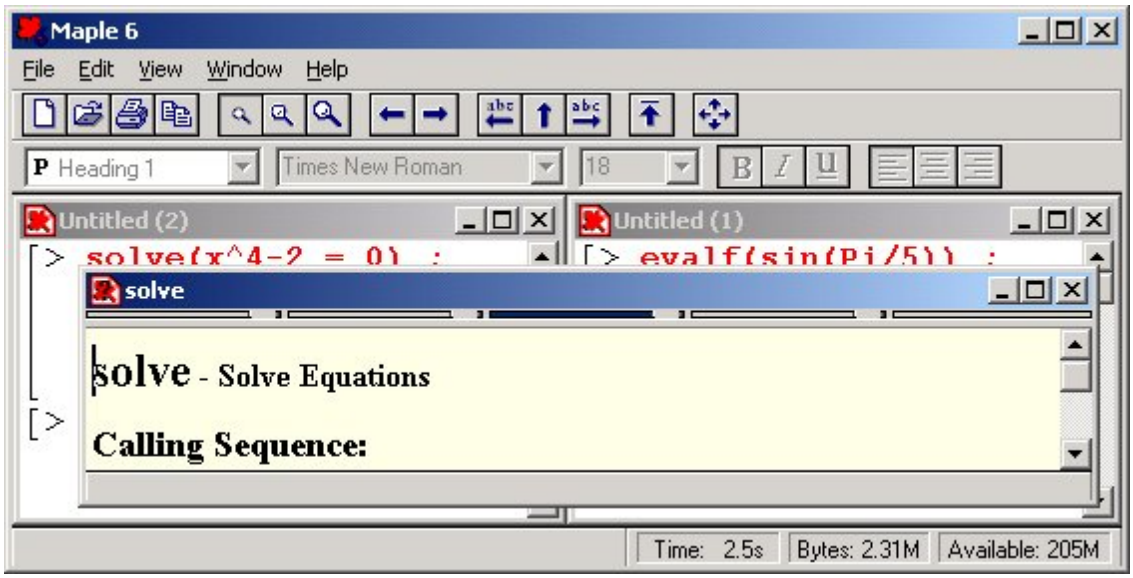
Un système d'aide à l'utilisateur, très riche, très bien conçu, est accessible par le menu **Help**. Par exemple, si on souhaite voir la documentation de base de la procédure Maple prédéfinie `solve`, on demande l'option **Topic Search** du menu **Help** :



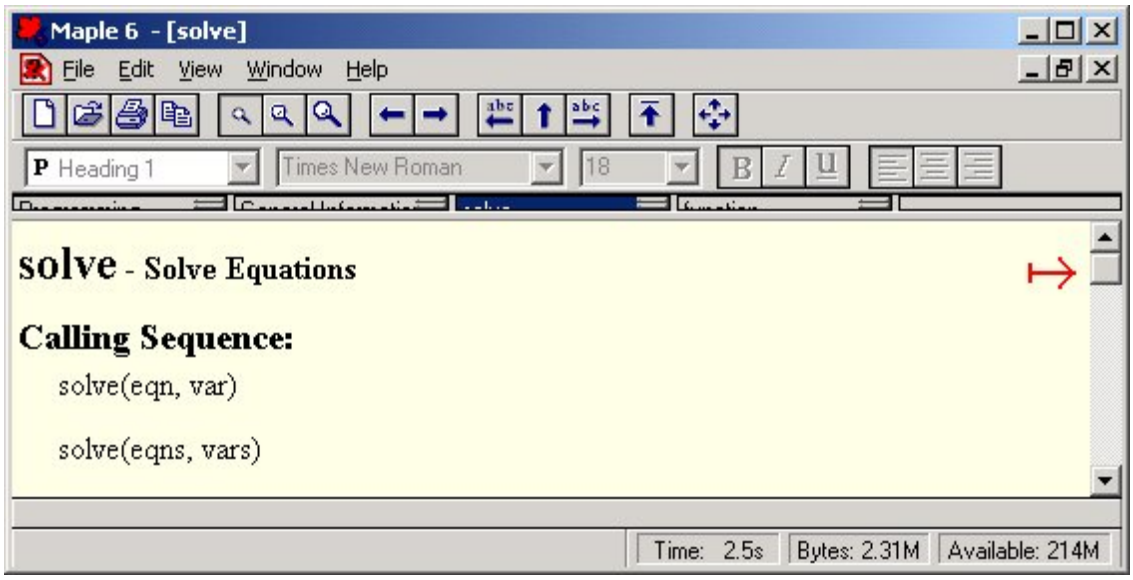
ce qui fait apparaître une fenêtre de dialogue où on peut entrer le nom de la procédure pour laquelle la documentation est souhaitée :



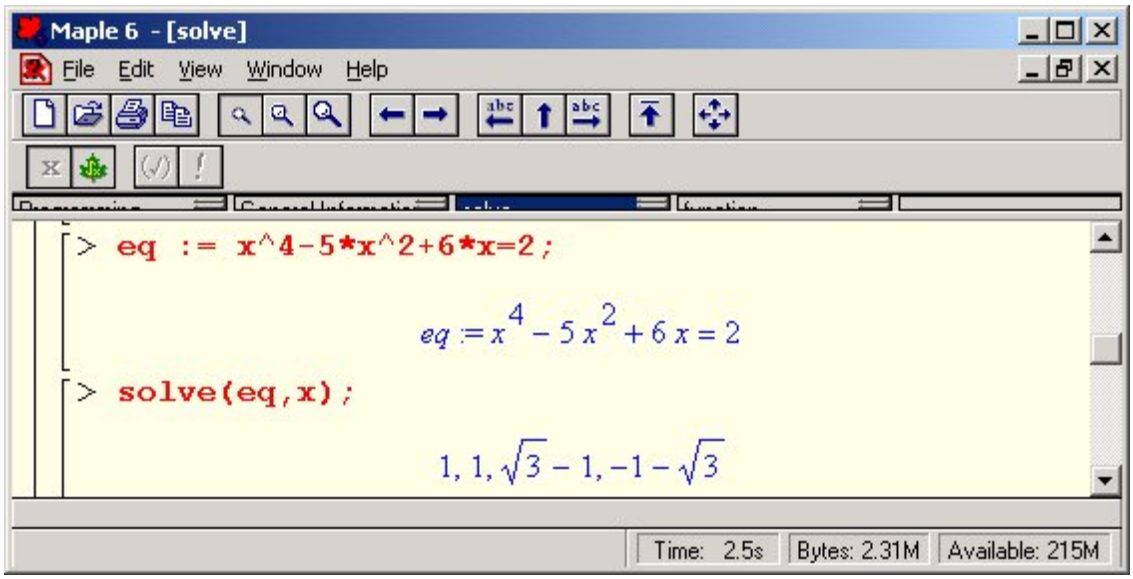
On obtient :



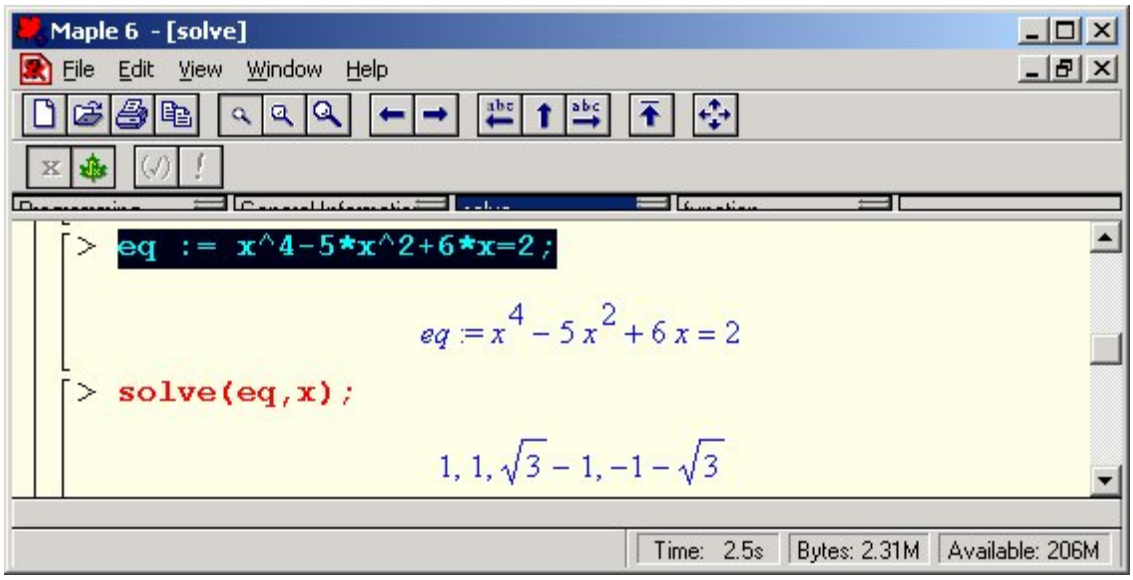
Sur la reproduction donnée ici, la fenêtre aide est minuscule et l'aide est en définitive invisible à moins de faire passer cette fenêtre en mode plein écran, en cliquant sur son bouton □ :



puis en faisant glisser le bouton ascenseur, montré ci-dessus par la flèche rouge, on peut atteindre la partie qui nous intéresse :

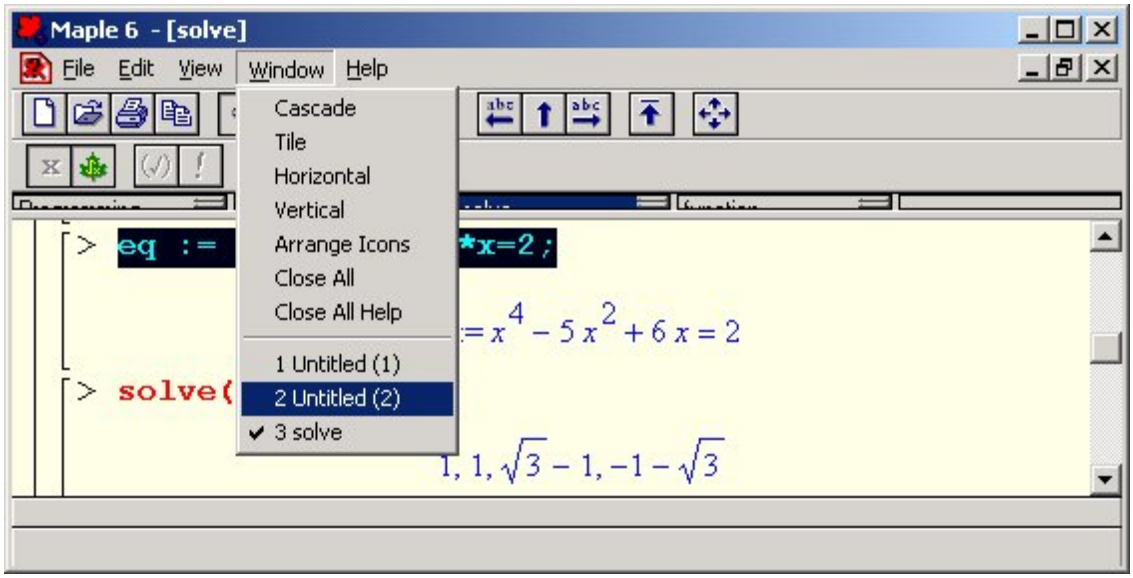


La jolie équation montrée dans la documentation peut nous intéresser, d'où l'idée de l'essayer dans notre feuille de travail ; un balayage de souris permet de *sélectionner* cette équation :

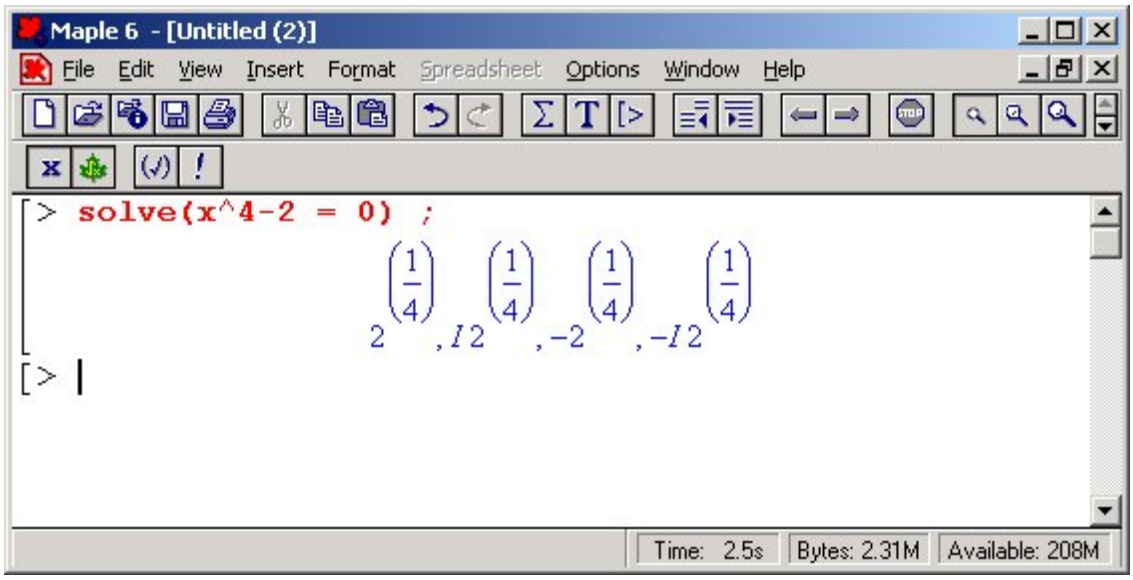


puis la touche **Ctrl-C** ou le bouton  copie le texte sélectionné dans le *presse-papiers* (clipboard), pour utilisation ultérieure.

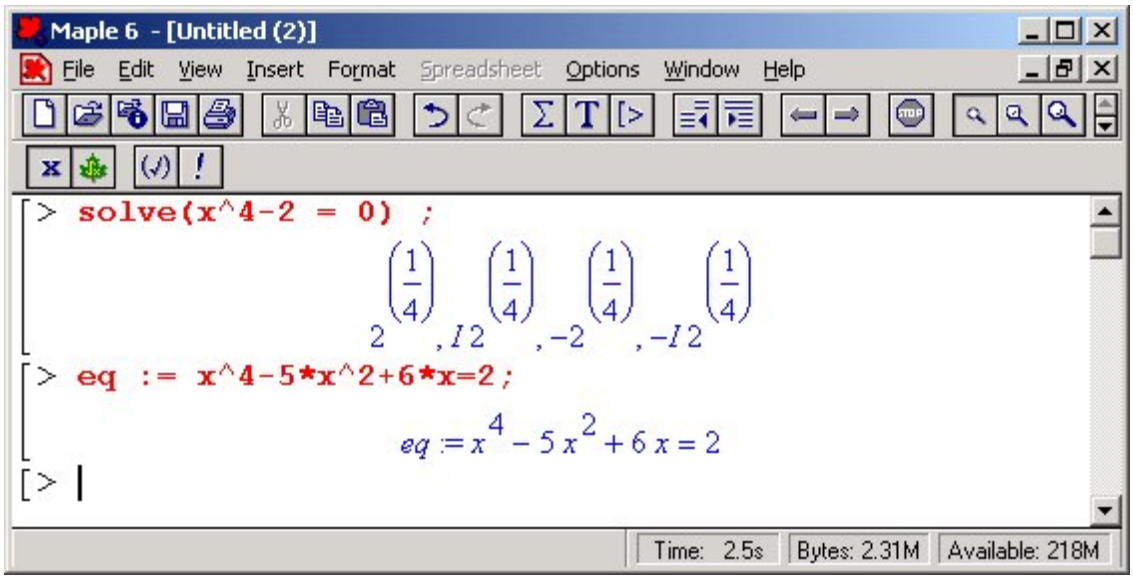
On peut maintenant retourner à la deuxième fenêtre en demandant l'option **2 Untitled(2)** du menu **Window** :



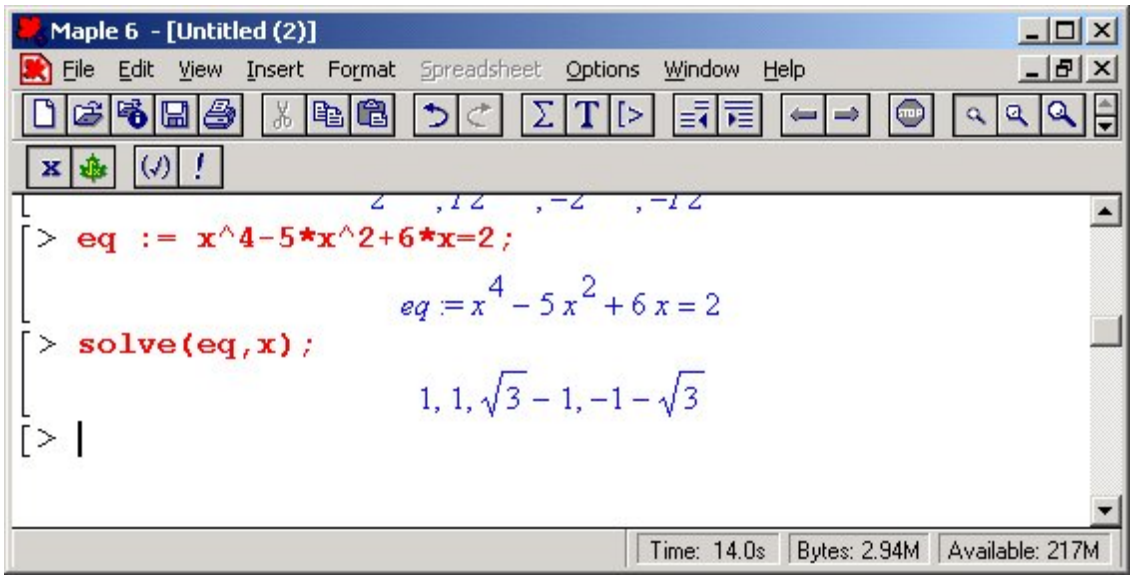
et on retrouve bien notre fenêtre, qui est d'ailleurs passée aussi en mode plein écran comme la fenêtre d'aide :



La touche **Ctrl-V** ou le bouton  permet alors de recopier notre équation là où le curseur était resté positionné dans la feuille de travail ; la touche <**Entrée**> demande l'évaluation et donne le même résultat que celui qui était montré dans la documentation :



On apprendra plus tard qu'on a ainsi *affecté* au symbole `eq` un objet **equation** ; on peut maintenant demander la résolution de cette équation comme il était montré dans la documentation :



Etc. etc. etc.

Il est absolument inutile de chercher dès le début à connaître toutes les possibilités de ce système de fenêtrage ; vous le découvrirez petit à petit au gré de vos expériences et aussi quelquefois en observant d'autres utilisateurs au travail, expérience souvent bien utile.

Habituellement, au cours d'une session, l'utilisateur a une feuille de travail *principale*, celle où il fait les calculs les plus importants, les organisant selon un ordre séquentiel logiquement défini. Souvent l'utilisateur voudra *sauvegarder* sa feuille de travail, par exemple pour la réutiliser pour une démonstration ; plus souvent il s'agira de continuer ultérieurement un travail commencé mais pas terminé ; d'autres fois il s'agira d'améliorer la feuille de travail pour y modifier une méthode de calcul, en améliorer la présentation, etc. On expliquera un peu plus loin comment gérer ces questions.

Nous supposerons toujours dans ce manuel qu'une seule feuille de travail est active, ce qui nous permettra de parler sans ambiguïté de «la» feuille de travail. C'est un espace de travail où l'utilisateur écrit des commandes Maple, demande à Maple de les exécuter, corrige telle commande, la fait réexécuter, ajoute une nouvelle commande, la fait exécuter, fait exécuter l'ensemble de toute la feuille, etc. De temps à autre, la feuille de travail sera cachée derrière une feuille d'aide, demandée par le menu **Help** ou un moyen équivalent.

Pendant la même session l'utilisateur voudra de temps à autre mener de petites expériences annexes sans risquer de trop «abîmer» sa feuille de travail principale. Il ouvrira alors une ou plusieurs feuilles annexes à durées de vie souvent assez courtes qu'il sera rarement nécessaire de sauvegarder. Il peut aussi travailler en parallèle sur plusieurs feuilles de travail «principales»[8].

[8] Dans la configuration standard, les calculs effectués sur une feuille sont connus des autres feuilles, et il ne s'agit donc pas d'un vrai «parallélisme» ; l'option «`-km p`» de la commande système lançant Maple demande au contraire que les différentes feuilles s'ignorent complètement et plusieurs sessions différentes peuvent alors être menées de front.

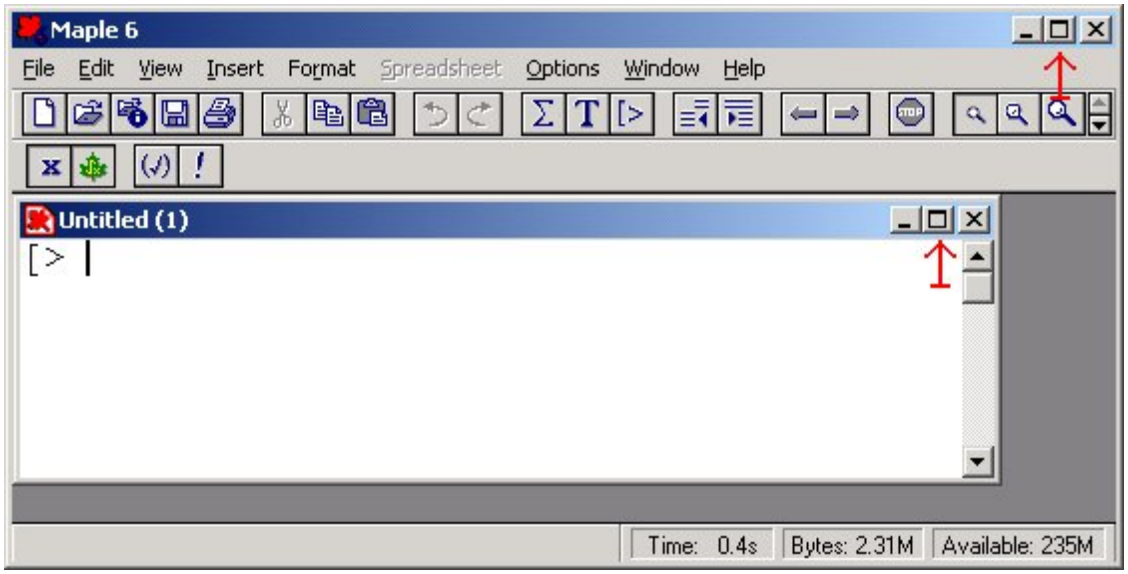
Fréquemment notre utilisateur va devoir consulter l'aide en ligne de Maple. Divers procédés lui permettent d'afficher des feuilles d'aide décrivant l'usage de telle ou telle procédure, comprenant presque toujours des exemples typiques pour en expliquer plus facilement le fonctionnement.

L'utilisateur va devoir gérer ses fenêtres. Acquérir assez rapidement un bon style en la matière est important. Les débutants ont tendance à laisser ouvertes beaucoup trop de feuilles d'aide ; ils sont alors aussi à l'aise pour travailler que sur un bureau ordinaire quand d'innombrables papiers, livres et canettes de bière y sont entassés...

Conseil 3 — *Le plus souvent on peut s'organiser dans une session pour avoir une seule feuille de travail principale, au plus une seule feuille de travail à petite expérience, à supprimer dès que la dite expérience est terminée. Enfin il est préférable de détruire les feuilles d'aide dès que leur consultation est terminée ; le cas échéant on peut les rouvrir très facilement.*

1.4 Deuxième session.

Si Maple est lancé directement[9], il apparaît avec une seule feuille de travail vierge intitulée `[Untitled-(1)]`, titre visible sur le bandeau-titre de la fenêtre. On agrandit au besoin la grande fenêtre Maple en plein écran en cliquant sur le bouton Windows ▣ en haut à droite de la fenêtre. On agrandit aussi la feuille de travail en plein écran par le même procédé, à l'aide de ses *propres* boutons :



Le *curseur*, un trait vertical, indépendant du pointeur souris, clignote au début de la feuille de travail vierge derrière le *prompt Maple*, à savoir les caractères «[>».

[9] Sous Windows, Maple peut aussi être lancé en *ouvrant* un fichier feuille de travail, dont le code extension est normalement «`.mws`» ; le cas échant, la feuille en question est préalablement installée dans la session ; voir cependant ce qui est expliqué Section 1.5.

En fait le crochet ouvrant «[» n'est pas un caractère, c'est une sorte d'accolade pour indiquer la portée d'un *groupe d'exécution* (Execution Group), notion qu'il est inutile de détailler pour le moment ; vous verrez clairement de quoi il s'agit par la suite. Dans les exemples cités dans ce manuel, nous ne montrerons pas, sauf cas très particuliers, ces indicateurs. Le seul prompt Maple est en fait le caractère >.

Comme premier exemple de travail Maple, supposons qu'on souhaite évaluer l'intégrale suivante :

$$\int_1^{\infty} \frac{1+x}{1+x^2+x^4}\,dx,$$

Il faut entrer :

```
> int((1+x)/(1+x^2+x^4), x=1..infinity) ;
```

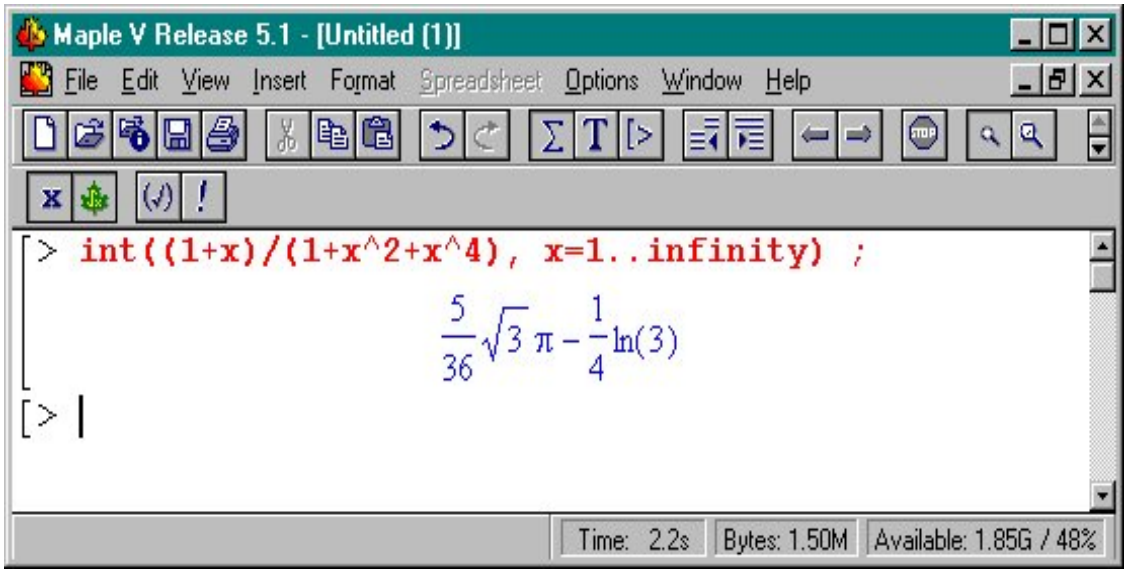
Le texte entré est une expression *Maple-Input*, constituée d'une suite de caractères ASCII, forme externe d'un *objet Maple* dont on s'apprête à demander l'*évaluation*. Pour ce faire, après avoir entré le « ; » terminal, indispensable, on tape la touche <**Entrée**>. Il est important de comprendre à ce point que cette touche <**Entrée**> *n'est en aucune façon le caractère de passage à la ligne suivante* comme dans presque tous les traitements de texte. Conventionnellement, sous Maple, cette touche demande à Maple l'évaluation de tous les objets du groupe d'exécution où le curseur est positionné ; ici ce groupe est réduit à la seule ligne définissant l'intégrale cherchée. Une fois que cette touche <**Entrée**> est... entrée, Maple évalue l'intégrale demandée et affiche le résultat comme suit :

```
> int((1+x)/(1+x^2+x^4), x=1..infinity) ;
```

$$\frac{5}{36}\sqrt{3}\pi - \frac{1}{4}\ln(3)$$

```
> |
```

Les manuels «grand public» préfèrent souvent montrer une copie «bitmap» de la fenêtre obtenue :



On voit que c'est beaucoup plus encombrant et souvent finalement bien moins précis que ce qui est indiqué par une typographie standard simulant avec une bonne approximation ce qui est observé dans la fenêtre Maple. On n'utilisera désormais une copie *exacte* d'une fenêtre Maple que dans des circonstances relativement exceptionnelles.

On a donc obtenu le résultat :

$$\int_1^\infty \frac{1+x}{1+x^2+x^4}\,dx \;=\; \frac{5}{36}\sqrt{3}\pi - \frac{1}{4}\ln 3,$$

déjà pas si facile à atteindre avec papier-crayon.

Très souvent, comme la syntaxe requise pour entrer les expressions à évaluer est relativement complexe et ne pardonne aucune imprécision, l'utilisateur Maple fait des fautes de natures assez variées, demandant un minimum de lucidité, quelquefois beaucoup, pour être identifiées puis corrigées. Supposons par exemple que nous oublions dans l'entrée de l'expression intégrale la première parenthèse fermante, celle qui ferme le numérateur `(1+x_` : Maple affiche alors un message d'erreur en petits caractères mauves, sur la ligne inférieure :

```
> int((1+x/(1+x^2+x^4), x=1..infinity) |;
';' unexpected
```

Le curseur clignote devant le point-virgule terminant la ligne entrée, pour indiquer le point où une anomalie a été détectée : il est incohérent de trouver un « ; » terminant une instruction alors que la première parenthèse ouvrante n'a pas encore de parenthèse fermante associée. Comme c'est souvent le cas, Maple ne pouvait pas deviner que la parenthèse manquait après `(1+x_`. Continuons à faire l'âne et insérons une nouvelle parenthèse juste avant le point-virgule terminal. *Dès que* cette parenthèse est insérée, on peut *immédiatement* taper <**Entrée**> pour demander la réexécution de la commande, quel que soit le point où le curseur y est positionné ; il est en particulier inutile de repositionner le curseur après le point-virgule.

```
> int((1+x/(1+x^2+x^4), x=1..infinity)) ;
                                        ∞
> |
```

Pas d'erreur détectée, mais le résultat n'est pas exactement celui qui est espéré ! Vu l'apparente correction du résultat (!), Maple a ouvert un nouveau groupe d'exécution où il a positionné le curseur en attente de l'expression suivante à évaluer.

Notre intégrale semblait convergente, et pourtant Maple donne le résultat ∞, signifiant au contraire qu'il l'a trouvée divergente. Le mécanisme d'appel de procédure de Maple est très puissant, mais de temps à autre il implique que ce qui est en fait une erreur d'entrée «réussisse» à être interprété d'une façon assez surprenante pour l'utilisateur non initié. Il se trouve qu'ici l'expression à évaluer, malgré l'*apparente* faute de parenthésage, est tout-à-fait correcte pour Maple, sauf qu'il a évalué en fait :

$$\int_1^\infty \left(1 + \frac{x}{1+x^2+x^4}\right) dx,$$

qui est bien une intégrale divergente. Il n'est pas vraiment évident de comprendre comment Maple a pu aboutir à cette interprétation ; nous verrons ce point en temps utile, voir page 115. On entre ici en *debugging* proprement dit. Il faut élucider comment Maple a pu obtenir ce résultat bien troublant. L'examen soigneux de l'expression entrée finit par faire comprendre que le numérateur `1+x` de l'intégrande n'est pas correctement parenthésé. On positionne le curseur derrière le `x` critique par un clic de souris ou à l'aide des touches directionnelles du clavier («↑»,«↓»,«←» et «→») et on ajoute la parenthèse manquante ; on tape <**Entrée**> et on obtient ce résultat :

```
> int((1+x)/(1+x^2+x^4), x=1..infinity)) ;
')' unexpected
```

On a cette fois une parenthèse en trop ! On l'efface et on obtient enfin un résultat correct :

```
> int((1+x)/(1+x^2+x^4), x=1..infinity) ;
```

$$\frac{5}{36}\sqrt{3}\pi - \frac{1}{4}\ln(3)$$

```
> |
```

Chaque fois qu'un résultat est (ou, pour Maple, paraît) correct, Maple ouvre après l'affichage du résultat un groupe d'exécution où il positionne le curseur. Si un tel groupe était déjà présent, Maple ne crée pas un nouveau groupe, mais il positionne le curseur au début de la dernière ligne du groupe d'exécution suivant celui qui vient d'être exécuté. On ne montrera plus désormais ce «groupe suivant» dans nos exemples.

Notre calcul d'intégrale est cette fois réussi, au moins pour un mathématicien. Le résultat *exact* obtenu, nécessitant pour être exprimé le radical, le nombre π et le logarithme, n'est pas forcément celui qui est souhaité par exemple par le physicien, quand il cherche un résultat *numérique*. Notre physicien pourra alors pour atteindre le but recherché exécuter :

```
> evalf(%) ;
```

.4810966632

Le symbole `%` représente le dernier résultat obtenu, et la procédure `evalf` demande la conversion en un flottant de l'objet désigné. Mais supposons qu'on réalise alors qu'on voulait que la borne inférieure de l'intégrale soit 0 et non pas 1. Dans ce cas on peut remonter le curseur pour modifier la fin de la commande d'intégration, y remplaçant «`..., x=1..inf...`» par «`..., x=0..inf...`». On réexécute la commande en tapant <**Entrée**>, le nouveau résultat s'affiche et le curseur est positionné au début du `evalf(%)` ; il suffit alors de retaper <**Entrée**> pour obtenir la nouvelle valeur numérique de l'intégrale. On obtient ceci :

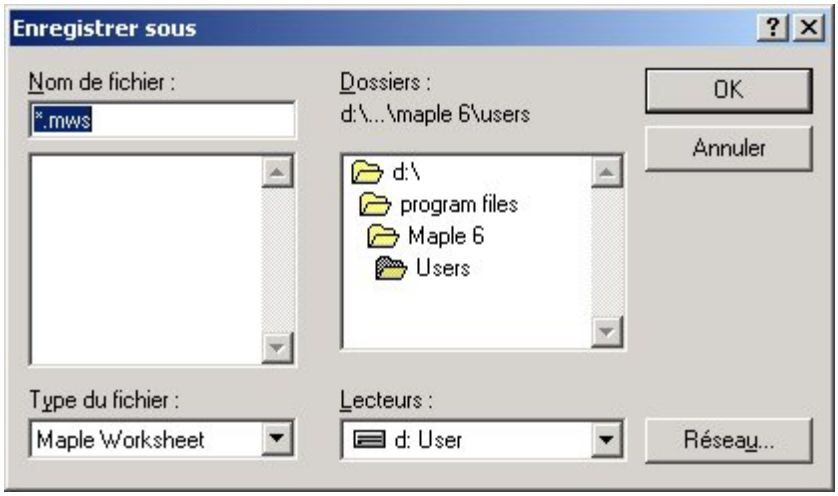
```
> int((1+x)/(1+x^2+x^4), x=0..infinity) ;
```

$$\frac{5}{18}\sqrt{3}\pi$$

```
> evalf(%) ;
```

$$1.511499471$$

C'était le but de cette session qu'il faut donc maintenant terminer, mais il est peut-être préférable de la *sauvegarder* en vue d'une réutilisation ultérieure. Pour ce faire on active l'option **Save** du menu **File**, ou encore on utilise le raccourci clacier **Ctrl-S** ; une fenêtre de dialogue intitulée «Enregistrer sous» s'ouvre, permettant de préciser le fichier de sauvegarde souhaité. Il faut prendre son temps pour en comprendre les possibilités.

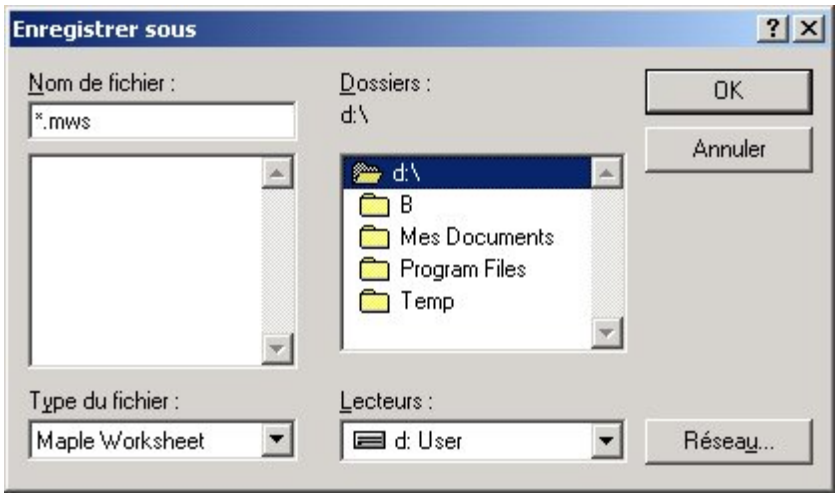


Cinq mini-fenêtres sont incluses. La première, une seule ligne où le curseur est positionné, indique `*.mws`, ce texte apparaissant *sélectionné*, c'est-à-dire surchargé par une bande colorée, bleue sous le standard Windows. La fenêtre juste inférieure rappelle pour mémoire les fichiers déjà présents dans le *dossier en cours*. A droite de cette fenêtre se trouve justement une autre fenêtre qui indique la situation actuelle par rapport à l'ensemble des *dossiers* de l'unité de disque active. En haut, les dossiers «actifs» sont *ouverts*. Ici le répertoire en cours est donc :

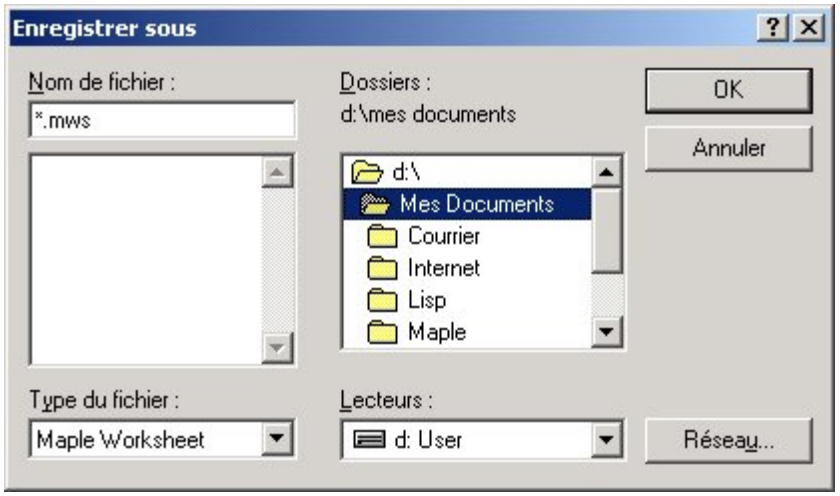
```
D:\Program Files\Maple 6\Users\
```

C'est ce que vous trouverez[10] la première fois après installation de Maple. Il est donc important de savoir comment gérer cette situation. Il ne faut en aucune façon sauvegarder son fichier dans ce dossier à considérer comme *système*. La tradition sous Windows consiste, en utilisant par exemple Windows-Explorer, à créer dans le dossier `D:\Mes Documents\` un dossier `Maple`. Ceci fait, indépendamment de Maple, il faut maintenant, dans Maple, «remonter» jusqu'à `C:\` en cliquant rapidement deus fois dans la fenêtre de dialogue Maple sur cette indication ; on obtient quelque chose comme :

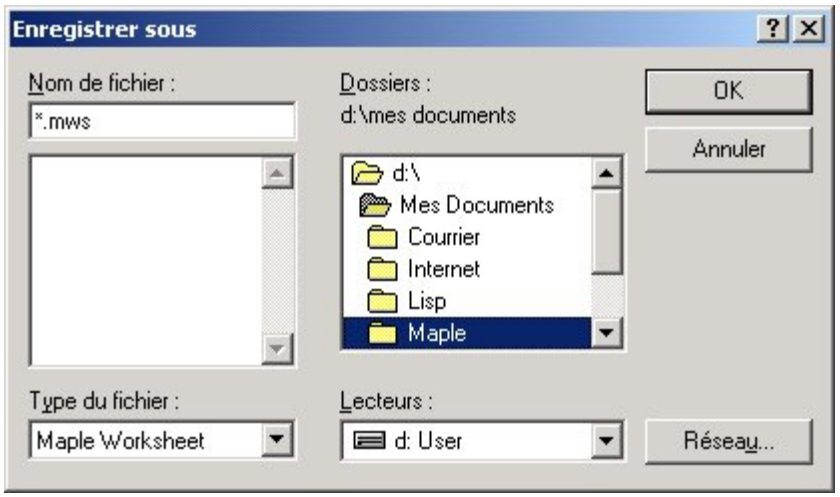
[10] L'installation Windows du rédacteur prévoit le dossier système Maple et son utilisation sur le *disque* «`D:\`» ; les installations les plus simples utilisent en général le disque «`C:\`».



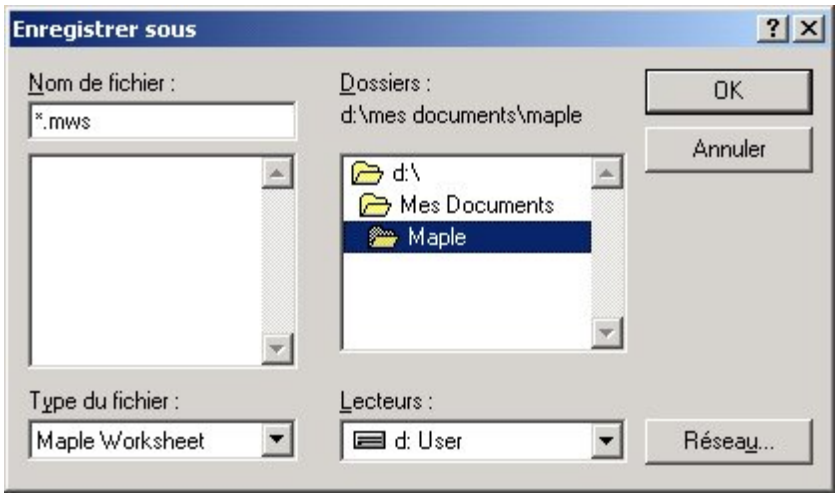
Puis il faut «descendre» pour atteindre notre dossier, donc cliquer d'abord (deux fois) sur `Mes Documents` :



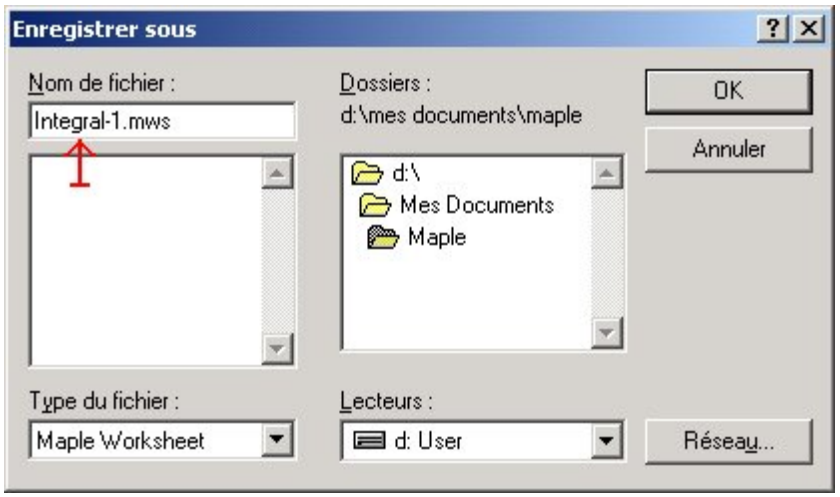
Le dossier `Maple` installé, par exemple à l'aide de Windows Explorer, sous le dossier `Mes Documents`, peut être sélectionné par un clic de souris, ou par utilisation de la touche directionnelle '↓', ou encore en entrant (deux fois) '`m`' au clavier :



Tapant <**Entrée**> ou cliquant deux fois sur l'indication `Maple`, on *ouvre* ce dossier, vide puisqu'il vient d'être créé, comme indiqué par la fenêtre de gauche :



On va alors avec la souris dans la micro-fenêtre en haut à gauche, on efface[11] l'étoile «∗» et on le remplace par exemple par `Integral-1` :



Enfin on clique sur `OK`, ou on tape simplement <**Entrée**>. La fenêtre de dialogue disparaît, mais notre belle feuille de travail est enregistrée comme le fichier :

```
D:\Mes Documents\Maple\Integral-1.mws
```

On peut changer le disque actif à l'aide de la mini-fenêtre intitulée «Lecteurs». Le système d'évolution dans l'espace disque est très commode, mais demande un peu de temps pour être bien compris. Ne négligez pas cette question : il est fréquent de voir les débutants persuadés d'avoir «perdu» une feuille de travail, alors qu'en fait elle a été sauvegardée dans un dossier bien différent de celui qui était prévu...

[11] Effacement très souvent oublié par les débutants, auquel cas Maple *refuse* l'enregistrement, sans autre explication.

Warning 4 — *Dans la fenêtre «Enregistrer sous», bien noter que le dossier* actif *est le dernier dossier* visuellement ouvert *de la mini-fenêtre de droite.*

Ne changez rien à la mini-fenêtre intitulée «Enregistrer au format :» ; autrement dit considérez comme une norme sans exception que votre feuille de travail est sauvegardée sous un nom de fichier d'*extension* `.mws`, mnémonique de *Maple-WorkSheet*.

Notez que quand votre feuille est sauvegardée, le nom choisi apparaît dans le bandeau titre de votre fenêtre Maple. D'une façon générale est ainsi désignée la feuille de travail actuellement active, `[Untitled(x)]` si c'est une feuille non sauvegardée, le nombre `x` évoluant au gré des créations de nouvelles feuilles de travail. La liste des fenêtres actuellement existantes est disponible dans le menu **Windows**, la fenêtre active étant marquée ✓.

Vous pouvez alors, maintenant que votre feuille est sauvegardée, continuer votre travail, entrer de nouvelles commandes, etc. Vous pouvez aussi fermer votre session Maple, par exemple en cliquant sur le bouton Windows ☒ en haut à droite de la fenêtre Maple.

Si vous tentez de terminer une session Maple et qu'une ou plusieurs fenêtres ne sont pas sauvegardées, Maple vous demande de confirmer votre décision pour chaque fenêtre successivement. Prenez soin de réfléchir quelques secondes pour éviter de perdre un travail peut-être précieux.

1.5 Réactiver une feuille de travail sauvegardée.

Il y a plusieurs façons de procéder. Le plus simple consiste à utiliser l'Explorer Windows et à *ouvrir* (double-clic) le fichier correspondant. Une session Maple s'ouvre où apparaît votre feuille de travail, exactement dans l'état où vous l'aviez laissée, y compris la position du curseur. Autre méthode vous ouvrez comme précédemment une session Maple, et vous utilisez le raccourci-clavier **Ctrl-O** ; une fenêtre de dialogue s'ouvre, très analogue à celle qui servait à la sauvegarde. Evoluant dans l'espace disque comme expliqué plus haut, vous finissez par faire apparaître le fichier de sauvegarde dans la fenêtre gauche de la fenêtre de dialogue où figurent les fichiers du dossier en cours. Par double-clic sur le fichier, la feuille de travail s'ouvre.

Il a été dit ci-dessus que vous retrouvez votre feuille de travail *exactement* dans l'état où vous l'avez laissée ; *visuellement* oui, en fait pas du tout : les calculs qui avaient été effectués dans la session précédente sont «perdus», à moins d'en demander explicitement la réexécution. Pour comprendre ce dont il s'agit, préparez une feuille de travail avec cette mini-session :

```
> x := 34 ;
```
$$x := 34$$
```
> x + 3 ;
```
$$37$$
```
> |
```

Vous laissez le curseur là où il est positionné et vous sauvegardez cette feuille de travail ; vous terminez la session Maple et allez prendre un café. Ceci fait, vous revenez à votre poste de travail, relancez une session Maple et ouvrez la feuille qui avait été précédemment sauvegardée. Le curseur est positionné toujours au dernier groupe d'exécution, pour le moment vide. Vous entrez au clavier «`x + 3 ;`<**Entrée**>». Votre feuille de travail devient :

```
> x := 34 ;
```
$$x := 34$$
```
> x + 3 ;
```
$$37$$
```
> x + 3 ;
```
$$x + 3$$

Dans la première session, on avait affecté au symbole `x` l'entier `34`, puis on avait évalué l'objet `x+3` ; on détaillera bientôt le fonctionnement de ces choses, mais il est naturel de voir là `37` comme résultat de l'évaluation, compte tenu de la valeur de `x`. La session est alors arrêtée après sauvegarde.

Ensuite on lance une *autre* session Maple ; on y ouvre la feuille de travail sauvegardée à la fin de la session précédente. On retrouve le curseur positionné là où il était au moment de la sauvegarde, mais ceci ne signifie *en aucune façon* que les instructions précédentes ont été préalablement exécutées. En particulier, pour le symbole `x`, tout se passe comme si la valeur de la session précédente, `34`, était en fait *perdue*, et c'est pourquoi l'évaluation de `x+3` retourne seulement `x+3` : un symbole «sans valeur» a, sous Maple, lui-même comme valeur, en fait valeur *par défaut* ; ce mécanisme est très important en calcul symbolique et sera détaillé plus loin. C'est pourquoi on a maintenant à l'écran deux évaluations apparemment successives de `x+3` avec des résultats différents.

Mais vous pouvez si vous le souhaitez faire réexécuter toute votre feuille. Activez l'option **Execute Worksheet** du menu **Edit**. Toute la feuille va être réexécutée et cette fois les deux évaluations de `x+3` donnent le résultat `37`, à cause de l'affectation préalable de `34` à `x`. Autrement dit la feuille est inchangée sauf que le dernier résultat est maintenant `37`. Un corollaire est le conseil qui suit.

Conseil 5 — *Organisez vos feuilles de travail «à conserver» de telle façon qu'elles soient* entièrement *réexécutables sans incident. Sinon elles peuvent rapidement devenir difficilement utilisables. Pour que ce travail de mise au propre ne soit pas trop pénible, faites la mise à jour nécessaire de temps à autre pendant votre session.*

L'option **Execute Worksheet** du menu **Edit** vous permet de vérifier rapide-

ment l'état de votre feuille de travail à ce propos. Toutes les commandes Maple de votre feuille sont alors exécutées séquentiellement. Même si une commande termine sur erreur, Maple continue et lance aussitôt l'exécution de la commande suivante, jusqu'à la fin de la feuille. Le défilement est tellement rapide qu'une telle erreur peut passer visuellement inaperçue, et votre feuille n'est peut-être pas alors dans l'état que vous souhaitez. Mais vous pouvez lancer la recherche arrière[12] de toutes les occurences de «Error» dans votre feuille pour conclure à ce propos.

Maple 6 est la première version de ce logiciel possédant un mécanisme de *sauvegarde automatique*, voir l'option **Autosave** du menu **Options** et la page de documentation «`worksheet,options,autosave`». Il arrive en effet de temps à autre qu'un calcul trop complexe, ou encore demandé de façon erronée, provoque un crash complet de la session Maple ; dans un tel cas, tous les calculs précédemment préparés peuvent être *perdus*. L'option **Autosave** permet de demander à Maple la sauvegarde systématique, à intervalles réguliers, de votre feuille de travail, sous un nom de fichier particulier construit à partir du nom du fichier associé en ajoutant `_MAS` (Maple AutoSave). Les risques de «grosses pertes» sont ainsi très sensiblement diminués. Pour tout travail d'envergure, il *faut* absolument utiliser cette possibilité.

1.6 Le système d'aide.

Le système d'aide en ligne Maple est remarquablement bien fait et contribue certainement à son succès. Le système d'aide est lui-même autodocumenté, ce qui dispense le rédacteur de ce manuel de s'étendre exagérément sur le sujet. Si vous êtes parfait débutant Maple, vous devez d'abord ouvrir le menu **Help** et choisir l'option **Using Help**. Vous verrez que les rubriques d'aide sont essentiellement :

1. *New User's Tour* : Découpé à son tour en plusieurs «Tours» spécialisés : calculs numériques, calculs algébriques, graphiques, calculus, etc. Ces pages d'aide contiennent de très nombreux exemples de petits calculs Maple, calculs au sens large, faciles à répéter. Le caractère publicitaire n'est pas absent ; la plupart des exemples sont assez impressionnants par leur efficacité apparente. En général l'utilisateur est un peu abusé : il est souvent victime à ce stade d'une fâcheuse tendance à croire que tout et n'importe quoi est facilement faisable sous Maple ; pourtant dès qu'il cherche à appliquer ce qu'il a observé dans le New User's Tour pour le calcul apparemment voisin qui l'intéresse, il doit déchanter... La technique à acquérir est en fait sensiblement plus difficile que ce qui est suggéré dans le New User's Tour !
2. *Topic Search* : C'est d'une certaine façon le «symétrique» du New User's Tour ; c'est en effet le *Reference Guide* de Maple, très complet, donc nécessairement assez indigeste pour une lecture linéaire. Chaque procédure Maple prédéfinie y est bien documentée. La nature des arguments selon les différents cas de figure est détaillée. De nombreux exemples pour chaque type d'utilisation possible sont proposés, eux aussi facile à répéter.

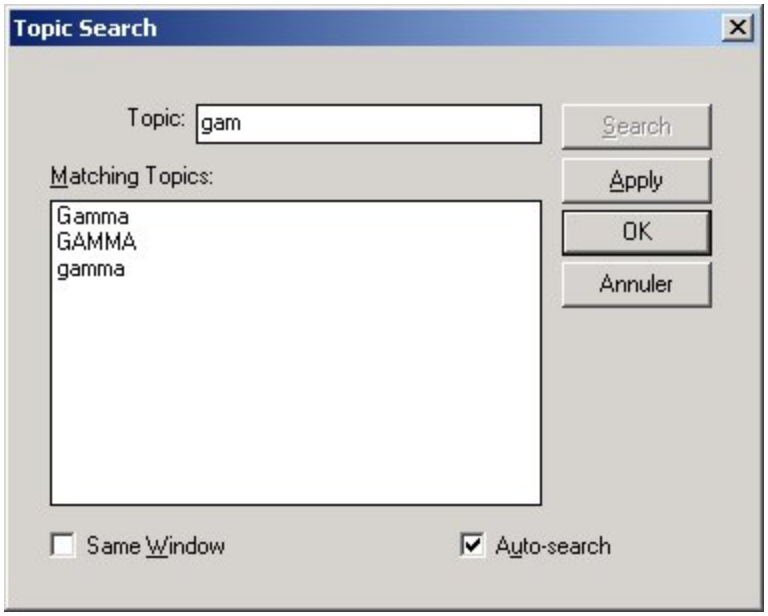
[12] Raccourci **Ctrl-F**, bouton **Previous**.

3. *Full Text Search* : Très utile dans les cas difficiles ; vous proposez un mot et Maple cherche toutes les occurences de ce mot dans les feuilles d'aide ; elles sont de plus classées par la fréquence d'apparition. Voir ci-dessous un usage fréquent.

Malgré le côté forcément publicitaire du New User's Tour, il ne doit pas être négligé. Après plusieurs années d'utilisation de Maple, on a la surprise d'y trouver encore de temps à autre des «tuyaux» utiles !

1.6.1 Le «Reference Guide».

La principale source d'information est bien entendu le *Reference Guide*, accessible par l'option **Topic Search** du menu **Help**. La combinaison **(Alt-H)+T**[13] est plus rapide et devient l'habitude. Une fenêtre de dialogue s'ouvre demandant quelle est la page de documentation souhaitée. En première approximation, chaque page correspond à une procédure Maple prédéfinie. Par exemple pour obtenir la documentation de la procédure `GAMMA`, chercher la page `GAMMA`. Quand on entre le début du nom de la procédure, Maple affiche immédiatement la liste des pages qui pourraient correspondre ; on découvre ainsi souvent la bonne orthographe à utiliser pour des noms à la devine. Sous Maple, minuscules et majuscules sont distinguées, et l'utilisateur est souvent hésitant ; mais l'outil de recherche **Topic Search** est au contraire *tolérant* à ce propos. Par exemple il n'est pas évident que pour la fonction eulérienne $\Gamma(t)$ c'est la procédure `GAMMA`, tout en majuscules, qu'il faut utiliser, mais on le découvre ainsi en entrant approximativement «`gam`» :



Au passage, les pages proposées peuvent donner des idées pour des sujets plus ou moins connexes ; ainsi quand on cherche la documentation pour la fonction Γ,

[13] Il faut comprendre d'abord la combinaison *simultanée* **Alt-H**, suivie de la touche **T** seule, la touche **Alt-** étant alors relâchée.

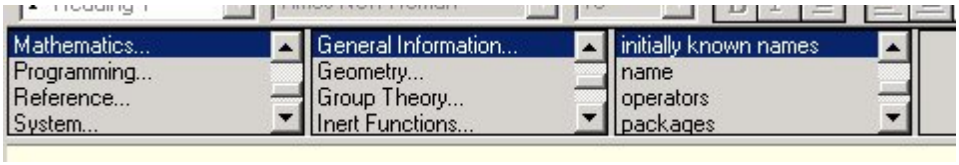
on découvre en même temps la page appropriée pour la constante d'Euler γ, c'est cette fois `gamma` qu'il faut utiliser.

Certaines pages de documentation portent sur des *sujets* et non sur des procédures particulières. Par exemple la demande pour `gamma` envoie en fait à la page «`initially-known names`».

Si vous cliquez sur le bouton **Apply** de la fenêtre **Topic Search**, la page sélectionnée est affichée, mais la fenêtre de dialogue *reste ouverte*, vous permettant de continuer à explorer les autres possibilités.

Il arrive souvent qu'on soit *certain* de la procédure qu'il faut utiliser dans un cas particulier, par contre la façon de l'utiliser n'est pas claire, notamment sur ce qu'il faut prévoir pour ses arguments. Dans ce cas, on peut entrer sur sa feuille de travail le nom de la procédure en question, mettons `randmatrix` ; la touche fonction **F1** ouvre alors directement la feuille d'aide appropriée. Dans le même registe, la méthode antique consistant à entrer la pseudo-commande «`?randmatrix`» fonctionne toujours, mais n'a plus aucune raison d'être ; ne laissez pas ces pseudo-commandes dans votre feuille de travail : elles gêneraient considérablement la réexécution entière de la feuille.

Inversement, si au contraire dans un cas particulier on a un doute sur la pertinence de la feuille d'aide examinée, un autre élément d'information peut être utilisé. Le Reference Guide est entièrement structuré comme un arbre[14] à cinq niveaux hiérarchiques. Quand une feuille est affichée, le chemin joignant votre feuille à la racine de l'arbre est décrit dans un bandeau intercalé entre votre feuille et la barre d'outils :



Les branches voisines sont aussi affichées, de sorte que l'exploration de cet arbre permet quelquefois d'atteindre *logiquement* la feuille qui en fait était appropriée. La manœuvre usuelle de souris permet d'augmenter ou au contraire de diminuer la hauteur de ce bandeau, et donc de s'adapter à la situation du moment. Par example, si vous cherchez à effectuer une simplification visuellement possible dans une expression résultat compliquée, vous allez probablement tenter `simplify`, souvent sans succès. À tout hasard vous explorez la page `simplify` sans rien y trouver de nouveau par rapport à ce que vous pensez connaître. En bout de course, pensez alors à explorer l'arborescence du système d'aide. Pour ce faire examinez le bandeau où vous voyez que le chemin jusqu'à `simplify` est :

[14] Un examen soigneux montre que ce n'est pas strictement parlant un arbre : il peut exister plusieurs chemins de la racine vers une feuille, mais ce moteur de recherche n'en est que plus intéressant ;

```
Mathematics
   Algebra
      Expression Manipulation
         Simplifying
            Simplify
```

Explorant l'arbre, vous voyez qu'il y a beaucoup d'autres façons de simplifier que d'utiliser la procédure `simplify` ; par exemple les procédures `normal`, `radnormal`, `combine`, qui souvent correspondent à ce qui en fait est recherché. En désespoir de cause, on peut même se demander si c'est vraiment une simplification qui est recherchée. Au même niveau de la hiérarchie que `Simplify`*`ing`*, on trouve par exemple `Factoring` qui peut mener sur d'autres pistes. Cet outil de recherche ne doit jamais être négligé.

Une autre méthode peut encore être exploitée quand une feuille d'aide se révèle inappropriée pour un problème particulier : la fin de cette feuille affiche fréquemment un certain nombre de liens hypertextes vers des sujets voisins ; c'est souvent de cette façon qu'on atteint enfin la page cherchée !

1.6.2 Le cas des packages.

Un *package* Maple est un ensemble cohérent de procédures sur un sujet plus ou moins particulier, dont l'usage ne s'impose pas forcément d'emblée à l'utilisateur ordinaire. Par exemple le package `linalg` (algèbre linéaire) contient les procédures évoluées de manipulations matricielles. Le logiciel propose ses propres packages et l'utilisateur peut en créer d'autres ou encore étendre les packages existants.

Un package n'est *chargé* qu'à la demande de l'utilisateur, par la procédure `with`, sans quoi les procédures de ce package sont inaccessibles[15]. Il est possible aussi de charger seulement certaines procédures de tel ou tel package.

L'accès à la documentation des procédures proposées par les divers packages pose un problème qu'il n'est pas difficile de surmonter, mais qu'il faut connaître. Si vous savez d'avance que la procédure qui vous intéresse est dans tel package, il faut entrer *d'abord* le nom du package avant celui de la procédure. Par exemple si vous cherchez la page expliquant la procédure `exponential` du package `linalg`, demandez sous **Topic Search** la page «`linalg,exponential`». En particulier, dès que vous avez entré `linalg`, toutes les procédures du package `linalg` sont affichées dans la fenêtre des possibilités, information souvent utile. Une page introductive est aussi prévue pour chaque package, dont le nom est celui du package.

Si vous êtes certain du nom de votre procédure, vous pouvez aussi directement entrer le nom de la procédure ; le système d'aide saura trouver votre page, mais il ne vous aidera pas par une indication positive dans la fenêtre des possibilités. Essayez pour `eigenvalues`. Il arrive que des procédures *différentes*, de même nom, figurent dans des packages différents. Par exemple il existe deux procédures `exponential`, l'une dans `linalg` calcule l'exponentielle d'une matrice, l'autre dans le sous-package

[15] Plus précisément elles sont accessibles par un mécanisme assez lourd qui doit aussi citer le nom du package.

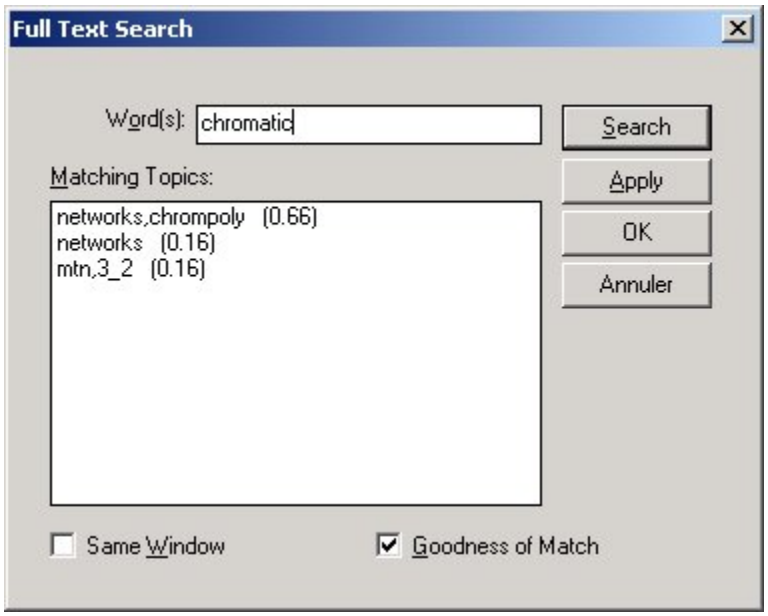
`random` du package `stats` est un générateur de nombres aléatoires correspondant à une loi exponentielle. Si vous demandez sous **Topic Search** la page `exponential`, Maple vous ouvre une page auxiliaire vous proposant le choix par liens hypertextes entre les pages «`linalg,exponential`» et «`stats,random`» ; en cliquant sur l'un de ces liens, vous obtenez la page souhaitée.

1.6.3 La recherche par mot (Full Text Search).

Un autre type de recherche de page d'aide est disponible, *par mot*, très utile dans les cas difficiles. Vous y accédez par le menu **Help**, option **Full Text Search**, ou, plus rapidement, par la combinaison **(Alt-H)+F**. À nouveau une fenêtre de dialogue est ouverte, vous demandant le ou les mots (**Word(s)**) que vous proposez. Pendant la frappe, rien n'est affiché dans la fenêtre des possibilités.

Quand le mot clé de la recherche est entré, vous tapez <**Entrée**> ou cliquez le bouton **Search**. La liste des *possibilités* est alors affichée : c'est la liste des pages où le mot clé est présent, dans le texte *ou* dans les calculs. De plus les pages sont classées par un indice de fréquence d'apparition, de sorte que souvent la page que vous cherchez figure parmi les premières affichées.

C'est un puissant moyen d'exploration. Par exemple, si vous cherchez une procédure calculant le *nombre chromatique* d'un graphe, vous tentez d'abord votre chance dans **Topic Search** avec `chrom...` mais la fenêtre des possibilités est vide. À tout hasard vous insistez et demandez la page `chromatic` : une fenêtre message accompagnée d'un flop sonore vous indique que rien n'est accessible de cette façon. C'est le moment d'essayer la recherche *par mot*. Un <Escape> efface la fenêtre inutile et **(Alt-H)+F** ouvre la fenêtre **Full Text Search**. Vous proposez le mot `chromatic`, puis <**Entrée**> vous donne la liste des possibilités :



Il n'y en a que trois, la première étant libellée «`networks,chrompoly`». Vous en

déduisez qu'un package `networks` est disponible, contenant entre autres une procédure `chrompoly`, calculant très probablement le *polynôme chromatique* d'un graphe. On sait que le nombre chromatique s'en déduit ; le package `networks` fournira tous les outils nécessaires pour construire et manipuler les graphes ; le problème est résolu.

1.7 L'édition des feuilles de travail Maple.

Cette section peut ou même doit être omise en première lecture : elle est très fastidieuse. Mais tôt ou tard, si vous ne découvrez pas par vous-même comment procéder dans telle ou telle circonstance, les informations qui y sont données devraient vous être utiles. Dans l'ensemble, les techniques d'édition disponibles sous Maple sont commodes, mais pas si simples. Quelques points sont même franchement étranges et peuvent gêner sévèrement les débutants, tant qu'ils n'ont pas bien compris l'architecture sous-jacente.

Maple a prévu le nécessaire pour que l'utilisateur puisse indéfiniment modifier les instructions déjà entrées, exécutées ou non, quelle que soit leur position dans la feuille de travail. Ceci fait, il peut les réexécuter ou non. C'est très souple, très riche en possibilités de mise au point, mais crée un espace de liberté si vaste que l'utilisateur débutant y est un peu perdu !

1.7.1 <Return> ou <Shift-Return> ?

Considérons d'abord les groupes d'exécution de plusieurs lignes. Soit à créer la procédure `SquareSum`[16] prenant en argument une liste de données et retournant la somme des carrés de ces données. On examinera plus loin le contexte théorique où le problème est défini et comment il peut être résolu. Il s'agit ici d'examiner seulement quelques points de traitement de texte, à la création de la procédure et à son utilisation. Voici une solution possible[17] :

```
> SquareSum := proc(list)
    RETURN(add(item^2, item=list))
  end ;
```

$$SquareSum := \textbf{proc}\ \ldots\ \textbf{end}$$

```
> SquareSum([2,3,4]) ;
```

$$29$$

```
> SquareSum([a,b,c]) ;
```

$$a^2 + b^2 + c^2$$

Si vous ne connaissez pas la technique ad hoc, vous aurez les plus grandes difficultés à imiter *exactement* ce qui est montré ci-dessus. La procédure `SquareSum`

[16] Noter qu'en Maple majuscules et minuscules sont distinguées dans les symboles, comme en C ou Java.

[17] La réponse (ouput) Maple pour la procédure entrée est en fait un peu plus vaste ; l'affichage simplifié prévu ici est suffisant ; on procèdera presque toujours de même dans la suite.

est organisée sur trois lignes, de sorte qu'après avoir entré la première ligne, vous allez, quoi de plus naturel, actionner la touche <**Entrée**> pour taper la ligne suivante. Vous voyez alors Maple vous compliquer la vie comme suit :

```
> SquareSum := proc(list)
> |
Warning, premature end of input
```

Maple est effectivement passé à la ligne suivante, a même inséré en début de ligne un *autre* signal de prompt (>) ; de plus, sous cette ligne, il fait figurer en petits caractères bleus un Warning signalant une fin prématurée d'input. Vous pouvez maintenant entrer la deuxième ligne `RETURN(add(item^2, item=list))`, puis taper à nouveau <**Entrée**> ; un troisième prompt s'affiche et le Warning descend pour être maintenant en quatrième ligne. Vous entrez le «`end ;`» terminal et quand pour la troisième fois vous tapez la touche <**Entrée**>, le Warning disparaît comme par enchantement et vous obtenez ceci :

```
> SquareSum := proc(list)
>   RETURN(add(item^2, item=list))
> end ;
```

SquareSum := **proc** ... **end**

La seule différence avec ce qui avait été montré au début comme solution de référence est la présence de prompts au début de *chaque* ligne de texte source de la procédure. Ce n'est pas franchement gênant, si bien que les utilisateurs débutants commencent en général par se contenter de cette façon de faire.

Pourtant, au fur et à mesure que vous progresserez, vous écrirez des procédures de plus en plus longues ; pendant une mise au point, il arrive de temps à autre qu'on décide de couper une instruction un peu longue sur une ligne de telle façon qu'elle occupe deux lignes ou plus ; d'autres fois il faut *insérer* dans le texte de nouvelles instructions devenues nécessaires, expérience faite. Considérons la procédure `log2` ci-dessous, destinée à calculer le logarithme de base 2 d'un réel positif.

```
> log2 := proc(x)
>   RETURN(evalf(ln(x)/ln(2)))
> end ;
```

log2 := **proc** ... **end**

```
> log2(3) ;
```

1.584962501

```
> log2(-3) ;
```

$1.584962501 + 4.532360143\,I$

Cette procédure est correcte, mais retourne comme il se doit un complexe si l'argument `x` est négatif. Il peut se faire, dans un contexte donné, que cette circonstance soit exclue, auquel cas le programmeur préfèrera le cas échéant interrompre immédiatement l'exécution avec un message d'erreur clair. La solution suivante[18] peut

[18] Il est plus simple de typer le paramètre, voir page 80, mais le message d'erreur peut ne pas être celui qui est souhaité.

être adoptée :

```
> log2 := proc(x)
>   if x <= 0 then ERROR
>     ("the argument of log2 must be positive.")
>   fi ;
>   RETURN(evalf(ln(n)/ln(2)))
> end ;
```

log2 := **proc** ... **end**

```
> log2(-3) ;
Error, (in log2) the argument of log2 must be positive.
```

Supposons donc que vous ayez d'abord entré la première version, puis que vous préfériez la deuxième. L'habitude des traitements de textes peut vous suggèrer pour la modification la solution suivante : positionner le curseur à la fin de la première ligne, puis entrer un «retour chariot» <**Entrée**> pour intercaler la ligne «`if x <= 0 ...`» ; vous pourrez le faire autant de fois que vous voulez, la disposition reste absolument inchangée !! De plus, à chaque essai, la définition de procédure est évaluée et le curseur va se positionner après le résultat affiché en bleu, au début du groupe d'exécution suivant.

Devant ce comportement étrange de l'éditeur Maple, vous allez probablement tenter autre chose : positionner le curseur au début de la deuxième ligne, puis entrer devant «`RETURN(...`» le début de l'instruction conditionnelle que vous voulez insérer :

```
> log2 := proc(x)
>   if x <= 0 then ERROR|RETURN(evalf(ln(n)/ln(2)))
> end ;
```

Ceci fait, vous tapez à nouveau la touche <**Entrée**> pour partager la longue ligne en deux parties, espérant continuer de la même façon sur la ligne suivante. Eh bien non, la deuxième ligne refuse la séparation ! De plus le texte procédure est *évalué* dans son état très provisoire, incompréhensible pour Maple, d'où l'erreur :

```
> log2 := proc(x)
>   if x <= 0 then ERRORRETURN(evalf(ln(n)/ln(2)))
    end ;
reserved word `end` unexpected
```

La solution *à connaître absolument* consiste à taper non pas <**Entrée**> comme il semble si naturel, mais à taper simultanément les touches «⇑»[19] et <**Entrée**>, autrement dit ce qu'on appelle par abus de langage *la* touche <⇑-**Entrée**>. Alors oui, au point où était positionné le curseur, un «retour chariot» est *inséré*. Ceci est à long terme si important qu'un énoncé très précis est nécessaire.

1. La touche <**Entrée**>, si elle est actionnée quand le curseur est dans une commande Maple, demande l'*évaluation* du groupe d'exécution englobant ;

[19] Appelée aussi touche «**Maj**» (caractères majuscules) sur certains claviers, ou encore «**Shift**» selon la terminologie anglaise.

de *tout* le groupe d'exécution, quelle qu'en soit la taille. Ce n'est que très auxiliairement que Maple introduit un «retour chariot» *à la fin* du groupe d'exécution pour séparer l'entrée (input) du groupe, en caractères télétypes rouges, de la sortie (output), en petits caractères bleus. Cette touche n'introduit *aucun* «retour chariot» là où le curseur était positionné.

2. La touche <⇑-**Entrée**> *insère* banalement un «retour chariot» là où le curseur est positionné, sans lancer quelque évaluation que ce soit.

Ce choix des concepteurs de Maple est assez discutable, mais il est ainsi. En conséquence, quand une commande nécessite plusieurs lignes de texte, il est *bien meilleur de passer à la ligne suivante à l'aide de* <⇑-**Entrée**>. Vous ne recevrez pas continuellement le Warning intempestif «`premature end of input`», et vous n'aurez logiquement qu'un seul prompt «>» au début de votre groupe d'exécution. C'est seulement quand votre groupe d'exécution sera *entièrement terminé*, par exemple quand vous aurez entré le «`end ;`» terminal de votre procédure, c'est seulement à ce moment que vous taperez une touche <**Entrée**> simple, sans «⇑», pour demander l'évaluation. Ceci *doit devenir réflexe* :

1. La touche <⇑-**Entrée**> pour le passage ordinaire à la ligne suivante ;
2. la touche à considérer comme *spéciale* <**Entrée**> pour la terminaison du groupe et l'évaluation.

Par ailleurs, les fautes de syntaxe détectables seront détectées exactement dans les mêmes conditions au moment de <**Entrée**>, ce qui permet de mieux disjoindre la phase d'écriture du texte de celle de l'analyse syntaxique et correction.

1.7.2 Bien effacer, c'est d'abord bien sélectionner.

L'état de la feuille de travail devient quelquefois si anarchique que la seule méthode de mise au propre adaptée consiste en un sérieux *nettoyage* où des *effacements* de segments de feuilles vont jouer un rôle important. C'est une opération qui n'est pas si facile pour le profane, parce que le curseur a tendance à naviguer de façon assez erratique à travers les blocs de programme surtout quand il rencontre des zones contenant des résultats (Maple-output). Un autre phénomène vient compliquer ces manœuvres : il est fréquent pendant les manipulations d'effacement que les messages d'erreur se transforment subitement en zones de textes commentaires ; les messages bleus ou mauves virent au noir sans crier gare et se mélangent au texte source (Maple Input), auquel cas l'utilisateur ne sait plus très bien que faire.

Par exemple, si, compte tenu de ce qui est expliqué ci-dessus vous avez appris à entrer les lignes successives séparées par des combinaisons <⇑-**Entrée**>, il peut vous arriver d'entrer ce texte erroné où le message d'erreur, pour une fois assez clair, vous indique clairement que vous avez une parenthèse en surnombre :

```
> SquareSum := proc(list)
    RETURN(add(item^2, item=list))|)
    end ;
')' unexpected
```

Le curseur est justement positionné là où il faut enlever la parenthèse fautive, mais vous avez le doigt un peu lourd, la touche <**Suppr**> mange non seulement la parenthèse mais la ligne «`end ;`» suivante et aspire bientôt le message d'erreur. Terrorisé, vous levez le doigt et votre feuille a alors l'allure suivante :

```
> SquareSum := proc(list)
    RETURN(add(item^2, item=list))|')' unexpected
```

où l'ex-message d'erreur est maintenant noir ! Dans ce genre de circonstance, le plus simple est de poursuivre l'effacement, donc de réappuyer sur la touche <**Suppr**> pour achever d'annihiler le message d'erreur. Si au lieu de cela vous tapez <**Entrée**> ou <⇑**-Entrée**>, le bout de message d'erreur reste vivace et d'ailleurs n'a aucune influence sur le calcul, sinon que sa présence persistante est esthétiquement bien désagréable. En fait vous avez vraiment transformé le message d'erreur en un *commentaire* dit de type **Texte**. Le lecteur aura peut-être un peu de mal à reproduire les accidents qui viennent d'être décrits, car l'expérience montre que les types de fautes commises sont extrêmement variables d'un utilisateur à l'autre, selon les habitudes de chacun, notamment selon les autres logiciels auparavant pratiqués : chaque logiciel donne peu à peu à ses utilisateurs des *habitudes*, des *réflexes* quelquefois, très différents d'un logiciel à l'autre, dont il n'est pas si facile de se débarrasser.

En tout état de cause vous serez amené assez fréquemment à entreprendre des effacements plus ou moins complexes. Sauf les effacements faciles de quelques caractères avant ou après le curseur, vous avez intérêt à *d'abord sélectionner* la zone à effacer.

Une méthode de sélection est le *balayage* avec la souris, utilisée maintenant dans la quasi-totalité des logiciels. On positionne le curseur au début de la zone à sélectionner, on clique et, *sans relâcher la pression*, on bouge la souris jusqu'à atteindre la fin de la zone à sélectionner ; pendant le balayage, une zone colorée s'étend et visualise l'extension de la partie de la feuille qui est sélectionnée. On peut même étendre la zone balayée au delà de ce qui est visible en positionnant la souris, touche restant appuyée, juste après la frontière de la fenêtre qui entre alors en déroulement pour répondre à ce qui est manifestement votre attente. Ensuite on relâche le clic et la zone finalement sélectionnée reste colorée. *Alors seulement*, vous appuyez sur la touche <**Suppr**> et la zone sélectionnée disparaît. Ceci peut être aussi la phase initiale d'une opération coupé-collé ou copié-collé. Le fait de clairement séparer la phase de sélection de la phase de suppression permet de réussir des effacements complexes, par exemple assez vastes, pouvant porter sur plusieurs groupes d'exécution, résultats compris.

La phase de sélection peut aussi être réalisée sans souris, avec le curseur, avec

une précision encore meilleure. Le principe est simple : si pendant l'action des touches de mouvement du curseur, quelles qu'elles soient, vous maintenez *en plus* la touche «⇑» enfoncée, la zone parcourue par le curseur est sélectionnée. Par exemple, si, cas assez fréquent, vous voulez effacer *toute* la feuille à partir d'un point donné, vous avez un moyen très simple pour le faire : vous positionnez le curseur au point coupure, puis vous actionnez *simultanément* les *trois* touches <**Ctrl-⇑-Fin**> ; ce seul geste sélectionne la zone voulue. Ensuite un coup de <**Suppr**> et la suppression est faite. Notez aussi dans ce registre que si vous êtes positionné en début de groupe d'exécution, juste après le prompt Maple (>), vous pouvez en actionnant deux fois la touche directionnelle «←» positionner le curseur *avant* le prompt. Vous pouvez alors commencer la sélection de ce point si vous souhaitez entre autres effacer *tout* le groupe d'exécution, prompt compris.

Conseil 6 — *Prenez l'habitude pour les effacements un peu complexes de sélectionner d'abord, à la souris ou au clavier, la zone à effacer. Ceci fait vous pouvez alors procéder à l'effacement ou encore à un coupé ou un copié pour un collé ultérieur.*

Vous pouvez aussi effacer un *paragraphe* à l'aide de la combinaison <**Ctrl-Suppr**> ; c'est quelquefois assez commode pour supprimer rapidement de grandes zones de texte. Se méfier toutefois d'un piège : après la suppression, il est fréquent que le curseur ne soit pas positionné après le paragraphe supprimé, mais plus loin, ce qui est un peu désagréable. Un paragraphe est une zone de texte source (Maple Input) située entre deux prompts, ou *le* résultat d'*une* instruction, ou encore un segment de texte commentaire disjoint du texte source Maple.

1.7.3 Les fins de lignes étranges.

On a expliqué plus haut pourquoi il fallait préférer <⇑-**Entrée**> à <**Entrée**> pour le passage à la ligne suivante, lors de l'entrée d'*une* commande de plusieurs lignes. Vous n'êtes pas encore complètement tiré d'affaire ! Il vous reste alors à amadouer les comportements exotiques du curseur en fin de ligne, à l'origine de crises de nerfs mémorables...

Les textes sources Maple (Maple-Input) où les lignes sont séparées par des <⇑-**Entrée**> présentent en effet une caractéristique étrange, assez déroutante pour les débutants. Le dernier caractère d'une ligne *terminée*, qui «paraît» être un blanc, est en fait un séparateur de lignes. Ceci ne se produit que pour les lignes proprement terminées par <⇑-**Entrée**>. Une fois ceci bien compris, les conséquences sont logiques, mais il faut un certain temps pour acquérir les réflexes ad hoc.

Reprenons notre procédure `SquareSum` pour mettre en évidence ces phénomènes. Supposons qu'on oublie l'une des deux parenthèses fermantes de la ligne 2 ; à l'évaluation, on obtient le résultat suivant ;

```
> SquareSum := proc(list)
    RETURN(add(item^2, item=list)
    end ;
reserved word 'end' unexpected
```

Vu la position du curseur, pour atteindre le point fautif avec le clavier, on procède en principe comme suit : on entre la touche directionnelle «↑» pour atteindre la ligne critique, puis la touche **Fin** pour aller à la fin de cette ligne, là où la parenthèse est manquante. Curieusement, le curseur se positionne en fait un caractère plus loin, après ce qu'on *croit* être un espace, qu'en fait on n'avait pas entré ! Selon l'humeur du moment, si on ne connaît pas la véritable nature de ce caractère mystérieux, on choisit l'une de ces deux solutions :

1. On décide d'effacer l'espace supposé par un **BackSpace** et on a la stupéfaction de voir la ligne suivante «`end ;`» remonter derrière le curseur !

```
> SquareSum := proc(list)
    RETURN(add(item^2, item=list)    end ;
reserved word 'end' unexpected
```

2. On décide au contraire de laisser cet espace et on entre immédiatement la parenthèse manquante «`)`» ; cette fois on a la surprise de la voir apparaître en fait au début de la ligne suivante !

```
> SquareSum := proc(list)
    RETURN(add(item^2, item=list)
  )
    end ;
reserved word 'end' unexpected
```

Mais qu'est-ce qui peut expliquer ces étrangetés ? On a tout compris quand on sait que le caractère en fin de ligne qu'on croyait être un espace était en fait un *séparateur de lignes*, souvent appelé «retour chariot» (Endline ou Return[20] selon la terminologie anglaise). Supposons donc que le curseur soit positionné en fin de ligne derrière ce caractère déguisé en espace ; alors :

1. Si vous entrez un **BackSpace**, vous supprimez le séparateur de lignes et la ligne suivante vient donc se positionner à la fin de la ligne en cours !
2. Si au contraire vous entrez un caractère banal, il est entré *à droite du* séparateur de lignes, donc sur la ligne suivante !

C'est d'une logique imparable, au moins quand on *sait*. Que fallait-il donc faire pour insérer notre parenthèse diabolique à la bonne position ? Il fallait, après avoir positionné le curseur en fin de ligne à l'aide de la touche **Fin**, utiliser la touche directionnelle «←» pour positionner le curseur *avant* le séparateur de lignes ; on peut alors enfin entrer la parenthèse là où c'est nécessaire.

On vous conseille de consacrer à l'occasion un quart d'heure ou une demi-heure pour bien comprendre ces mécanismes. Une technique vous aidera grandement. Le

[20] L'analyse des fichiers Maple montre que ce n'est en fait ni l'un ni l'autre, car Maple a ses propres méthodes de codage pour les séparateurs de lignes.

bouton  de la barre d'outils rend «visibles» les caractères normalement invisibles sur votre feuille de travail. Les phénomènes ésotériques expliqués longuement dans cette section seront alors très clairs. Dans le même registre, vous pouvez ainsi *voir* que les touches <**Entrée**> et <⇑-**Entrée**> ne produisent pas du tout le même résultat.

1.7.4 Autres commandes d'édition.

Insertion d'un groupe d'exécution.

Fréquemment vous déciderez d'insérer un nouveau groupe d'exécution entre deux groupes déjà existants. Ceci se fait à l'aide de **Ctrl-J** (ajout *après* le groupe où le curseur est positionné) ou **Ctrl-K** (ajout *avant*). Rien n'empêche de créer successivement de la sorte plusieurs groupes vierges consécutifs, par exemple pour compléter la feuille de travail par de nouveaux calculs intermédiaires qui se révèlent nécessaires. Noter que sous Unix, il est fréquent que ces commandes «décident» d'être inopérantes, ce qui est fort désagréable. Il faut alors se rabattre sur la commande menu «**Insert**|**Execution Group**».

Undo et redo.

Il est assez fréquent qu'on souhaite *défaire* ce qu'on vient de *faire*, surtout quand ce qu'on vient de «faire» est une suppression, auquel cas il s'agit plutôt de refaire ce qu'on vient de défaire... La commande **Undo** du menu **Edit** est prévue à cet effet. Le raccourci clavier **Ctrl-Z** est commode pour la répétition de cette commande, autrement dit pour continuer à défaire ; mais la mémoire de Maple dans ce registre est assez réduite. On peut inverser le mécanisme avec la commande **Redo**.

Trouver.

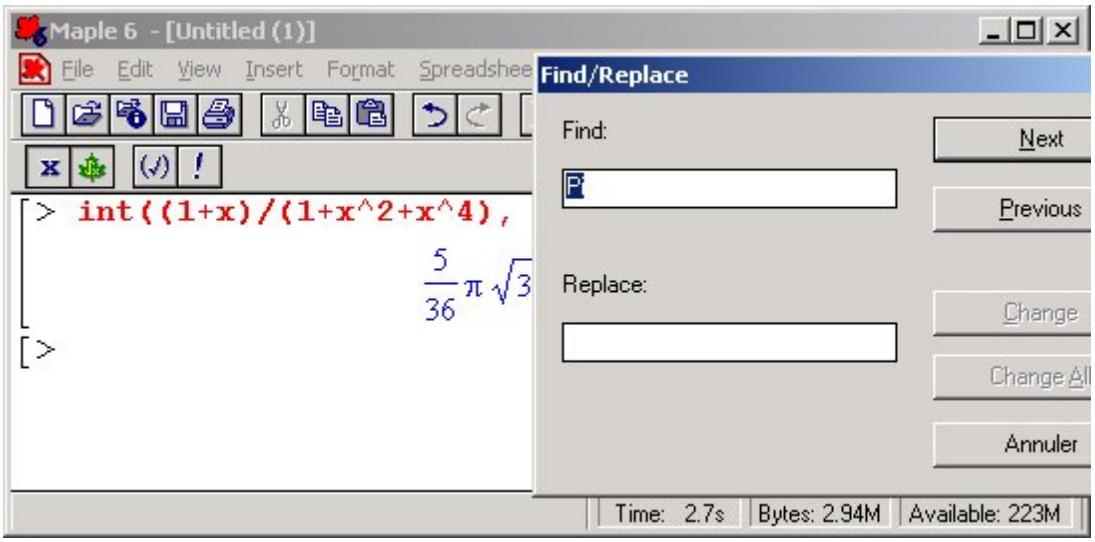
La commande **Find** du menu **Edit** ouvre une fenêtre de dialogue vous demandant quelle chaîne de caractères vous voulez chercher ; le raccourci **Ctrl-F** est disponible. Quand le texte à chercher est défini dans la mini-fenêtre, vous avez encore le choix de chercher en avant ou en arrière.

Cette technique est utile quand votre feuille de travail devient importante et que vous n'avez pas une mémoire précise du point où par exemple une modification doit être effectuée. C'est aussi un outil bien commode pour l'exploration des feuilles d'aide un peu vastes ; on s'y sent souvent un peu noyé dans la multitude d'indications techniques concernant les nombreux cas d'utilisation prévus ; pensez alors à **Ctrl-F** : ce peut être le moyen de localiser rapidement l'information qui vous intéresse.

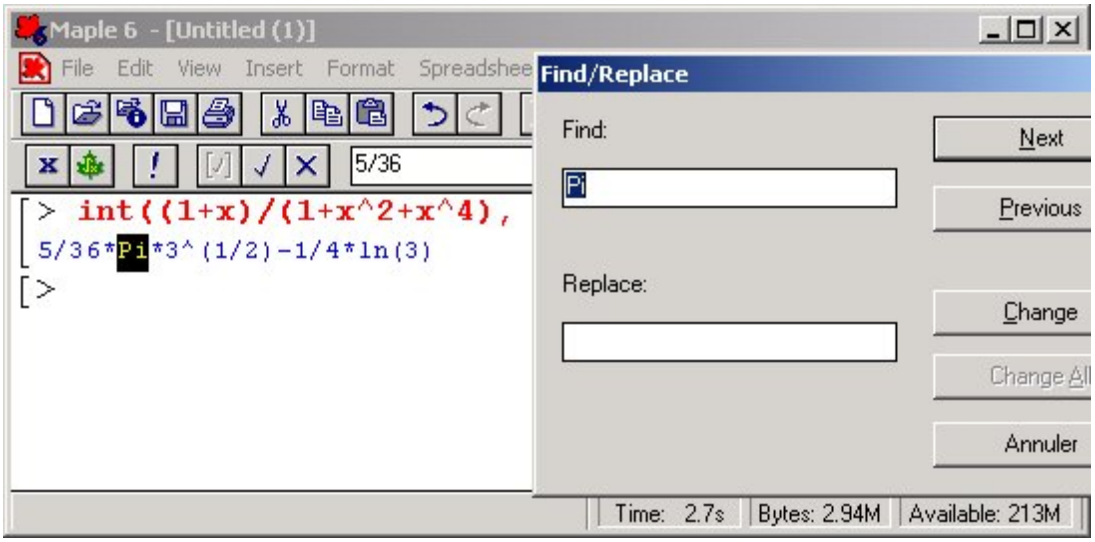
Noter que la recherche n'est pas possible de cette façon dans les *résultats* des calculs (Maple output). C'est dommage, mais quand ceci devient quasiment indis-

pensable, par exemple pour rechercher l'occurence de telle chaîne dans un résultat fleuve occupant plusieurs dizaines d'écrans, vous pouvez procéder comme suit. Choisissez l'option **Maple Notation** (menu **Options**, option **Output Display**), et faites réexécuter l'instruction critique. Le résultat est alors affiché sous forme d'une chaîne ASCII où la recherche devient possible. Vous pouvez ensuite revenir à l'option **Standard Math Notation** pour revenir aux affichages sorties Maple standard.

Comparer par exemple les affichages suivants ; dans le premier cas, si l'affichage résultat est **Standard Math Notation**, la recherche de la chaîne de caractères `Pi` est inopérante :



alors que si l'affichage choisi est **Maple Notation** :



le résultat est beaucoup moins lisible, mais par contre la recherche de la chaîne `Pi` réussit. Dans ce petit exemple, la différence est insignifiante, mais, si, comme cela arrive de temps en temps, un ou des résultats occupent plusieurs écrans, ce peut être le point clé pour localiser tel ingrédient particulier.

Copie «output → input».

Dans le même ordre d'idées, il est quelquefois utile de savoir copier une zone output de la feuille de travail vers une zone input. C'est notamment une astuce à connaître pour préparer des démonstrations sophistiquées où par exemple une matrice *input* relativement grande est d'abord générée de façon aléatoire.

Montrons comment il faut procéder. Supposons qu'on veuille commencer avec une matrice 5×10 dont tous les termes sont compris entre -9 et $+9$; il est très fastidieux de créer soi-même une telle matrice et on ne peut jamais simuler correctement que cette matrice est vraiment «quelconque» ; la seule façon d'y arriver consiste à utiliser un générateur aléatoire. Une telle matrice peut être obtenue comme suit, en utilisant la procédure `rand`.

```
> MM := matrix(5, 10, rand(-9..+9)) ;
```

$$MM := \begin{bmatrix} 8 & 4 & -5 & -5 & 3 & -5 & 8 & 5 & -1 & 0 \\ 3 & -4 & -5 & -2 & -1 & -9 & 5 & 8 & 4 & -3 \\ 8 & -9 & 7 & 5 & 0 & 4 & 4 & 2 & 5 & -5 \\ 9 & -9 & 4 & 0 & -8 & -5 & 4 & 7 & -9 & 5 \\ 7 & -8 & 3 & 5 & -9 & 4 & 0 & -1 & -5 & 5 \end{bmatrix}$$

Vous essayez la suite de votre démonstration avec cette matrice tombée du ciel et si elle correspond à votre projet, vous pouvez vouloir donner l'impression que cette matrice a été choisie par vous-même ! Il est assez fastidieux de recopier soi-même l'instruction Maple ad hoc :

```
> MM := matrix([[8, 4, 5, ...
```

Pour éviter ce travail pénible, vous pouvez balayer à la souris l'*output* (bleu) $MM := \ldots$; un gros pavé noir surcharge votre output qui apparaît maintenant en jaune sur fond noir. Vous actionnez la combinaison clavier **Ctrl-C** ou **Ctrl-Insert** pour copier ce qui est sélectionné dans ce que Microsoft-France appelle le «presse-papiers», actionnez ensuite **Ctrl-J** pour faire apparaître un nouveau groupe d'éxécution, puis **Ctrl-V** ou **⇑-Insert** pour y copier le contenu du presse-papiers ; vous obtenez ainsi après quelques manœuvres évidentes d'édition quelque chose comme :

```
> MM := matrix(
          [[8, 4, -5, -5, 3, -5, 8, 5, -1, 0],
           [3, -4, -5, -2, -1, -9, 5, 8, 4, -3],
           [8, -9, 7, 5, 0, 4, 4, 2, 5, -5],
           [9, -9, 4, 0, -8, -5, 4, 7, -9, 5],
           [7, -8, 3, 5, -9, 4, 0, -1, -5, 5]]) ;
```

avec le même résultat que précédemment à l'évaluation, à condition de ne pas avoir oublié d'ajouter le point-virgule indispensable en fin d'instruction, absent de l'output original. Ceci fait, vous pouvez maintenant supprimer de votre feuille de travail l'instruction initiale réalisant la génération aléatoire de la matrice par l'intermédiaire de la procédure `rand`. Vous obtenez ainsi une feuille de travail don-

nant l'impression que cette matrice a été choisie par vous-même ! Cette technique est quelquefois *abolument indispensable* quand la génération devient vraiment (!) aléatoire, par exemple quand on utilise la procédure `randpoly` pour la génération de polynômes aléatoires.

On réutilisera cette technique de copie «output → input» page 47 pour une autre raison qui sera expliquée alors, essentiellement pour mettre en évidence la différence entre format externe d'un objet et objet proprement dit.

-o-o-o-o-o-o-o-

Chapitre 2

Le modèle «Read-Eval-Print» de Maple

2.1 Pourquoi un modèle ?

L'architecture des grands logiciels n'est pas simple, leur utilisation non plus. Mais la *formation* à leur utilisation l'est encore moins ! On entend dire souvent, quand il s'agit par exemple de planifier un cours Maple semestriel, qu'on s'organisera pour se débarrasser rapidement des technicités informatiques et aborder aussi vite que possible la seule partie noble de cet enseignement, celle qui est consacrée aux «vraies» mathématiques. Ceux qui tiennent ce discours ne savent pas de quoi ils parlent.

L'utilisation sérieuse de Maple, comme celle de tous les logiciels d'envergure, y compris les plus populaires, nécessite une initiation *informatique* à ne négliger sous aucun prétexte. L'outil usuel pour ce faire consiste à présenter, explicitement ou implicitement, un *modèle* de fonctionnement. Le modèle est une version abstraite ultra-simplifée du logiciel permettant de décrire de façon aussi élémentaire que possible les grandes lignes de sa structure. Un logiciel a un *noyau* (kernel) qui est en quelque sorte son cœur[1]. Le *modèle* est de façon analogue le noyau de son apprentissage. On utilisera ici le modèle Read-Eval-Print, selon la terminologie devenue usuelle pour présenter la structure du fonctionnement de la machine Lisp.

2.2 Read-Eval-Print.

2.2.1 Cycle Input-Output.

L'utilisateur de Maple voit d'abord un cycle Input-Output. Pendant la phase Input, il entre au clavier une instruction Maple ; cette phase est terminée par l'entrée d'un « **;** » *et* de <**Entrée**>. La phase Output est alors lancée, se terminant

[1] Un étymologiste amateur peut raisonnablement suspecter que la coïncidence sonore «*cœur*» ↔ «*ker*nel» n'est pas accidentelle.

dans les bons cas par l'affichage juste après l'instruction entrée du résultat obtenu. Un nouveau groupe d'exécution est ouvert, et le curseur est positionné juste après le prompt. Une nouvelle phase Input commence où l'utilisateur va entrer son instruction suivante, etc.

```
> 4 ;
```

$$4$$

Dans l'exemple précédent particulièrement simple, l'utilisateur a entré l'unique *instruction* «`4`» (Input) et la phase Output a produit l'objet 4. Dans l'exemple d'intégration qui avait été un peu considéré Section 1.4 :

```
> int((1+x)/(1+x^2+x^4), x=1..infinity) ;
```

$$\frac{5}{36}\sqrt{3}\pi - \frac{1}{4}\ln(3)$$

l'Input est la chaîne de caractères «`int(...infinity)`», alors que l'Output est la jolie formule résultat.

L'utilisateur peut en fait entrer *plusieurs* instructions Maple séparées par point-virgule, à condition de ne jamais taper le caractère <**Entrée**> qui demande l'exécution, mais «seulement» la combinaison <⇑**-Entrée**> pour chaque fin de ligne. Autrement dit on peut «empiler» plusieurs phases Input avant de faire exécuter les différentes phases Output correspondantes :

```
> Int(x^2, x) = int(x^2,x) ;
  Diff(Int(x^2, x), x) = diff(int(x^2, x), x) ;
```

$$\int x^2\,dx = \frac{1}{3}x^3$$

$$\frac{\partial}{\partial x}\int x^2\,dx = x^2$$

Mais par rapport au modèle qu'il s'agit de décrire, ces considérations sont secondaires.

Le calcul de l'intégrale $\int_1^\infty \frac{1+x}{1+x^2+x^4}\,dx$ demande sur un PC de puissance moyenne environ une demi-seconde de calcul, et c'est précisément là que Maple est utile. C'est la phase d'*évaluation*. L'*évaluateur* Maple a considéré l'Input entré par l'utilisateur (pléonasme) ; l'évaluateur était missionné pour l'*évaluer*. C'est un travail qui peut être très bref, comme plus haut quand il s'agissait d'évaluer l'Input «`4`» ; Maple constate alors presque instantanément qu'il n'y a aucun travail digne de ce nom à effectuer, et il *retourne* aussitôt l'Output «`4`» ; c'était très bref, mais il est quand même capital pour la suite de comprendre qu'il y a bien eu une phase d'évaluation. Au contraire, pour le calcul de l'intégrale, la phase d'évaluation est fort complexe : de nombreuses recherches doivent être faites par Maple pour comprendre qu'il doit effectuer une intégration, qu'il doit examiner la nature de l'intégrande, voir si l'intégrale est définie ou indéfinie, adopter une stratégie pour trouver une primitive, la mettre en œuvre, etc. C'est un travail sophistiqué, mettant en œuvre des ressources très variées, mais peu importe, tout ceci est quand même *une* phase d'évaluation.

2.2.2 L'évaluateur.

Il est donc indispensable d'enrichir le cycle Input-Output pour en faire un cycle à trois temps que, compte tenu d'une très ancienne tradition provenant de Lisp, on préfère appeler le cycle Read-Eval-Print. Cette terminologie a un autre avantage, elle prend définitivement le point de vue de Maple. Maple apparaît comme un chef d'orchestre avec trois exécutants :

1. Le *Lecteur* : c'est la composante de la machine Maple chargée de *lire* le texte entré par l'utilisateur, autrement dit l'Input. Le paradoxe usuel, si coriace pour les super-débutants, est toujours là ; l'utilisateur a l'impression au contraire d'*écrire*, mais son point de vue ne nous intéresse pas ; on ne veut considérer que le point de vue de Maple, qui doit bel et bien *lire* un objet Maple, appelé Input, à savoir l'objet écrit par l'utilisateur. Le Lecteur est l'exécutant consacré à cette tâche.
2. L'*évaluateur* : c'est vraiment le cœur de Maple ; l'évaluateur doit *évaluer* l'objet Maple que le Lecteur vient de lire. Ce peut être un travail quasi-instantané ; inversement il peut à l'occasion nécessiter plusieurs jours de calcul ; peu importe, c'est *une* évaluation et c'est l'évaluateur qui en est l'exécutant.
3. L'*«Imprimante»* qu'il est plus raisonnable dans le contexte d'aujourd'hui d'appeler l'*Afficheur* : une fois que l'évaluateur a terminé son travail, l'Afficheur doit faire le nécessaire pour afficher le résultat de l'évaluation à l'écran, résultat obtenu par l'évaluateur. C'est un travail assez technique, peu important en théorie, très délicat en pratique, mais absolument indispensable sans quoi le travail de l'évaluateur serait perdu. L'Afficheur est dédié à cette troisième phase.

On a déjà utilisé sans explication une terminologie qu'il est temps d'éclaircir. L'évaluateur travaille en fonction de l'expression (Input) qu'il doit évaluer. Quand il a terminé son travail, on dit qu'il *retourne* le résultat, c'est l'Output que l'afficheur doit maintenant afficher. Cette terminologie est devenue usuelle en informatique, mais l'expérience montre qu'elle trouble encore beaucoup de mathématiciens, et non des moindres... Pour la justifier, il faut penser que l'évaluateur est un correspondant qui reçoit, disons par la Poste, une *lettre* lui commandant tel ou tel travail ; quand le travail est exécuté, l'évaluateur en *retourne* le résultat au demandeur, en fait par l'intermédiaire de l'afficheur. En particulier il ne s'agit pas d'un refus *retour à l'expéditeur* ! La commande est en général honorée et ce qui est *retourné* est au contraire, au moins les bons jours, ce qui est souhaité !

Maple se contente de faire travailler les trois exécutants Lecteur, Evaluateur et Afficheur à tour de rôle. Compte tenu de cette organisation en cycles Read-Eval-Print, l'utilisateur de Maple doit organiser son travail sous forme d'une suite d'*évaluations* d'objets à définir en fonction du problème à résoudre. Le travail de compréhension de Maple se partage donc parallèlement en trois parties.

1. On doit savoir comment travaille le *Lecteur* de Maple ; bien entendu la connaissance de la structure interne du lecteur ne nous intéresse pas ; par contre nous devons connaître les mécanismes permettant de décrire l'objet

Maple dont on va demander l'évaluation. Cet objet a une structure interne qui peut être assez riche, et il va falloir comprendre comment il est possible de décrire cet objet plus ou moins complexe à l'aide d'une chaîne de caractères ordinaires, l'Input transmis au lecteur.

2. On doit savoir transformer notre problème mathématique à résoudre en une ou plusieurs évaluations. Ceci nécessite bien sûr une connaissance parfaite du travail de l'évaluateur. La nature *récursive* de l'évaluateur joue de ce point de vue un rôle essentiel.
3. Un minimum de technique sera aussi nécessaire pour que l'affichage du résultat de l'évaluation corresponde raisonnablement au but qui est recherché.

2.2.3 Récursivité des types de données.

Un autre ingrédient facilite sensiblement la couverture de ces sujets. On expliquera en temps utile ce qu'il faut comprendre par nature *récursive* de l'évaluateur. Avant de penser évaluation, il faut considérer quels sont les objets évaluables, autrement dit quels sont les différents objets sur lesquels on peut faire travailler l'évaluateur. C'est une question de *structure de données*, l'un des sujets clés de l'informatique moderne. Il se trouve que la définition des objets Maple, plus précisément de leur structure possible, est elle aussi *récursive*. Ceci veut dire qu'un nombre relativement petit de structures élémentaires va être suffisant pour décrire, grâce à la récursivité du mécanisme, *tout* objet Maple. Par exemple l'objet :

```
int((1+x)/(1+x^2+x^4), x=1..infinity)
```

est un objet **function**[2]. Un tel objet est toujours de la forme (externe) :

```
obj0(obj1, obj2, ..., objn)
```

Peu importe ce que sont les objets composants `obj0`, ..., `objn`, notre objet composé est un objet **function** à propos duquel des règles *universelles* de construction, d'évaluation et d'affichage sont applicables. Les différents composants sont des objets quelconques ; ils peuvent être à leur tour, pourquoi pas, des objets **function**, ou bien des objets d'autres *types* qu'on étudiera le moment venu, il y en a une grande variété. Dans notre exemple, le composant `obj0` est l'objet **symbol**[3] «`int`», le composant `obj1` est l'objet «`*`»[4] «`(1+x)/(1+x^2+x^4)`», enfin le dernier composant `obj2` est l'objet **equation** «`x=1..infinity`». Autrement dit tout *type de donnée* va donner lieu à la description des objets de ce type comme constitués d'objets composants, en général quelconques, assemblés selon des règles ne dépendant que

[2] Les questions de types sont importantes ; on utilise pour le souligner une police un peu sophistiquée pour les noms de types ; choisis par les concepteurs de Maple, ce sont des noms anglais.

[3] Ainsi *symbole* équivaut à «objet **symbol**».

[4] À de rares exceptions près, Maple code de façon interne les quotients `A/B` comme le produit `(A^1)*(B^(-1))`, de sorte que pour Maple ces objets qui extérieurement paraissent être des quotients sont en fait des produits, de type «`*`».

du type de donnée en question. Certains types sont *terminaux*, on les appelle *atomiques* : ce sont des objets irréductibles en plus petits objets. Les symboles sont de cette nature.

Ce mécanisme est très bénéfique. Il va donc s'agir en définitive de bien connaître les types de données disponibles, qu'on peut considérer comme des *constructeurs* élémentaires, capables de construire à partir d'autres objets, simples ou complexes, peu importe, un nouvel objet certainement plus complexe. Pour chaque type de donnée, il va falloir connaître sa structure *interne* ; mais un modèle *théorique* pour cette structure interne, plus commode, suffira ; il va falloir aussi connaître sa forme *externe*, autrement dit comment on peut préparer un tel objet comme une chaîne de caractères qu'on donnera en Input au lecteur Maple pour qu'il construise effectivement l'objet souhaité, en vue de son évaluation . Le plus souvent on retrouvera la même forme externe à l'affichage, si un objet de ce type est un résultat d'évaluation ; cependant, pour le dernier point, un affichage plus proche des habitudes des mathématiciens est souvent préféré. Considérons par exemple l'intégrale préparée pour le Lecteur plus haut ; il pourrait se faire que cette intégrale (non évaluée) soit au contraire le *résultat* d'une évaluation, auquel cas elle serait alors affichée :

$$\int_1^\infty \frac{1+x}{1+x^2+x^4}dx$$

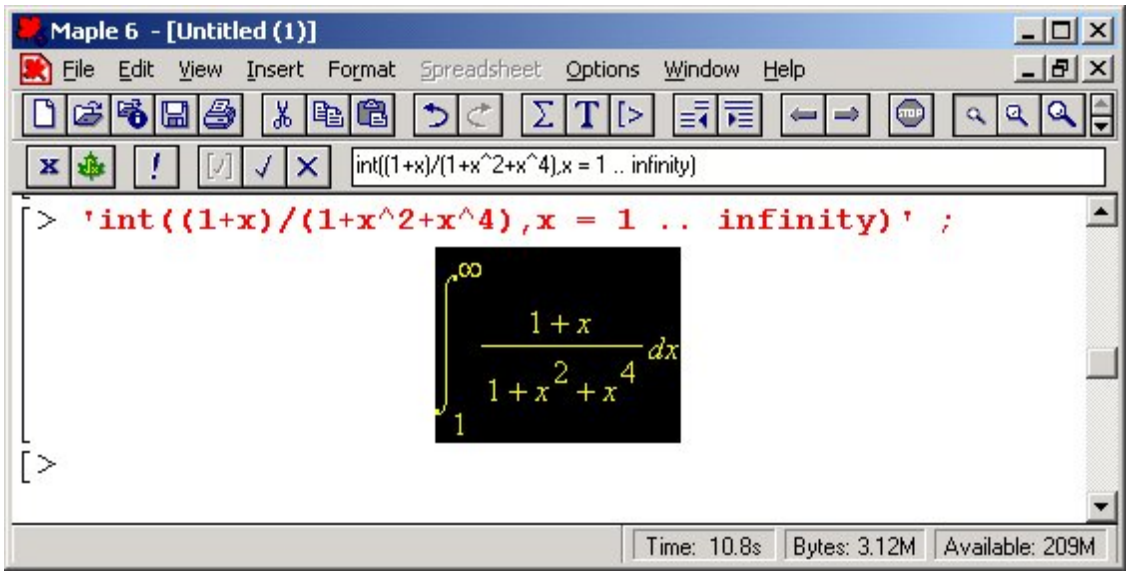
mais ces différences entre forme externe *à lire* et forme externe *affichée* sont assez secondaires.

Faisons cette petite expérience. Soit l'instruction :

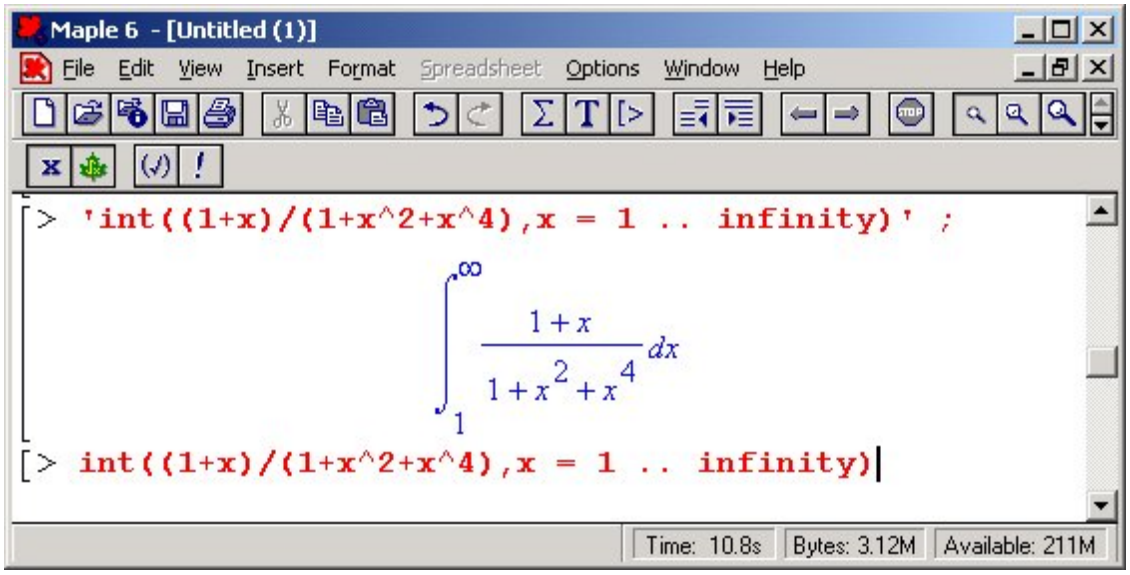
```
> 'int((1+x)/(1+x^2+x^4), x=1..infinity)' ;
```

$$\int_1^\infty \frac{1+x}{1+x^2+x^4}dx$$

On a entré à nouveau notre intégrale fétiche, à ceci près qu'elle est maintenant *quotée* : le premier et le dernier caractère sont maintenant des *quotes*, c'est-à-dire des apostrophes ou, si on préfère, des accents aigus isolés. L'anglicisme *quote* est laid, mais sa sémantique est si particulière qu'on préfère l'utiliser. Il s'agissait ici d'empêcher l'évaluation de l'intégrale et, comme on l'observe à l'écran, l'intégrale est retournée *telle quelle*, sauf qu'elle *apparaît* sous la forme typographique usuelle des manuels de mathématiques. Notre expérience consiste à nous démontrer que la forme *interne* est bien la même. Pour ce faire on amène la souris sur le résultat qui vient d'être obtenu, et on sélectionne ce résultat par la manœuvre usuelle de balayage : la belle intégrale est maintenant dessinée en jaune dans un rectangle noir ; Maple fait plus, il affiche dans une mini-fenêtre de la barre contextuelle, la forme dite **Maple-Notation** de la même intégrale ; on l'a soulignée par une flèche rouge dans la reproduction ci-dessous :



On voit que cette forme **Maple-Notation** n'est autre que notre Input, à ceci près que les quotes d'encadrement ont disparu. On détaillera ce point un peu plus loin[5]. On demande un *copié* de cette intégrale, par exemple par **Ctrl-Insert**. On positionne ensuite le curseur au prompt en attente et on demande cette fois un *collé*, par exemple par <⇑**-Insert**>. On voit alors apparaître l'intégrale sous la forme nécessaire pour un *Input*, une chaîne de caractères rouges «`int((1+x)/...infinity)`», celle qu'on a déjà vue plusieurs fois :



La différence d'affichage entre ce que vous voyez là où le copié a été fait, d'une part, et là où le collé a été fait, d'autre part, est due au contexte qui n'est pas le même ; Maple en a tenu compte pour afficher à sa guise le *même* objet *interne*

[5] Règle Eval-10

sous deux formes externes assez différentes en apparence, équivalentes sur le fond. L'opération *copié* aurait aussi bien pu être effectuée dans la fenêtre contextuelle.

On peut mettre en évidence la séparation du travail entre Lecteur et Evaluateur d'une part, Evaluateur et Afficheur d'autre part, comme suit. Considérons l'expression absurde suivante, dont prudemment on empêche d'abord l'évaluation (quotes) :

```
> 'int(sin(x), [1,2,3])' ;
```

$$\int \sin(x) d[1,2,3]$$

```
> % ;
Error, (in int) wrong number (or type) of arguments
```

L'intégrale est une expression qui, sans convention préalable, n'a pas de sens. Mais c'est un objet Maple *légal*, que le Lecteur de Maple sait lire, que l'Afficheur sait afficher. L'instruction suivante demande l'évaluation de cet objet ; l'évaluateur ne peut traiter cette intégrale et *retourne* donc un message d'erreur.

2.3 Notion d'environnement.

2.3.1 Un peu d'histoire.

Quand une session Maple est en repos, en attente de la prochaine commande, l'état *interne* de la session est décrit par ce qu'on appelle un *environnement*. C'est à nouveau une question de *modèle* : l'état interne en question n'est autre que la longue chaîne de bits en attente dans la *mémoire* de Maple, plus précisément dans les pages mémoire réservées à Maple par le système d'exploitation. Cette liste plutôt indigeste de 0's et de 1's va pourtant vous permettre des calculs peut-être de la plus haute importance. Pour la maîtriser, il est nécessaire de la *modéliser* ; le concept moderne d'*environnement* est l'outil clé.

Turing a inventé sa fameuse machine théorique au milieu des années 30 [3]. La mémoire de sa machine était aussi une suite de 0's et de 1's dont l'état évoluait pendant le calcul ; à la fin du calcul, on y lisait le résultat. Quand Von Neumann a réalisé la première machine concrète en s'inspirant du modèle de Turing, il y a ajouté une amélioration : le mécanisme d'adressage de la mémoire : chaque unité mémoire est repérée par un numéro, son *adresse* ; les instructions machines citent souvent de telles adresses pour utiliser d'une façon ou d'une autre le contenu de l'unité mémoire correspondante, ou encore pour y copier une chaîne de bits à utiliser plus tard. Le modèle théorique correspondant est la machine RAM [1].

Pour des programmes complexes, la gestion des adresses numériques devient vite insupportable ; divers mécanismes de représentations *symboliques* des adresses ont été imaginés pour décharger le programmeur. Les *langages assembleurs*, même les plus primitifs, disposent d'un tel mécanisme ; les langages évolués a fortiori. Dans un programme C, une déclaration «`int n ;`» demande au compilateur la réservation d'un mot mémoire où un entier peut être installé ; son adresse réelle

restera inconnue de l'utilisateur, mais elle pourra être citée *indirectement* à l'aide du *symbole* `n`. Si la déclaration est locale à une procédure appelée dynamiquement, l'adresse réelle machine est en général variable d'une invocation à l'autre de la procédure, mais le mécanisme d'*adressage symbolique* dispense justement l'utilisateur des technicités sous-jacentes : elles sont gérées par le système.

Ce mécanisme d'adressage symbolique s'est à son tour montré insuffisant. Les méthodes d'indirections *dynamiques* utilisées en langages machines et assembleurs depuis longtemps étaient inaccessibles aux utilisateurs de langages dits évolués comme Fortran[6]. Pour y remédier, on a imaginé un nouveau type de donnée, celui de *pointeur* : un symbole de type pointeur représente une case mémoire contenant l'adresse d'une autre case mémoire. Les mécanismes satellites d'allocation mémoire avec mise à jour d'un pointeur vers l'adresse de la zone mémoire allouée permettent de nouvelles techniques de programmation très puissantes, traitements de listes et d'arbres notamment. Les langages PL/I puis Pascal ont été les premiers langages populaires ayant parfaitement intégré ce type de données.

Les langages plus modernes, Lisp notamment[7], sont allés plus loin. Par une organisation soigneuse des types de données disponibles, la notion de pointeur y a été *en apparence* complètement supprimée, alors qu'*en fait* les pointeurs y sont plus présents qu'ils ne l'ont jamais été ! C'est un point très intéressant de l'évolution des langages informatiques. L'idée clé est la suivante : compte tenu de l'importance des pointeurs, au lieu de laisser à l'utilisateur la liberté de décider quand un objet est de type pointeur ou non, on organise le langage comme si tout objet était un *pointeur potentiel* ; c'est le système sous-jacent qui prendra en définitive les décisions à ce propos. Typiquement, si un symbole `n` «repère» un entier, le système doit de toute façon réserver un mot mémoire pour `n` ; si la «valeur» de `n` est un entier court, le système peut décider d'installer cet entier directement dans la case réservée pour `n` ; si au contraire c'est un entier long ou très long, le système décide alors d'installer dans la case `n` l'*adresse* d'une liste contenant l'ensemble de tous les ingrédients nécessaires pour définir l'entier long en question ; quelques bits «système» sont réservés dans la case `n` pour indiquer quelle est la solution utilisée à un instant donné. Ce type de mécanisme est devenu standard dans les langages très évolués. L'utilisateur n'est d'ailleurs pas directement concerné par les techniques d'implémentation ; pour lui tout se passe comme si un symbole était *toujours* un pointeur.

C'est l'organisation adoptée dans Maple. Un symbole est un pointeur vers un objet de type quelconque, à quelques variantes près qu'on verra en temps utile. Le couple `[symbole → objet pointé]` est un *chaînage*. L'ensemble de tous les chaînages en mémoire à un moment donné est l'*environnement* en cours, et c'est cet objet assez sophistiqué qu'il s'agit d'étudier dans cette section.

[6] Les versions récentes de Fortran, depuis Fortran 90, disposent aussi du type pointeur ; la notion de classe est prévue pour 2020 et la programmation fonctionnelle pour 2050.

[7] Lisp est en fait un langage très ancien, qui est né au début des années 60, mais il n'est devenu utilisable pour les grands projets qu'assez récemment, depuis la définition de la norme ANSI *Common Lisp*.

2.3.2 Chaînages.

Vocabulaire.

De bons petits exemples valent mieux que des tonnes de théorie. Considérons cette micro-session Maple :

```
> x := 34 ;
```

$$x := 34$$

Quand vous commencez une session Maple, l'environnement en cours est l'*environnement initial*. Ci-dessus l'exécution de l'instruction «`x := 34`» *modifie* l'environnement initial. Un nouveau *chaînage* `[x → 34]` y est ajouté, alors qu'auparavant aucun chaînage de *source* `x` n'y était *présent*. Si un chaînage de source `x` est présent dans l'environnement, par abus de langage, on dit que le symbole `x` est *présent* dans l'environnement, ou plus brièvement qu'il est présent. L'environnement initial n'est autre que l'ensemble de tous les chaînages préinstallés quand une session Maple est commencée. Un chaînage est un couple `[symbol → object]` de *source* un symbole et de *but* un *objet* Maple quelconque ; on dira simplement un *objet*.

Un environnement est donc un ensemble de chaînages décrivant quels sont à ce moment les symboles *affectés*. Chaque symbole affecté est la source d'un chaînage, dont le but est un objet quelconque. Un symbole est source au plus d'*un chaînage*. Un symbole qui n'est pas affecté est un *symbole libre* ; ces symboles libres vont jouer un rôle essentiel en calcul symbolique, un type de calcul qui est la principale raison d'être de Maple.

Il est très tentant ici d'appeler *valeur* d'un symbole affecté le but du chaînage correspondant, mais, dans un premier temps, *il ne faut surtout pas le faire*. On verra que dans un environnement donné, la valeur d'un symbole, presque toujours bien définie, *n'est pas* en général le but du chaînage de source ce symbole. On donne un exemple un peu plus loin. Il faut dire, si `[symbol → object]` est un chaînage, que le symbole `symbol` *pointe* vers l'objet `object`.

Voir un chaînage.

Le lecteur souhaite probablement voir un chaînage de l'environnement initial. Pour voir le chaînage éventuel de source le symbole `symb`, il faut exécuter «`eval(symb, 1)`». Par exemple la mise en évidence du chaînage `[x → 34]` installé ci-dessus peut être obtenue par :

```
> eval(x, 1) ;
```

34

Si, au début d'une session Maple vous voulez voir par exemple le chaînage de source le symbole `sin`, procédez de même.

```
> eval(sin, 1) ;
```

readlib('sin')

La procédure `sin` sait calculer le sinus de nombres réels ou complexes. L'*environnement initial* contient ce qu'il faut pour faire savoir à Maple, le cas échéant, que la procédure `sin` peut être obtenue par chargement à partir de la bibliothèque standard. Vous êtes peut-être tenté par un exemple plus simple, consistant à «voir» par exemple la valeur de π dans l'environnement initial, mais vous allez être déçu :

```
> eval(Pi, 1) ;
```

π

Quand un symbole, d'après «`eval(..., 1)`», pointe vers lui-même, c'est qu'en fait ce symbole est *absent de l'environnement*, et c'est le cas du symbole `Pi` si important pour les mathématiciens ; noter au passage qu'on trouve ici aussi une situation où l'*unique* objet interne, le symbole de nom `Pi`, a une forme externe input `Pi` assez différente de la forme externe output choisie par l'Afficheur : π. Mais où donc par exemple est localisée l'information que π est peu différent de 3.14 ? On l'obtient par la procédure `evalf` :

```
> evalf(Pi, 4) ;
```

3.142

C'est donc le symbole `evalf` et non pas le symbole `Pi` qui est présent dans l'environnement initial de telle façon que l'information $\pi \sim 3.142$ soit accessible ; le deuxième argument `4` de `evalf` demandait une approximation à quatre chiffres décimaux. Bien noter que le fait que `Pi` soit *absent* de l'environnement ne signifie pas qu'il n'y soit pas *référencé*. C'est un point de terminologie important : un symbole est *présent* dans un environnement si et seulement s'il est la *source* d'un chaînage. On peut par exemple modifier le chaînage de source `x` par :

```
> x := Pi ;
```

$x := \pi$

de telle sorte que `x` pointe vers le symbole `Pi` ; le symbole `Pi` est donc désormais référencé, mais il reste absent de l'environnement. Le symbole `Pi` est *libre*, ce qui est un peu étrange par rapport à son statut mathématique ; ce sont les procédures utilisatrices de `Pi` qui sont «au parfum» et sauront faire les calculs quelquefois difficiles que vous attendez. Vous pouvez aussi être tenté par :

```
> Pi := 355/113 ;
Error, attempting to assign to `Pi` which is protected
```

On voit que `Pi` est un symbole mieux que libre, il est *protégé* ; en particulier vous ne pouvez pas le rendre présent dans l'environnement[8].

[8] À moins de lever la protection (`unprotect`), mais, sauf cas très spécial, c'est franchement déconseillé !

Un symbole qui ne pointe pas vers sa valeur.

Examinons maintenant une situation où la valeur d'un symbole n'est pas l'objet pointé par ce symbole.

```
> a := b ;
```
$$a := b$$
```
> b := c ;
```
$$b := c$$

Deux chaînages `[a` $\rightarrow$ `b]` et `[b` $\rightarrow$ `c]` sont maintenant installés. Regardons l'objet pointé par `a`, puis la *valeur* de `a` :

```
> eval(a, 1) ;
```
$$b$$
```
> a ;
```
$$c$$

On voit que la *valeur* `c` du symbole `a`, autrement dit le résultat de l'*évaluation* du symbole `a` n'est pas l'objet pointé par `a`. Le fonctionnement de l'évaluateur Maple est véritablement la pièce maîtresse du logiciel ; les *règles* de fonctionnement de l'évaluateur seront réunies dans des paragraphes spéciaux intitulés «**Eval**».

Eval 7 — *La* valeur *d'un symbole*[9] *est définie comme suit :*

1. *Si le symbole est* libre*, c'est le symbole lui-même (évaluation triviale) ;*
2. *Si le symbole est* présent *dans l'environnement, le but du chaînage correspondant est évalué et le résultat de cette évaluation sera la valeur du symbole dans cet environnement (post-évaluation).*

Voir cependant l'exception Eval-16.

C'est la première fois que la nature *récursive* de l'évaluateur est observée : si le symbole est présent, *sa valeur est... la valeur* de l'objet pointé. En conséquence, un objet pointé par un symbole peut ne pas être la valeur ce ce symbole ; il en sera ainsi si l'objet pointé est *à évaluation non triviale*. C'était le cas pour le symbole `a` plus haut : l'objet pointé `b` avait une valeur non triviale `c`. Si on ajoute maintenant :

```
> c := d ;
```
$$c := d$$
```
> a ;
```
$$d$$

et ainsi de suite.

Warning 8 — *Si la* valeur *du symbole* `symb` *est l'objet* `obj` *différent de* `symb`*, ceci* n'implique pas *que le symbole* `symb` *pointe vers l'objet* `obj`*. La seule conclusion correcte est la suivante : la* valeur *de l'objet* `obj2` *pointé par* `symb` *est* `obj`*.*

[9] On verra Chapitre 6 qu'il y a une règle d'évaluation particulière pour les paramètres de procédures et leurs symboles locaux.

Et si on introduit un *cycle* dans l'ensemble des chaînages ? Le cycle le plus simple serait celui d'un symbole pointant vers lui-même, mais ceci est impossible. Si on demande qu'un symbole pointe vers lui-même, Maple comprend qu'on veut en fait le libérer ! Dans ces jeux (?) de chaînage, il peut être important de revenir à l'environnement initial, ce qui est obtenu par la commande :

```
> restart ;
```

Ceci restaure l'environnement initial et en particulier détruit les chaînages précédemment installés par l'utilisateur. Ne pas s'étonner de l'absence apparente d'output ; il y en a bien un mais c'est l'objet vide, dont l'affichage est aussi vide ! Ce point sera détaillé plus tard. Si vous demandez maintenant :

```
> a := a ;
```

$$a := a$$

l'environnement *n'est pas modifié* ; c'est une convention : demander qu'un symbole pointe sur lui-même, c'est demander *conventionnellement* de le libérer[10]. S'il était déjà libre, l'environnement n'est donc pas modifié. Prenons maintenant au contraire le cas où le symbole `a` serait présent :

```
> a := b ;
```

$$a := b$$

Comment libérer `a` ? Ceci ne fonctionne pas :

```
> a := a ;
```

$$a := b$$

```
> a ;
```

$$b$$

Vous avez ici la surprise de voir qu'après avoir demandé (input) que `a` pointe vers `a`, Maple décide de le faire pointer... vers `b`, comme le montre l'output ! Que comprendre ? Il y a ici un autre point de récursivité très important.

Eval 9 — *Une instruction d'*affectation «`symb := obj`» *est exécutée en deux temps :*

1. *L'objet* `obj` *est évalué et, si cette évaluation termine, elle retourne un objet* `obj2` *;*
2. *Un chaînage* `[symb → obj2]` *est installé dans l'environnement. Si le symbole* `symb` *était déjà présent dans l'environnement, le chaînage antérieur est retiré.*

C'est pourquoi, dans la commande qui semblait si innocente «`a := a`», Maple a d'abord évalué le second membre `a` et l'évaluateur a retourné `b` ; Maple décide donc d'affecter `b` à `a` comme le montre ce qui est indiqué par l'afficheur ; en définitive la situation reste inchangée : l'ancien chaînage est détruit, mais le même est

[10] Voir aussi les procédures `unassign` et `assigned` expliquées Section 2.6.

aussitôt réinstallé ! D'une façon générale, une instruction «`symb := symb`» modifie l'environnement comme suit : le chaînage de source `symb` est redéfini pour pointer vers la *valeur* actuelle de `symb`. L'expérience ad hoc est la suivante.

```
> restart ;
  a := b ; b := c ; c:= d ;
```
$$a := b$$
$$b := c$$
$$c := d$$
```
> eval(a,1) ;
  a ;
```
$$b$$
$$d$$
```
> a := a ;
```
$$a := d$$
```
> eval(a,1);
  a ;
```
$$d$$
$$d$$

Bien, mais comment alors affecter `a` à `a`, autrement dit comment retirer de l'environnement le chaînage de source `a` ? Il faut *quoter* le membre de droite de l'instruction d'affectation, pour empêcher son évaluation.

Eval 10 *L'évaluation d'une expression* quotée `'expr'` *retourne l'objet* `expr` *non évalué, quotes retirés.*

```
> a := 'a' ;
```
$$a := a$$
```
> a ;
```
$$a$$

Il est donc, malgré les apparences, impossible d'installer un chaînage `[a` $\rightarrow$ `a]`. Mais peut-on installer deux chaînages `[a` $\rightarrow$ `b]` et `[b` $\rightarrow$ `a]` ? Oui, mais pas comme ça :

```
> a := b ; b := a ;
```
$$a := b$$
$$b := b$$

Car dès que `a` pointe vers `b`, l'évaluation de «`b := a`» demande en fait de faire pointer `b` vers la *valeur* de `a`, à savoir `b`, et en définitive `b` reste libre. Il faut donc faire :

```
> restart ;
> a := b ; b := 'a' ;
```
$$a := b$$
$$b := a$$

Et cette fois on a bien installé un *cycle d'ordre 2* dans les chaînages. On peut dans ces conditions, compte tenu de la récursivité de l'évaluation des symboles (cf. Eval-7), être assez perplexe sur le résultat de l'évaluation de `a` (ou `b`). Maple aussi :

```
> a ;
Error, too many levels of recursion
```

C'est notre premier exemple d'une évaluation qui ne termine pas. Les deux chaînages «réciproques» `[a` $\rightarrow$ `b]` et `[b` $\rightarrow$ `a]` provoquent une récursivité sans terminaison de l'évaluateur. Le «résultat» obtenu dépend du système sous-jacent ; le message d'erreur affiché ici est obtenu sous Windows ; sur certains systèmes Unix, aucun message d'erreur n'est affiché ; de plus l'évaluation indéfinie ne peut même pas être interrompue de l'intérieur de Maple par le bouton stop ; la seule «solution» est l'interruption extérieure, par exemple par la commande Unix `kill` émise par un autre process ; mais la session Maple est perdue !

Les évaluations ne permutent pas toujours.

Une autre conséquence des règles d'évaluation Eval-7 et Eval-9 est la dissymétrie surprenante pour les non-initiés entre les deux séquences d'instruction suivantes.

```
> restart ; a := b ; b := c ; b := d ;
  a, eval(a, 1) ;
```

$$a := b$$
$$b := c$$
$$b := d$$
$$d, b$$

```
> restart ; b := c ; a := b ; b := d ;
  a, eval(a, 1) ;
```

$$b := c$$
$$a := c$$
$$b := d$$
$$c, c$$

On voit que l'*ordre* des affectations est important. Dans le premier cas, `b` est affecté à `a` *avant que* `c` soit affecté à `b` ; compte tenu de l'affectation de `d` à `b`, il en résulte que la *valeur* de `a` est `d`, différente de l'objet pointé par `a`. Dans le second cas, les deux premières affectations sont échangées ; l'affectation demandée «`a := b`» affecte donc la *valeur* de `b`, soit `c`, à `a` ; le nouveau chaînage installé pour `b` ne modifie donc pas la valeur de `a`.

Le lecteur est normalement troublé par les virgules séparant `a` de `eval(a, 1)`, alors qu'on devrait séparer par un point-virgule les deux instructions demandant, la première, l'évaluation de `a`, la seconde, l'évaluation de `eval(a, 1)`. On verra par la suite qu'en fait le texte «`a, eval(a, 1)`» est la forme externe d'un *unique* objet Maple à deux *composants* ; en Maple, la virgule *n'est pas* un séparateur, c'est un *opérateur* constructeur de *suites*. Ces notions seront détaillées plus loin

Chapitre 4. Pour le moment, vous pouvez considérer qu'est ainsi disponible une méthode permettant d'afficher deux *objets* Maple ou plus sur une seule ligne. Par contre vous ne pouvez faire de même avec deux instructions d'affectation, car ces instructions *ne sont pas* des objets Maple.

```
> a := 1 , a := 2 ;
':=' unexpected
```

2.3.3 Bugs possibles.

Les mécanismes d'évaluation de Maple sont puissants, mais provoquent souvent des bugs difficiles pour les novices. Compte tenu de ce qui vient d'être expliqué sur l'évaluation des symboles, on préfère anticiper un peu et expliquer immédiatement deux bugs fréquents directement liés à ces questions. Ils proviennent de symboles que l'utilisateur *croit* libres et qui en fait ne le sont pas.

Variables prétendues libres.

On a déjà vu, on donnera les détails plus tard, que les intégrations raisonnables peuvent être tentées à l'aide de la procédure `int`. Par exemple :

```
> int(exp(-x^2)/(1+x^2), x=-infinity..infinity) ;
```

$$\pi\,\mathbf{e} - \pi\,\mathbf{e}\,\mathrm{erf}(1)$$

```
> evalf(%) ;
```

$$1.343293422$$

Il faut en particulier utiliser une variable *libre*, ici `x`, pour faire savoir à Maple par rapport à quelle variable l'intégration doit être effectuée, et aussi, s'il s'agit d'une intégrale définie, quel est l'intervalle d'intégration. Pour Maple, cette «variable» libre est en fait un *symbole*, mais par abus de langage, tenant compte aussi du contexte mathématique, on parle souvent de «variable» libre. Toujours est-il que ce symbole doit être effectivement libre, sous peine d'erreur assez souvent rebelle. Le scenario est habituellement le suivant ; on fait pendant un moment de «petits calculs» et il est alors tentant, quand on souhaite conserver pour un temps une valeur numérique un peu compliquée de l'affecter à un symbole quelconque, pourquoi pas `x`. Par exemple la valeur numérique de la fonction Γ en $1/3$ pourrait être utilisée de façon un peu répétitive et pour éviter de devoir retaper à chaque fois «`evalf(GAMMA(1/3))`», vous pouvez choisir de demander :

```
> x := evalf(GAMMA(1/3)) ;
```

$$2.678938537$$

Le choix du symbole `x` n'est pas merveilleux ; il faut toujours mieux utiliser un symbole avec un minimum de lisibilité, peut-être ici `g3`, mais comme justement on sait que dans dix minutes, ce point sera sans objet, en mal d'inspiration on choisit souvent un symbole très «pauvre», `x` par exemple.

Bien ! Effectivement la session Maple évolue, la question faisant intervenir ponctuellement la quantité $\Gamma(1/3)$ est traitée, on passe à la suite du plan de travail ; les divers sujets se succèdent et quelquefois beaucoup plus tard, mais dans la *même* session, on doit évaluer l'intégrale eulérienne indiquée plus haut.

```
> int(exp(-x^2)/(1+x^2), x=-infinity..infinity) ;
Error, (in int) wrong number (or type) of arguments
```

On est pourtant absolument certain d'avoir déjà traité ce type d'intégrale comme on le voit ici, sans avoir rencontré de problème significatif. Aurait-on oublié quelque chose au sujet de la syntaxe déjà un peu complexe ? On examine patiemment la page d'aide de la procédure `int` ; elle contient beaucoup d'exemples, certains assez impressionnants, mais rien ne permet d'y découvrir une faute quelconque dans notre utilisation. Alors quoi ?

Conseil 11 — *Dans une instruction fautive rebelle, examinez soigneusement le statut des symboles utilisés, notamment de ceux qui devraient être libres.*

Une erreur peut en effet provenir d'un symbole qui normalement devrait être *libre*, mais qui en fait est *présent* dans l'environnement en cours. Les règles d'évaluation impliquent le plus souvent que les occurences de ce symbole sont remplacées par sa valeur, et ceci peut être à l'origine d'une erreur. C'est ce qui se produisait plus haut. L'instruction d'intégration ne peut être réussie que si le symbole `x` est libre ; en fait ici `x` pointe vers $\Gamma(1/3) = 2.678938537$, de sorte que le mécanisme d'évaluation préalable des arguments de `int` demande en fait l'évaluation de :

```
> int(exp(-2.678938537^2)/(1+2.678938537^2), 2.678938537=-infinity..infinity);
```

La variable `x` a disparu, remplacée par sa valeur. La fonction à intégrer est devenue une constante, l'intégrale est maintenant divergente ! L'expression «`symb = ...`» expliquant par rapport à quelle variable il faut intégrer est devenue *non sens* et provoque l'erreur de type.

Variable pilote évaluée trop tôt.

Une variante fréquente de l'erreur montrée dans la section précédente est la suivante. Un opérateur commode qu'on examinera en détail plus loin, l'opérateur «`$`», permet de construire facilement des *suites* plus ou moins compliquées ; par exemple pour construire la *liste* des carrés des entiers de 10 à 20, on peut demander :

```
> SquareList := [i^2 $ i=10..20] ;
```

$$SquareList := [100, 121, 144, 169, 196, 225, 256, 289, 324, 361, 400]$$

Une lecture correcte de l'objet évalué va comme suit : la liste des i^2 pour (`$` = «pour») i variant de 10 à 20. La liste en question est effectivement retournée.

Comme tout langage impératif, Maple propose des instructions itératives. Par exemple pour afficher les valeurs de l'intégrale de $1/(1+x^3)$ entre 0 et les premières puissances de 10, on pourrait demander :

```
> for i from 1 to 5 do
    evalf(int(1/(1+x^3), x = 0..10^i))
    od ;
```

1.204201575
1.209149577
1.209199076
1.209199571
1.209199574

Mais si, ceci fait, vous demandez *maintenant* la construction de `SquareList` comme ceci avait pourtant été réussi auparavant :

```
 > SquareList := [i^2 $ i=10..20] ;
Error, wrong number (or type) of parameters in function $
```

vous obtenez une erreur étrange. C'est le moment de penser à Conseil-11 ; le seul symbole suspect est `i`, et il faut donc l'examiner :

```
> i
```

6

On voit que le symbole `i` a la valeur `6` ! En effet l'instruction itérative `for` a demandé de faire varier `i` de 1 à 5 ; le processus a été interrompu quand `i` avait atteint la valeur `6`, mais le chaînage `[i → 6]` est resté présent[11] dans l'environnement. Donc, compte tenu du mécanisme d'évaluation, l'instruction d'affectation demandée équivaut à :

```
> SquareList := [6^2 $ 6=10..20] ;
Error, wrong number (or type) of parameters in function $
```

qui est incohérente, car ce qu'on appellera plus tard la *variable pilote* pour la procédure «`$`» est ici l'entier `6`, mais un entier n'est pas une variable ! Si on empêche (quotes) l'évaluation de `i` :

```
> SquareList := [i^2 $ 'i'=10..20] ;
```

$$SquareList := [36, 36, 36, 36, 36, 36, 36, 36, 36, 36, 36]$$

cette fois la procédure «`$`» peut travailler, mais l'évaluation prématurée du premier terme `i^2` ne donne pas le résultat escompté ! Pour obtenir le bon résultat, il faut aussi quoter le premier `i` :

```
> SquareList := ['i'^2 $ 'i'=10..20] ;
```

$$SquareList := [100, 121, 144, 169, 196, 225, 256, 289, 324, 361, 400]$$

Conseil 12 — *En présence d'une erreur rebelle, réfléchir si une évaluation imprévue n'a pas impliqué l'invocation d'une procédure avec des arguments incorrects.*

[11] Contrairement à ce qui est maintenant standard, par exemple dans les langages C, Java et Lisp.

2.4 Évaluateur récursif.

Si on a lu soigneusement la section précédente, en répétant patiemment les exemples prévus sur son poste de travail, on commence à avoir une idée intuitive du fonctionnement de l'évaluateur Maple. Deux principes généraux sont applicables.

2.4.1 Post-évaluation.

Eval 13 — *Quand un résultat d'évaluation est obtenu, Maple examine si ce résultat est à son tour* non trivialement *évaluable. Si la réponse est négative, le processus d'évaluation est terminé. Si au contraire la réponse est positive, le résultat de l'évaluation sera en définitive le résultat de l'évaluation du résultat précédemment obtenu.*

Il est clair que dans la deuxième hypothèse aucune garantie ne peut être obtenue sur la terminaison du processus d'évaluation. L'exemple des deux chaînages réciproques `[a → b]` et `[b → a]` a montré qu'il peut effectivement arriver que l'évaluation ne termine pas. Pour illustrer la portée plus générale de Eval-13, considérons la situation suivante. On fait pointer le symbole `a` vers la liste des deux symboles `b1` et `c1` ;

```
> restart ;
> a := [b1, c1] ;
```

$$a := [b1, c1]$$

Ensuite on crée deux suites de chaînages `[b1 -> b2 -> b3]` et `[c1 -> c2 -> c3 -> c4]`. On verra page ▙▟ les détails concernant l'opérateur «`||`» (concaténation[12]) ; les affectations résultantes sont affichées.

```
> for i from 1 to 2 do b||i := b||(i+1) od ;
```

$$b1 := b2$$
$$b2 := b3$$

```
> for i from 1 to 3 do c||i := c||(i+1) od ;
```

$$c1 := c2$$
$$c2 := c3$$
$$c3 := c4$$

Ceci étant fait, grâce aux procédures très particulières `seq` et `eval`, détails plus tard, on peut afficher les phases *intermédiaires* du processus d'évaluation. On affiche ensuite le résultat de l'évaluation *banale* de `a` :

[12] Jusqu'à Maple 5, l'opérateur de concaténation était l'opérateur `dot`, représenté par un *point* (‘.’) et non pas par la *double barre* (‘||’).

```
> seq(eval(a, i), i=1..5) ;
```

$$[b1, c1], [b2, c2], [b3, c3], [b3, c4], [b3, c4]$$

```
> a
```

$$[b3, c4]$$

Il a donc fallu quatre phases d'évaluation pour aboutir au résultat définitif. Si on ajoute un chaînage `[c4 → c2]`, il est clair que l'évaluation ne termine plus :

```
> c4 := 'c2' ;
```

$$c4 := c2$$

```
> a ;
Error, too many levels of recursion
```

2.4.2 Pré-évaluation.

La règle Eval-13 indique qu'une évaluation *a posteriori* est toujours tentée. La règle d'évaluation suivante demande au contraire une pré-évaluation.

Eval 14 — *Quand un objet* function *:*

`obj0(obj1, obj2, ..., objn)`

doit être évalué, l'opérateur `obj0` *et les opérandes*[13] `obj1`*, ...,* `objn` *sont évalués séquentiellement avant d'être utilisés. L'évaluation de* `obj0` *est de type* explicite[14] *alors que l'évaluation des autres arguments est de type* implicite*.*

La règle Eval-14 a de très nombreuses exceptions, et son interprétation précise est assez délicate, surtout quand il s'agit du traitement de l'opérateur `obj0`. Si la valeur procédure de `obj0` est une procédure *non standard*, le traitement des opérandes peut être très différent.

Eval 15 — *Une procédure est* standard *si la règle Eval-14 s'applique* strictement *quand elle est invoquée.*

On se contentera sagement dans un premier temps d'illustrer l'esprit général de la règle Eval-14, ce qui suffira pour commencer. Plus tard, au fur et à mesure qu'on avancera dans la connaissance de Maple, cette règle sera à plusieurs reprises complétée, quelquefois pour examiner son application dans certaines situations exceptionnelles, d'autres fois pour en préciser l'interprétation à propos de tel ou tel détail, d'autres fois encore ce qui advient quand une procédure est non standard.

Considérons les procédures mathématiques suivantes. La procédure `Psi` calcule la $(n+1)$-ième dérivée logarithmique de la fonction Γ : $\Psi(n, x) = \Psi^{(n)}(x)$ où $\Psi(x) = \Gamma'(x)/\Gamma(x)$:

[13] Appelés aussi *arguments*.
[14] Voir à ce propos Eval-16.

```
> Psi(2, 3.4) ;
```

$$-.1155897239$$

La procédure `round` arrondit un flottant à l'entier le plus proche :

```
> round(3.8) ;
```

$$4$$

Considérons enfin l'évaluation de l'expression suivante :

```
> Psi(round(tan(2.5)), sin(5.1)) ;
```

$$2.639819868 - 3.141592654\,I$$

Le résultat est un complexe dont la partie imaginaire semble $-\pi i$.

La règle Eval-14 explique que l'expression à évaluer étant un objet **function**, cet objet est *déstructuré* en sous-objets. Ici, trois composants numérotés de 0 à 2. Le composant `obj0` est le symbole `Psi` ; le symbole `Psi` est d'abord évalué, et Maple trouve comme résultat un objet **procedure** ; on peut le voir comme suit :

```
> eval(Psi) ;
```

proc . . . **end**

Raisonnablement, Maple préfère n'afficher que sous une forme très simplifiée l'objet **procedure** valeur, vous faisant grâce du texte source fort complexe. L'évaluateur observe que cette procédure est *standard*[15] et qu'elle est invoquée avec deux arguments ; le premier, appelé `obj1` dans la règle Eval-14 est l'expression `round(tan(2.5))`, le second, `obj2`, est l'expression `sin(5.1)`. La terminologie suivante sera systématiquement utilisée dans la suite, elle est particulièrement commode pour décrire le fonctionnement de la règle Eval-14 dans un cas particulier. L'expression `round(tan(2.5))` est le premier argument *provisoire* ; de la même façon, l'expression `sin(5.1)` est le deuxième et dernier argument *provisoire*. La règle Eval-14 explique que ces arguments doivent être *évalués avant d'être utilisés*. Les résultats de ces évaluations seront les arguments *définitifs*, ceux avec lesquels la procédure `Psi` va effectivement travailler.

C'est à ce point que l'évaluateur est *récursif*. L'*exemplaire* de l'évaluateur qui était en activité pour traiter l'évaluation principale est mis provisoirement au repos. Un deuxième exemplaire de l'évaluateur est invoqué pour effectuer l'évaluation nécessaire de notre argument provisoire `round(tan(2.5))`. Pour cette invocation, de nouveaux ingrédients sont identifiés ; l'opérande `obj0` est cette fois le symbole `round`, l'unique argument `obj1` est l'expression `tan(2.5)`. Cet argument, provisoire, doit aussi être évalué avant d'être utilisé, etc. Le schéma complet de l'évaluation est donc le suivant :

```
Eval1(Psi(round(tan(2.5)), sin(5.1)))
  obj0p = Psi
```

[15] Dans de nombreux cas qui seront examinés plus tard, la procédure est non-standard et les arguments ne sont pas traités de la même façon.

```
    Eval2(Psi)
    Eval2 -> procedure(Psi)
  obj0d = procedure(Psi)
  obj1p = round(tan(2.5))
    Eval2(round(tan(2.5)))
      obj0p = round
        Eval3(round)
        Eval3 -> procedure(round)
      obj0d = procedure(round)
      obj1p = tan(2.5)
        Eval3(tan(2.5))
          obj0p = tan
            Eval4(tan)
            Eval4 -> procedure(tan)
          obj0d = procedure(tan)
          obj1p = 2.5
            Eval4(2.5)
            Eval4 -> 2.5
          obj1d = 2.5
          procedure(tan) works with 2.5
        Eval3 -> -.7470222972
      obj1d = -.7470222972
      procedure(round) works with -.7470222972
    Eval2 -> -1
  obj1d = -1
  obj2p = sin(5.1)
    Eval2(sin(5.1))
      obj0p = sin
        Eval3(sin)
        Eval3 -> procedure(sin)
      obj0d = procedure(sin)
      obj1p = 5.1
        Eval3(5.1)
        Eval3 -> 5.1
      obj1d = 5.1
      procedure(sin) works with 5.1
    Eval2 -> -.9258146823
  obj2d = -.9258146823
  procedure(Psi) works with -1 and -.9258146823
Eval1 -> 2.639819868+3.141592654*I
```

On a utilisé quelques conventions assez évidentes pour ne pas être envahi par l'affichage de tout ce qui serait en principe nécessaire. Chaque invocation de l'évaluateur correspond à une séquence :

```
   Evaln(expr)
     ...
     ...
   Evaln -> ...
```

où en entrée figure l'expression à évaluer et en sortie l'objet *retourné* par l'évaluation. Quand cette invocation est active, l'invocation précédente est mise au repos. Une indication comme «`obj1p = ...`» indique qu'un composant avant évaluation

est identifié, «p» pour *provisoire* ; symétriquement une indication «obj1d = ...» indique le même composant *après* évaluation, «d» pour *définitif*. Enfin une indication comme procedure(sin) note de façon symbolique l'objet **procedure** associé au symbole sin, en général très complexe.

On voit que le mécanisme d'évaluation récursive d'un objet **function** suit une chronologie très précise et relativement complexe. L'objet à évaluer est d'abord déstructuré pour obtenir les différents composants. Le plus souvent, ces composants sont évalués avant d'être utilisés, auquel cas le mécanisme en cours de description s'applique à son tour, récursivement, aux composants. Quand les versions «définitives» des composants sont obtenues, la procédure valeur du composant obj0 travaille enfin avec les arguments définitifs.

Il peut se faire que la valeur de obj0 ne soit pas une procédure mais par exemple un symbole. Le mécanisme sera exactement le même, à ceci près que la dernière étape *travail de la procédure* est omise.

```
> restart ;
> f:= g ; g := h ; f(round(tan(2.5)), sin(5.1)) ;
```

$$f := g$$
$$g := h$$
$$\mathrm{h}(-1, -.9258146823)$$

Bref, en parlant simplement, Maple «évalue ce qu'il peut».

Au fait, pour quelle raison l'évaluation de notre expression $\Psi(\mathrm{round}(\tan(2.5)), \sin(5.1))$ retournait-elle un nombre complexe ? Comme l'a montré l'analyse de l'évaluation, il s'agissait en fait de $\Psi(-1, -.9258146823)$; Maple devait donc considérer la dérivée d'ordre -1 de la dérivée de $\ln(\Gamma)$; la dérivée d'ordre -1 est logiquement la primitive, et le résultat doit donc être $\ln(\Gamma(-.9258146823))$; mais la valeur de la fonction Γ en ce point est négative et la détermination principale du logarithme d'un nombre négatif a, au moins pour Maple, $-\pi i$ pour partie imaginaire.

2.4.3 Une première exception.

Le lecteur perspicace a en principe été troublé dans la section précédente quand on montrait la valeur procédure du symbole Psi. Il était indiqué d'évaluer eval(Psi) ; pourquoi pas seulement Psi ? Considérons ces trois possibilités :

```
> Psi
```

$$Psi$$

```
> eval(Psi) ;
```

proc ... **end**

```
> eval(Psi, 1) ;
```

proc ... **end**

On voit grâce au dernier essai que le chaînage du symbole Psi vers la procédure prédéfinie associée est bien là, mais l'évaluation banale de Psi semble ne pas tenir

compte de ce chaînage.

Eval 16 — *Si un symbole pointe vers une procédure ou une table, l'évaluation* implicite *de ce symbole retourne seulement le symbole initial ; l'évaluation* explicite *retourne la procédure ou la table.*

Par défaut, l'évaluation d'un symbole est implicite ; elle est explicite dans deux circonstances :

1. *L'évaluation est demandée par l'intermédiaire de la procédure* `eval` *;*
2. *Le symbole est évalué en position* opérateur *d'une expression fonction ou objet indexé.*

On verra plus tard le cas des objets indexés. L'exemple complètement artificiel suivant illustre le cas des objets **function** :

```
> f1 := proc (arg1)
    RETURN(arg1)
    end :
> f1(35) ;
```

$$35$$

```
> f2 := proc (arg2)
    RETURN(2*arg2)
    end :
> f2 ;
```

$$f2$$

```
> eval(f1) ;
```

$$\textbf{proc}(arg1)\ \text{RETURN}(arg1)\ \textbf{end}$$

```
> f1(f2) ;
```

$$f2$$

```
> % ;
```

$$f2$$

```
> %(exemple) ;
```

$$2\,exemple$$

Les deux procédures affectées à `f1` et `f2` sont analogues ; elles retournent respectivement l'unique argument passé ou son double. On voit que l'évaluation de `f1(35)` a bien fait intervenir la procédure, car le symbole `f1` est en position *opérateur*. Par contre l'évaluation de `f2` seul est *implicite* et la procédure but n'est pas retournée. Dans l'évaluation de `f1(f2)`, l'évaluation de `f1` est explicite, alors que celle de `f2` est implicite, d'où le résultat. Le pseudo-symbole `%` (objet «ditto») chaîne toujours vers le dernier résultat obtenu, mais l'évaluation est implicite et l'évaluation s'arrête donc à `f2`. Enfin dans la dernière expression, puisque `%` est en position opérateur, l'évaluation est explicite et la procédure valeur de `f2` va travailler, d'où le résultat.

Un tel exemple est complètement artificiel et a pour seul intérêt de mettre en évidence le mécanisme de fonctionnement de la règle Eval-16. Il illustre aussi comment justement concevoir des exemples artificiels très simples permettant de dévisser le fonctionnement de l'évaluateur.

2.5 D'autres objets de type function.

Contrairement à Lisp, Maple a sacrifié aux usages remontant aux langages préhistoriques style Fortran, où on considère comme insupportable de demander que toutes les expressions soient préfixées. L'expression qu'on écrit en Fortran, Ada, C, Java ou Maple sous forme *infixée* : «`(a + b) * (c + d)`» s'écrit en Lisp sous forme *préfixée* «`(* (+ a b) (+ c d))`». Les éditeurs modernes savent faire clignoter la parenthèse ouvrante correspondant à la parenthèse fermante où le curseur est positionné ; ce n'est malheureusement pas le cas de l'éditeur intégré de Maple. Quand un éditeur «moderne» est utilisé, l'expérience montre que la notation systématiquement préfixée est bien supérieure à la notation traditionnelle des mathématiciens, celle qui semblait si commode aux programmeurs Fortran. Mais ce n'est pas le sujet du jour.

Cette digression n'était là que pour expliquer au lecteur que bien des objets Maple qui ne semblent pas être des objets **function** quand on les entre au clavier de la façon usuelle en fait sont bel et bien des objets **function**. On peut savoir si un objet est d'un type donné à l'aide de la procédure `type`. Il faut prévoir deux arguments, l'objet dont le type ext examiné et le descripteur de type, un symbole dans les cas les plus simples.

```
> type(3.4, realcons), type(3.4, integer) ;
```

true, *false*

Attention, la procédure `type` est standard et évalue donc ses arguments avant de les utiliser ; si donc vous essayez :

```
> type(Psi(round(tan(2.5)), sin(5.1)), function) ;
```

false

pour «voir» si l'expression étudiée dans la section précédente est bien un objet **function**, la réponse est négative, car l'expression est *d'abord* évaluée ; le résultat est un nombre complexe et un nombre complexe n'est pas un objet **function**. Il faut comme toujours quoter pour empêcher l'évaluation.

```
> type('Psi(round(tan(2.5)), sin(5.1))', function) ;
```

true

Noter qu'un objet peut être de plusieurs types différents. Chaque type est un ensemble d'objets Maple et en général ces ensembles ont des intersections non triviales ; par exemple un entier (type **integer**) est aussi un réel (type **realcons**) :

```
> type(3, integer), type(3, realcons) ;
```

true, *true*

Il a été expliqué Section 2.3.3 comment construire une *liste* de carrés :

```
> SquareList := [i^2 $ i=10..20] ;
```

$$SquareList := [100, 121, 144, 169, 196, 225, 256, 289, 324, 361, 400]$$

Il est légal de présenter la même expression sans les crochets ouvrant et fermant, auquel cas on construit une *suite* :

```
> SquareSequence := i^2 $ i=10..20 ;
```

$$SquareSequence := 100, 121, 144, 169, 196, 225, 256, 289, 324, 361, 400$$

Les détails sur ces types itératifs très importants seront étudiés Chapitre 4. Il avait été expliqué Section 2.3.3 pourquoi les mécanismes de préévaluation pouvaient compliquer la vie si le symbole `i` était présent dans l'environnement. La similitude avec ce qui est énoncé dans la règle Eval-14 suggère que l'expression ci-dessus construite à partir de l'opérateur binaire «`$`» est aussi un objet **function**. C'est bien le cas :

```
> type('i^2 $ i=10..20', function) ;
```

true

On peut d'ailleurs présenter cette expression d'une façon plus «fonctionnelle» ; les backquotes (accents graves) encadrant le `$` initial sont indispensables :

```
> `$`(i^2, i=10..20) ;
```

$$100, 121, 144, 169, 196, 225, 256, 289, 324, 361, 400$$

La forme *interne* des deux objets présentés à Maple, forme infixée ou forme préfixée, est la même. C'est le Lecteur de Maple qui fait le travail nécessaire pour reconnaître l'une quelconque des deux formes et construire de toute façon sous forme fonctionnelle l'objet décrit. Pour en avoir le cœur net, comparons les deux objets :

```
> evalb(''`$`(i^2, i=10..20)' = 'i^2 $ i=10..20') ;
```

true

La procédure `evalb` travaille sur *un* argument de type **equation** et en compare les deux membres. Mais à propos, pourquoi les *backquotes*, en français les accents graves[16], encadrant le caractère `$` quand on présente l'expression sous forme fonctionnelle ? D'une part on veut le *symbole* `$` permettant d'atteindre la procédure appropriée ; d'autre part les symboles banals sont constitués de lettres, de chiffres et de «blancs soulignés» «`_`», le symbole ne devant toutefois pas commencer par un chiffre. Donc en principe l'unique caractère d'un symbole ne peut pas être «`$`». Mais toute chaîne de caractères devient un symbole si elle est encadrée par des *back*quotes :

[16] La police télétype utilisée par LaTeX est assez décevante de ce point de vue : il faut être très attentif pour distinguer les accents aigus des accents graves, autrement dit les quotes des backquotes.

```
> type(‘This & is # really [ a @ symbol‘, symbol) ;
```
true

Le symbole vide sert en particulier à d'innombrables tours de passe-passe dans les affichages, car les backquotes nécessaires à l'entrée ne sont pas affichés en sortie :

```
> type(‘‘, symbol) ;
```
true
```
> ‘‘ := ‘I’m not visible‘ ;
```
:= *I'm not visible*

Il est tentant de faire l'expérience «fonctionnelle» avec une expression somme, mais surprise :

```
> evalb(’‘+‘(a, b)’ = ’a + b’) ;
```
false

Les *deux* possibilités existent, produisent presque toujours le même résultat, mais les formes internes sont cette fois différentes ; l'opérateur Maple «+» infixé est très particulier et en particulier l'objet correspondant n'est pas un objet **function** ; comparer les deux résultats :

```
> type(’a + b’, function) ;
```
false
```
> type(’‘+‘(a, b)’, function) ;
```
true

Ces considérations ne sont pas là du tout pour vous apprendre les différences éventuelles plutôt subtiles, quand elles existent, entre les formes infixées et les formes préfixées ; il est rarissime que ces détails soient de quelque utilité. Il s'agit seulement d'explorer un peu notre espace de travail, de *fouiner*, comme c'est la règle dans tous les grands logiciels. Seul l'utilisateur connaissant un peu la nature générale de son espace de travail peut espérer y devenir un jour raisonnablement autonome. Il s'agit dans ce manuel de vous *expliquer* Maple, et surtout de vous rendre capable de trouver vous-même les informations qui peuvent être indispensables dans tel ou tel cas.

2.6 Procédures assigned, assign et unassign.

Cette section peut être omise en première lecture. Trois outils de la même famille y sont décrits, qui permettent d'analyser et de modifier l'environnement ; ils peuvent donc être utilisés pour traiter les chaînages. Dans les usages élémentaires de Maple, ils sont rarement de quelque utilité ; pour des usages un peu plus sophistiqués, ils deviennent vite utiles ou même indispensables ; c'est notamment le cas quand il s'agit d'utiliser les résultats des différents *solvers*, comme `solve`, `dsolve`, `rsolve`, etc.

La procédure `assigned` permet de savoir si un symbole est présent dans l'environnement.

```
> restart ; assigned(a) ;
```
$$false$$
```
> a := 1 ; assigned(a) ;
```
$$a := 1$$
$$true$$
```
> a := 'a' ; assigned(a) ;
```
$$a := a$$
$$false$$

La procédure `assigned` est *non-standard*, car elle n'évalue pas son argument avant de l'utiliser. Quand l'entier `1` est affecté au symbole `a`, le symbole devient présent et la procédure `assigned` permet de le vérifier. On retire ici `a` de l'environnement en utilisant la convention expliquée page 53, par «auto-affectation» du symbole ; on vérifie ensuite, encore avec la procédure `assigned`, que le symbole `a` est bien retiré de l'environnement.

Le caractère non-standard de la procédure `assigned` est mis en évidence par l'expérience suivante.

```
> restart ; a := b ;
```
$$a := b$$
```
> assigned(a), assigned(b) ;
```
$$true, false$$

Après l'affectation `a := b`, le symbole `a` est présent dans l'environnement mais non référencé, le symbole `b` est référencé mais non présent (voir page 51). On voit donc que l'évaluateur *n'a pas* évalué l'argument `a` de `assigned(a)` avant de l'utiliser ; sinon l'argument définitif aurait été `b` qui lui est absent de l'environnement, de sorte que la réponse aurait été négative.

La procédure `unassign` demande la libération de son (ou ses) argument(s). Cette fois au contraire, la procédure est *standard*, ce qui est mis en évidence par l'expérience analogue.

```
> restart ; a := b ; b := 3 ;
```
$$a := b$$
$$b := 3$$
```
> unassign(a) ;
Error, (in assign) invalid arguments
```

Ce type d'erreur, assez fréquente en mode interactif, est assez troublante pour l'utilisateur non initié. La procédure `unassign` est standard et l'argument `a` est donc évalué avant d'être utilisé. Les deux chaînages `[a → b]` et `[b → 3]` présents dans l'environnement impliquent que la valeur actuelle de `a` est l'entier `3`, voir la règle Eval-7. Mais libérer l'entier `3` n'a pas de sens, car la source d'un chaînage ne peut être qu'un symbole, pas un entier. Par ailleurs le message d'erreur cite curieusement

la procédure `assign` et non `unassign` ! Mais la procédure `unassign` en fait appelle à son tour la procédure `assign` avec des arguments ad hoc pour demander la libération souhaitée, et c'est seulement pendant cette invocation souterraine que l'erreur est détectée.

Dans le contexte ci-dessus, si l'utilisateur souhaite la libération du symbole pointé par `a`, pour empêcher l'évaluation de `b` vers 3, il peut donc demander :

```
> unassign('b') ;
```

Noter aussi que la procédure `unassign` semble «sans valeur» ; un nouveau prompt apparaît, mais aucun output n'est affiché. Plus précisément, on verra Chapitre 4 que l'output est la suite *dite* vide, un objet pas du tout «vide», mais dont par contre l'affichage est bien vide !

Dans le cas précédent on a utilisé le fait qu'on *savait* que le symbole `a` pointait vers `b`. À supposer qu'on l'ait oublié, il faudrait évaluer `eval(a, 1)` qui retournerait `b`, surtout pas `a` qui retournerait `3` ! Voir Section 50. Mais si `unassign` est utilisé au sein d'une séquence de programmation, il faut savoir demander la libération du symbole pointé par `a` sans connaître d'avance le nom du symbole en question. C'est justement le moment d'utiliser à bon escient `eval(..., 1)`, comme illustré maintenant.

```
> restart ; a := hidden_b ; hidden_b := 333 ;
                                        a := hidden_b
                                   hidden_b := 333
> unassign(eval(a, 1)) ; a ; hidden_b ;
                                          hidden_b
                                          hidden_b
```

On voit que le symbole `a` est resté présent, son chaînage vers `hidden_b` n'a pas été modifié ; par contre le chaînage de source `hidden_b` a été retiré, ce qui peut aussi être vérifié comme suit.

```
> assigned(a) ; assigned(hidden_b) ;
                                            true
                                            false
```

Si enfin on veut libérer le symbole `a`, vous avez compris que ceci est inopérant :

```
> unassign(a) ; assigned(a) ;
                                            true
```

car l'argument *définitif* est `hidden_b` et c'est donc ce dernier qui est libéré ; il était d'ailleurs déjà libre. Pour libérer le symbole `a`, il faut donc empêcher l'évaluation de `a` comme argument de `unassign` :

```
> unassign('a') ; assigned(a) ; a ;
```

$$false$$

$$a$$

Maple dispose symétriquement d'une procédure `assign` permettant au contraire de réaliser une affectation :

```
> assign(s = a) ; assigned(s) ; assigned(a) ; s ;
```

$$true$$

$$false$$

$$a$$

Bien noter que l'*unique* argument de `assign` a été ici l'objet **equation** «`s = a`», et non pas l'instruction affectation «`s := a`» qui serait de toute façon illégale ; en effet le texte «`s := a`» dénote une *instruction* et non un objet ; or un argument ne peut être qu'un objet. Au contraire le texte `assign(s = a)` *est* un objet et peut par exemple être affecté à un symbole :

```
> restart ; assign(s := a) ;
':=' unexpected
> restart ; obj := 'assign(s = a)' ;
```

$$obj := \text{assign}(s = a)$$

Il a fallu bien sûr quoter le texte en question pour empêcher son évaluation. La vérification de l'affectation ne peut dans un tel cas être faite qu'avec un appel de `eval(..., 1)` :

```
> eval(obj, 1) ;
```

$$\text{assign}(s = a)$$

car si on demande l'évaluation banale de `obj` :

```
> obj ;
```

Ne pas croire que puisque rien n'est affiché, rien n'est fait ! Comme la procédure `unassign`, la procédure `assign` retourne la suite vide ; mais comme la procédure `unassign`, l'invocation de la procédure `assign` modifie l'environnement, cette fois en y modifiant ou en y ajoutant un chaînage. Or ici on a demandé l'évaluation de `obj` qui pointe vers `assign(s = a)` ; l'application de la règle Eval-7 implique donc que l'objet `assign(...)` doit être évalué ; le chaînage `[s` $\rightarrow$ `a]` est donc installé et le résultat de l'évaluation de `obj` est le résultat de l'évaluation de `assign(...)`, donc la suite vide, invisible.

On ne voit pas bien a priori l'utilité de la procédure `assign` par rapport à l'affectation ordinaire «`s := a`» ; mais la procédure `assign` est standard et évalue donc ses arguments, ce qui est source de possibilités variées. Dans une séquence de programmation, il peut arriver par exemple qu'un symbole `s` pointe vers un autre symbole variable d'un moment à l'autre pendant l'exécution ; il peut arriver

qu'une affectation soit souhaitée pour le symbole *pointé* par `s`, ce que ne permet en aucune façon l'affectation ordinaire.

```
> s := a ; s := 3 ;
```

$$s := a$$
$$s := 3$$

```
> s, a ;
```

$$3, a$$

On voit que l'instruction d'affectation `s := 3` a affecté `3` au symbole `s` et non pas au symbole `a` valeur de `s` comme il pourrait l'être souhaité. On aimerait quelque chose comme :

```
> eval(s) := 3 ;
Error, invalid left hand side in assignment
```

mais on voit que cette instruction produit une erreur, car aucune évaluation ne peut intervenir *au niveau supérieur* dans le membre de gauche d'une instruction d'affectation. Il faut, si on veut affecter au symbole *valeur* de `s` par exemple l'entier `3`, utiliser la procédure `assign` :

```
> restart ; s := a ;
```

$$s := a$$

```
> assign(s = 3) ;
> eval(s, 1) ; eval(a, 1) ;
```

$$a$$
$$3$$

```
> assign(s = 33) ;
Error, (in assign) invalid arguments
```

On voit ainsi, et on ne peut le voir que de cette façon, que le chaînage `[s` $\rightarrow$ `a]` n'a pas été modifié, alors que le nouveau chaînage `[a` $\rightarrow$ `3]` a été installé. Quel que soit le symbole pointé par `s`, le résultat analogue aurait été obtenu par la même évaluation, grâce au mécanisme de pré-évaluation des arguments, dont on commence à comprendre l'intérêt. On voit en outre qu'une fois que le chaînage de `a` vers `3` est installé, on ne peut plus évaluer la même instruction `assign`, car cette fois la valeur de `s` est `3`, par l'intermédiaire de `a`, et l'affectation de `33` à `3` n'a pas de sens. Par contre on peut demander :

```
> assign(eval(s, 1) = 33) ;
> eval(s, 1) ; eval(a, 1) ;
```

$$a$$
$$33$$

On espère que le lecteur comprend à quel point ces jongleries exigent une lucidité *parfaite* sur les mécanismes d'évaluation, sous peine d'anarchie complète ! Mais inversement ces mécanismes ne sont quand même pas si compliqués, et si vous faites l'investissement approprié en la matière, le bénéfice est immense pour les travaux un peu significatifs.

Une autre cirsonstance où la procédure `assign` est bien utile, c'est quand la valeur d'*un* symbole est un objet **equation**, autrement dit une équation. Ceci se produit assez fréquemment quand on travaille avec les solvers. Il peut arriver par exemple qu'après avoir résolu une équation, vous obteniez un résultat équivalent au suivant :

```
> restart ; result := solution = exp(Pi^2/12) ;
```

$$result := solution = \mathbf{e}^{\frac{1}{12}\pi^2}$$

Autrement dit la valeur du symbole `result` est maintenant l'équation dont le premier membre est le symbole `solution` et le second membre est l'expression mathématique $e^{\pi^2/12}$. Cette situation où la valeur d'*un* symbole est une équation où le premier membre est à son tour un symbole est en effet très fréquente. Vérifions ce point.

```
> result ;
```

$$solution = \mathbf{e}^{\frac{1}{12}\pi^2}$$

Dans un tel contexte, on souhaite assez souvent en définitive *affecter* au symbole premier membre, ici `solution`, le second membre, ici $e^{\pi^2/12}$. Le mécanisme de pré-évaluation des arguments devient donc de plus en plus intéressant.

```
> assign(result) ;
```

Rien n'est affiché et pourtant l'affectation souhaitée est réalisée :

```
> solution ;
```

$$\mathbf{e}^{\frac{1}{12}\pi^2}$$

Le fait que la procédure `assign` ne retourne rien est assez désagréable, car à moins d'un supplément de votre part, vous ne pouvez pas vérifier que l'affectation souhaitée est réalisée ; or le cadre de travail où des mécanismes d'évaluation assez subtils risquent d'intervenir est assez dangereux.

Conseil 17 — *En mode interactif, après une instruction* `assign`*, vérifiez que l'affectation souhaitée est bien réalisée. Le mieux pour ce faire consiste à demander en plus l'exécution d'une équation :*

```
'symb' = eval(symb, 1) ;
```

Par exemple il aurait été meilleur de demander plus haut :

```
> assign(result) ; 'solution' = eval(solution, 1) ;
```

$$solution = \mathbf{e}^{\frac{1}{12}\pi^2}$$

L'utilisateur est ainsi certain que l'affectation souhaitée est réalisée.

Bien distinguer dans ces questions l'*instruction* d'affectation «:=» de l'*opérateur* équation «=».

-o-o-o-o-o-o-o-

Chapitre 3

Les types de données numériques

3.1 Types, généralités.

3.1.1 Les procédures type et whattype.

On a vu à la fin du dernier chapitre comment déterminer si un objet est d'un type donné, à l'aide de la procédure `type`.

```
> type(2., integer), type(2., float) ;
```

$$false, true$$

On voit que le flottant 2. n'est pas un entier pour Maple. La procédure `type` examine la *nature du codage* qui est utilisé pour l'objet désigné ; sa nature arithmétique au sens mathématique du terme n'est pas considérée. Il aurait été plus précis de dire que `2.` est un objet **float** mais n'est pas un objet **integer**.

Considérons dans le même registre les instructions suivantes :

```
> x := (sqrt(5)-1) * (sqrt(5) +1) ;
```

$$x := (\sqrt{5}-1)(\sqrt{5}+1)$$

```
> type(x, integer) ;
```

$$false$$

La procédure `type` est *standard* et évalue donc ses arguments avant de travailler. L'instruction ci-dessus, ne concerne donc pas le type du *symbole* `x`, mais le type de sa *valeur*. Comparer avec :

```
> y := 2 ; type(y, integer), type('y', integer), type('y', symbol) ;
```

$$y := 2$$

$$true, false, true$$

où cette fois la valeur de `'y'` est le symbole `y`.

Le lecteur lucide sait que la valeur de `x` est, pour le mathématicien, un entier, mais Maple ne le sait pas. Les expressions comprenant des radicaux peuvent être manipulées par des méthodes si variées que Maple préfère par défaut sagement

s'abstenir. De sorte que `x` pointe vers un objet non simplifié, assez complexe, dont il est légitime de demander «le» type :

```
> whattype(x) ;
                                        *
```

La procédure `whattype` ne retourne pas *le* type d'un objet, mais *un* type qu'elle essaie de déterminer au mieux parmi les types possibles pour l'objet en question. En effet un type n'est autre qu'un ensemble d'objets Maple, et ces ensembles ne sont pas en général disjoints. L'ensemble vide est en particulier un type, c'est le type **nothing**. Aucun objet n'est du type **nothing**, pas même le symbole `nothing` :

```
> type(nothing, nothing) ;
                                      false
```

Inversement tout objet est du type **anything**, en particulier le symbole `nothing` :

```
> type(nothing, anything) ;
                                      true
```

La procédure `whattype` pourrait donc répondre `anything` pour tout objet, mais son utilité ne serait pas évidente. On ne peut pas lui demander non plus de retourner le plus petit type contenant l'objet, car il existe toujours un type contenant seulement cet élément. Par exemple l'objet `sqrt(2)` est l'unique élément du type `identical(sqrt(2))` :

```
> type(sqrt(2), identical(sqrt(2))) ;
                                      true
> type(sqrt(3), identical(sqrt(2))) ;
                                      false
```

Mais que choisit `whattype` pour `sqrt(2)` ?

```
> whattype(sqrt(2)) ;
                                        ^
```

Il faut une bonne vue pour voir que le type choisi est désigné par le symbole accent circonflexe «^» ; si on veut en avoir le cœur net :

```
> type(sqrt(2), `^`) ;
                                      true
```

Le symbole désignant le type est exotique et doit donc être encadré par des *back-quotes* (accents graves) pour être entré comme tel, sinon :

```
> type(sqrt(2), |^) ;
`^` unexpected
```

Chaque fois qu'un objet est structuré, son codage commence par indiquer un

opérateur dominant, et la procédure `whattype` retourne cet opérateur[1]. Par exemple $\sqrt{2}$ est codé comme `'^'(2, 1/2)` = `2^(1/2)` = $2^{1/2}$ et l'opérateur maître est l'exponentiation, ce qui explique la réponse de `whattype`. De la même façon, notre nombre `x` est codé essentiellement comme `'*'((`$\sqrt{5}-1$`), (`$\sqrt{5}+1$`))` et l'opérateur maître est l'opérateur de multiplication «*», d'où la réponse qui nous était donnée pour le type de `x` par `whattype`.

3.1.2 Constructeurs de types.

L'opérateur `Not` «construit» le type complémentaire du type argument.

```
> type(2, Not(float)) ;
```

true

L'interprétation précédente, strictement parlant, est incorrecte. Contrairement à `not` qui est réellement un opérateur, bien sûr essentiellement logique, il n'existe pas de *procédure* `Not` et une tentative d'utilisation en ce sens est inopérante :

```
> eval(Not) ;
```

Not

```
> Not(anything) ;
```

Not(*anything*)

alors que l'utilisateur naïf attendrait plutôt, compte tenu de ce qui a été expliqué page 75, que le dernier résultat soit `nothing`. La méthode utilisée par Maple pour donner à l'utilisateur la possibilité de manipuler les types est la suivante. Des symboles (`integer`, `boolean`, `symbol`, ...) sont réservés pour les types *élémentaires* (**integer**, **boolean**, **symbol**, ...). D'autres types peuvent alors être *décrits* à l'aide d'expressions fonctionnelles citant un pseudo-constructeur et un ou plusieurs types pseudo-arguments ; par exemple le descripteur `Not(float)` utilise le pseudo-constructeur `Not` et le pseudo-argument `float`. Le mécanisme est bien sûr récursif.

```
> type(exp(I*3.14159), And(complexcons, Not(realcons))) ;
```

true

```
> type(exp(I*Pi), And(complexcons, Not(realcons)))
```

false

Quelquefois le pseudo-constructeur peut être utilisé sans argument, mais un argument implicite `anything` est alors supposé.

```
> type([1,2,3.], list(float)) ;
```

false

```
> type([1,2,3.], list(integer)) ;
```

false

[1] Les choses sont en fait un peu plus compliquées, comme on peut le voir à l'aide de la procédure `dismantle` ; mais l'approximation donnée ici suffit pour ce qui nous concerne.

```
> type([1,2,3.], list) ;
```

true

Comme observé ci-dessus, un pseudo-constructeur comme `Not` ou `list` ne construit rien, c'est seulement un descripteur. Ce sont les procédures telles que `type` qui vont ensuite utiliser ces pseudo-constructeurs à bon escient. La situation entre `Not` et `type` est exactement la même que celle qui avait été signalée page 51 entre `Pi` et `evalf` : l'information que l'utilisateur *croit* voir dans `Pi` est en fait dans `evalf`[2] qui saura, le cas échant, traiter le symbole `Pi` comme il convient ; de la même façon l'information que l'utilisateur *croit* voir dans `Not(...)` est en fait dans `type` qui sait interpréter une telle expression.

Maple offre aussi à l'utilisateur la possibilité de définir de nouveaux types ; il s'agit là cette fois d'une vraie *construction* de type. Par définition même de la notion de type, il faut donc ajouter à l'environnement la procédure *fonction caractéristique* du nouveau type. Si dans votre cadre de travail l'ensemble des entiers multiples de 3 joue un rôle essentiel, vous pouvez construire le type correspondant comme suit :

```
> `type/divby3` := proc(obj)
    RETURN(type(obj, integer)
             and
           obj mod 3 = 0)
    end :
```

La procédure examine d'abord si l'objet considéré est un entier, *puis* s'il est divisible par 3. Ceci fait, la notion d'objet `divby3` a maintenant un sens dans l'environnement.

```
> type(x, divby3), type(44, divby3), type(45, divby3) ;
```

false, *false*, *true*

Le lecteur mathématicien aimerait savoir ici comment élucider si le nombre désigné par `x` est, *du point de vue arithmétique*, un entier. C'est un point bien connu de logique qu'il ne peut exister de méthode *générale* pour obtenir une réponse ; c'est à l'utilisateur de s'adapter à chaque cas de figure ; pour une expression comme la valeur de `x` page 74, il suffit d'utiliser la procédure `radnormal` qui sait obtenir une forme canonique pour les expressions pas trop compliquées à base de radicaux :

```
> radnormal(x) ;
```

4

et la réponse est claire.

Pour illustrer[3] la difficulté fondamentale à élucider si un nombre réel est entier, on va anticiper un peu sur la Section 3.4. Considérons les nombres réels définis par une suite de Cauchy. Dans le cadre de travail Maple, une suite de Cauchy pourrait être une procédure associant à un entier n un rationnel x_n de telle façon

[2] Et dans les procédures qui savent tirer parti de la nature mathématique du nombre π.

[3] Les considérations qui suivent sont déconseillées aux lecteurs fragiles aux troubles existentiels.

que $|x_{n+k} - x_n| < 2^{-n}$ si $k > 0$. Le *théorème de Cauchy*[4] assure que la suite (x_n) est convergente et définit donc un unique réel, sa limite. Le réel $\sqrt{2}$ peut par exemple être défini par :

```
> sq2 := proc(n)
    local i, rslt ;
    rslt := 2 ;
    for i from 1 to evalf(log[2](n+1)) do
      rslt := rslt/2 + 1/rslt
      od ;
    RETURN(rslt)
    end :
> sq2(100) ;
```

$$\frac{157258404803291863335321 7}{111198484434986813793811 2}$$

```
> evalf(sq2(100)) ;
```

$$1.414213562$$

```
> evalb(abs(sq2(100)-sq2(300)) < 2^(-100)) ;
```

true

Les logiciens savent alors démontrer l'*existence* d'une procédure :

```
> RealOfTheThirdType := proc(n)
    ...
    ...
    end ;
```

satisfaisant les conditions requises ci-dessus pour une suite de Cauchy ; le réel ainsi «défini» est entier mais ce *fait* n'admet aucune démonstration. La preuve d'existence d'une telle procédure n'est pas, ne peut pas être, *constructive* ; pire, aucune procédure ne peut être identifiée comme telle ! Mais on est «physiquement» certain que de telles procédures existent ; aussi certain qu'on sait qu'après avoir ajouté deux billes à un sac qui en contenait deux, le sac contient quatre billes. C'est un avatar du théorème d'incomplétude de Gödel [2].

Vous pouvez par exemple «tomber» un jour sur une procédure de ce type qui vous retourne pour chaque entier n *que vous essayez* une approximation x_n vérifiant $|3 - x_n| \leq 2^{-n}$; observant ceci, vous *suspectez* alors que le nombre réel limite x ainsi désigné est en fait l'entier 3 ; vous cherchez donc une *démonstration* que, quel que soit l'entier n, le résultat x_n vérifie cette inégalité, ce qui *prouverait* que le réel limite est bien l'entier 3. Mais vous ne trouvez pas de telle démonstration. À défaut, vous essayez un grand nombre de nouvelles valeurs de n espérant enfin trouver le contre-exemple qui pourrait exister à ce que vous aviez cru dans un premier temps ; mais non, pour toutes ces nouvelles valeurs de n, l'inégalité $|3 - x_n| \leq 2^{-n}$ est «désespérément» satisfaite ; utilisant ce «résultat» qui *semble* décidément confirmer que le réel défini est entier, vous remettez en chantier la recherche d'une démonstration, que vous ne trouvez pas, etc. etc.

Quand vous serez un peu fatigué de ce travail sans fin, inévitablement vous

[4] On peut si on préfère considérer que ceci est une définition de la notion de nombre réel.

suspecterez que vous êtes en face d'un réel du troisième type, mais vous ne pourrez jamais le démontrer non plus... L'exemple du nombre de Ramanujan vient à l'esprit, mais en fait il n'est pas de cette nature.

3.1.3 Le nombre de Ramanujan.

Le nombre de Ramanujan, $e^{\pi\sqrt{163}}$, est assez instructif des difficultés qui peuvent être rencontrées quand on veut savoir si un nombre est entier ou pas. Affectons-le au symbole `r` :

```
> r := exp(Pi*sqrt(163)) ;
```

$$r := \mathbf{e}^{\pi\sqrt{163}}$$

et calculons l'approximation flottante :

```
> evalf(r) ;
```

$$.2625374126\,10^{18}$$

Le nombre de Ramanujan a 18 chiffres «avant la virgule» et il nous faut donc une précision d'au moins 20 chiffres :

```
> evalf(r, 20)
```

$$.26253741264076874443\,10^{18}$$

Il est un peu fatigant de compter les chiffres pour voir où la «virgule» est insérée ; un affichage plus sophistiqué à base d'impression formattée s'impose : puisque 20 chiffres «exacts» sont prévus et que 18 doivent se trouver avant la virgule, l'affichage de type `%21.2f` est le bon :

```
> printf("%21.2f", evalf(r, 20)) ;
262537412640768744.43
```

Mais peut-on avoir confiance dans la partie décimale ? Poussons un peu la précision.

```
> printf("%31.12f", evalf(r, 30))
262537412640768743.999999999999
```

La question semble décidément plus délicate que prévu ! Il n'est certainement pas possible que les derniers chiffres des deux résultats précédents soient «exacts» en même temps... Augmentons encore la précision :

```
> printf("%41.22f", evalf(r, 40)) ;
262537412640768743.9999999999992500725944
> printf("%51.32f", evalf(r, 50)) ;
262537412640768743.99999999999925007259719818568865
```

Il est clair maintenant que le nombre de Ramanujan n'est pas entier. On note au passage que les deux derniers chiffres d'un résultat flottant sont en général assez douteux. Le théorème suivant permet de déterminer la nature arithmétique du

nombre de Ramanujan.

Théorème 18 (Schneider) — *Si le nombre α est algébrique différent de 0 et 1, et si le nombre β est algébrique irrationnel, alors le nombre α^β est transcendant.*

Il en résulte que le nombre de Ramanujan r est transcendant. En effet le nombre $\beta = i\sqrt{163}$ est algébrique : il est racine de l'équation $\beta^2+163 = 0$, et de plus il n'est pas rationnel ; si le nombre $\alpha = r$ était entier, il serait en particulier algébrique ; il est aussi différent de 0 et 1 et donc $\alpha^\beta = r^{i\sqrt{163}}$ serait transcendant. Calculons ce dernier avec Maple :

```
> r^(I*sqrt(163)) ;
```

$$(\mathrm{e}^{\pi\sqrt{163}})^{I\sqrt{163}}$$

```
> simplify(%) ;
```

$$-1$$

qui n'est pas un nombre transcendant ! le nombre de Ramanujan est donc transcendant. Ramanujan avait deviné il y a bien longtemps, sans Maple et sans ordinateur, que son nombre, malgré sa taille imposante, était non entier mais très proche d'un entier.

3.2 Le type integer.

Un objet **integer** est un entier de taille (nombre de chiffres) arbitraire, la seule limitation étant due à la capacité mémoire. Les sous-types **posint** (entiers *strictement* positifs), **negint** (entiers strictement négatifs), **nonnegint** (entiers positifs ou nuls), **nonposint** (entiers négatifs ou nuls), **even** (entiers pairs) et **odd** (entiers impairs) sont disponibles ; noter la terminologie à l'américaine où *positif* signifie strictement positif et *non négatif* signifie positif ou nul. Il faut utiliser ces sous-types chaque fois que c'est possible pour préciser les arguments de procédures. La programmation de la fonction factorielle peut être ainsi réalisée :

```
> fact := proc(n::nonnegint)
    local i, result ;
    result := 1 ;
    for i from 1 to n do result := i*result od ;
    RETURN(result)
    end :
> fact(0) ;
```

$$1$$

```
> fact(-1) ;
Error, fact expects its 1st argument, n, to be of type nonnegint, but received -1
```

La syntaxe «`::nonnegint`» pour un paramètre `n` indique que ce paramètre doit être de type **nonnegint**. Comme on le voit, la vérification est faite à chaque invocation de procédure, ce qui peut permettre de détecter des incohérences. Aucun mécanisme

analogue n'est disponible pour les variables des procédures, qu'elles soient locales[5] ou globales.

La programmation récursive rituelle de la fonction factorielle échoue sous Windows pour $n \geq 530$ pour cause de récursivité trop profonde.

On voit aussi qu'on peut refuser l'affichage d'un output en remplaçant le terminateur « ; » (point-virgule) par « : » (deux-points) ; l'instruction est exécutée, mais l'affichage de l'objet retourné par l'évaluateur n'est pas fait. On utilisera systématiquement cette façon de faire pour la construction des procédures, où l'output est en général de peu d'intérêt.

3.3 Le type rational.

Maple permet de manipuler banalement les rationnels.

```
> e_rational := proc(n::nonnegint)
    RETURN(add(1/i!, i=0..n))
    end :
> e30 := e_rational(30) ;
```

$$e30 := \frac{55464002213405654539818183437977}{2040406613939931220279296000000}$$

Numérateurs et dénominateurs sont certainement premiers entre eux ; la procédure à utiliser pour calculer le (P)GCD de deux entiers est `igcd` :

```
> igcd(numer(e30), denom(e30)) ;
```

$$1$$

Mais le dénominateur est-il 30! ?

```
> 30! / denom(e30) ;
```

$$13$$

Noter aussi que `whattype` ne répondra pas `rational` pour le type d'un tel nombre, mais seulement `fraction` :

```
> whattype(e30) ;
```

$$\textit{fraction}$$

```
> type(e30, rational) ;
```

$$\textit{true}$$

Curieusement les objets qui paraissent d'abord être des fractions sont en fait des produits :

[5] Possible sous Maple 6.

```
> type(A/B, fraction) ;
```

false

```
> type(A/B, `*`) ;
```

true

car le produit `A/B` est codé de façon interne `(A^1)*(B^(-1))`. La seule exception notable à cette remarque concerne le cas où `A` et `B` sont en fait des entiers, celui où `A/B` est donc alors un objet **rational**.

3.4 Le type float.

Les entiers et rationnels permettent le calcul exact. Mais les quantités mathématiques ne sont que rarement rationnelles et il faut donc souvent se contenter d'approximations dites *flottantes* de nombres réels. On peut par exemple dans certains contextes se contenter de l'approximation flottante `3.14159` de π ; elle a l'inconvénient d'être un peu inexacte, mais l'avantage du style *numérique* dans son expression.

La méthode usuelle est utilisée par Maple ; un flottant est une expression $a.10^k$ où a est un nombre décimal avec un certain nombre de chiffres *significatifs* et k est un entier positif ou négatif. Les usages de la plupart des langages de programmation sont respectés. C'est le point décimal qui impose le caractère flottant du nombre.

```
> whattype(4) ; whattype(4.) ;
```

integer

float

Maple prévoit par défaut une précision de dix chiffres significatifs pour les objets **float**. On force l'évaluation vers un flottant, partout où elle est possible, par la procédure `evalf`.

```
> evalf(20^20/30^30) ;
```

$$.5092866261\,10^{-18}$$

```
> evalb(4. = 4) ;
```

false

```
> is(4. = 4) ;
```

true

On voit ici une subtilité qui peut à l'occasion être importante ; la procédure d'évaluation booléenne `evalb` est «hard» et compare en particulier les types des objets ; la comparaison de l'entier `4` et du flottant `4.` est donc négative. Au contraire la procédure `is`, dans le cas numérique, compare *numériquement*. Voir aussi :

```
> x := (sqrt(5)-1) * (sqrt(5) + 1) ;
```

$$x := (\sqrt{5}-1)\,(\sqrt{5}+1)$$

```
> evalb(x = 4), is(x = 4) ;
```

false, *true*

La procédure `is` est très peu documentée ; le listing source est disponible, mais très complexe. Sauf cas assez simples, son usage n'est pas très clair et cette procédure doit donc être utilisée avec beaucoup de prudence.

Pour obtenir un listing source, il faut d'abord modifier un paramètre de l'interface appelé `verboseproc` :

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

puis afficher la procédure en tenant compte de la règle Eval-16, donc en demandant une évaluation *explicite* :

```
> eval(is) ;
```

proc(*obj*, *prop*)
local ...

Vous obtenez alors le listing source de la procédure `is` ; on vous souhaite bon courage pour la décrypter. Une procédure du *noyau* Maple est déclarée `builtin` :

```
> eval(evalb) ;
```

proc () **option** *builtin* ; 99 **end**

Seuls les super-gurus de Waterloo en connaissent alors la nature.

3.4.1 Combien de chiffres «significatifs» ?

La précision des calculs flottants peut être explicitement demandée par le deuxième argument de `evalf` pour un calcul ponctuel :

```
> evalf(exp(1), 20) ;
```

2.7182818284590452354

Par défaut, Maple prévoit des calculs avec 10 chiffres «exacts», information qu'on trouve dans la variable `Digits` :

```
> Digits ;
```

10

Rien n'empêche d'en changer la valeur :

```
> Digits := 30 ;
```

$Digits := 30$

et désormais tous les calculs flottants vont être effectués avec la précision demandée.

```
> evalf(sqrt(3)) ;
```

$$1.732050807568877293527446341 51$$

Une autre façon de demander qu'une évaluation flottante soit effectuée, consiste à prévoir des arguments flottants. Si Maple voit qu'une fonction a un argument flottant, Maple en déduit logiquement que le contexte est *calcul numérique* et continue dans le même esprit, on dit qu'il y a *contagion* du type `float`. Comparer :

```
> Digits := 10 :
  sin(1), sin(1.), sin(3/2), sin(1.5) ;
```

$$\sin(1), .8414709848, \sin(\frac{3}{2}), .9974949866$$

Les utilisateurs sont souvent surpris sinon agacés quand Maple laisse une expression comme `sin(1)` sans évaluation. Mais Maple préfère toujours garder la possibilité de mener ultérieurement des calculs *exacts*. Comparer les deux cas suivants :

```
> is(sin(1+1/3) = sin(1)*cos(1/3)+sin(1/3)*cos(1)) ;
```

$$true$$

```
> is(sin(1+1./3) = sin(1)*cos(1/3)+sin(1/3)*cos(1)) ;
```

$$false$$

Dans le premier cas les arguments sont gardés exacts et Maple connaît assez de trigonométrie pour conclure à l'égalité. Dans le second cas l'unique point décimal ajouté a, par *contagion*, transformé l'argument du premier sinus en une approximation flottante inexacte, d'où le résultat. Plus simplement :

```
> is(sin(1/3) = sin(1./3)) ;
```

$$false$$

```
> is(1/3 = 1./3) ;
```

$$true$$

La cohérence de l'ensemble de ces résultats n'est pas évidente ; tout provient de la façon dont la «contagion» flottante est propagée. Comme aucune règle logique *complète* ne peut être définie en la matière, quelles que soient les décisions prises, il y aura toujours une zone frontière avec des incohérences *apparentes* qui peuvent néanmoins sérieusement perturber les programmes rédigés par les utilisateurs trop crédules.

Comparer encore :

```
> sin(10000 * Pi), sin(10000 * evalf(Pi)) ;
```

$$0, .410206761510^{-5}$$

Le deuxième résultat est banalement inexact, compte tenu de l'approximation flottante utilisée pour π ; le premier est remarquablement exact : c'est en examinant soigneusement la nature *arithmétique* multiple de π de l'argument du sinus que Maple peut conclure à la nullité, résultat parfait.

Conseil 19 — *Dans des calculs d'envergure, il est important d'avoir les idées*

claires sur leur nature, numérique *vs* exacte.

1. *Dans certains contextes, il est important de conserver les expressions* exactes *des résultats, même s'ils sont affichés d'une façon un peu encombrante ou compliquée.*
2. *Si au contraire le contexte est purement* numérique*, l'exactitude de résultats intermédiaires peut considérablement alourdir le calcul ou même le mettre en échec.*

L'exemple détaillé Section 3.9 montre dans un cas concret comment l'utilisateur peut être mené à changer plusieurs fois de stratégie à ce propos en cours d'un calcul.

3.5 Le type realcons.

Le type **real** n'existe pas en Maple :

```
> type(3.4, real) ;
Error, type `real` does not exist
```

ce qui surprend un peu le mathématicien, mais le type **realcons** est prévu avec une sémantique un peu plus précise : on dit en même temps que l'objet visé est une constante.

```
> type(3, realcons), type(4/5, realcons),
  type(6.7, realcons), type(Pi, realcons) ;
```

$$true, true, true, true$$

```
> a := Pi * cos(5) / GAMMA(sqrt(2)) ;
```

$$a := \frac{\pi \cos(5)}{\Gamma(\sqrt{2})}$$

```
> type(a, realcons) ;
```

$$true$$

```
> evalf(a) ;
```

$$1.005154190$$

La documentation explique qu'un objet est de type **realcons** si la procédure `evalf` peut le prendre en argument et retourner un flottant réel. Il en est bien ainsi pour la valeur du symbole `a` ci-dessus, mais avec des exemples un peu plus compliqués, l'explication donnée est prise en défaut.

```
> b := int(exp(x^2)/(1+x), x=2..3) ;
```

$$b := \int_2^3 \frac{e^{x^2}}{1+x} dx$$

```
> type(b, realcons) ;
```

$$false$$

```
> evalf(b) ;
```

$$374.9536307$$

L'utilisateur doit donc rester assez prudent pour l'utilisation de ce type **realcons**.

Noter aussi que la procédure `whattype` ne retournera jamais la réponse `realcons`, car elle ne considère, par principe, que le type de codage de l'objet désigné ; au contraire la question `type(xxx, realcons)` est une interrogation *arithmétique* sur la valeur de `xxx`. Dans le même esprit on aimerait un type **integercons** mais on a déjà mentionné page 77 les problèmes logiques insurmontables posés par un tel type. L'incohérence relevée ci-dessus pour le symbole `b` illustre précisément comment ce type de problème logique intervient vite, même dans un contexte relativement banal, dès qu'une procédure prétend pouvoir donner une réponse à une question arithmétique.

3.6 Alias.

Un alias est un symbole déclaré *synonyme* d'un objet Maple. Le premier réflexe à la lecture de cette définition consiste à penser à la notion de chaînage, voir Section 2.3.2 ; un chaînage est en effet aussi un lien, installé dans l'*environnement*, entre un symbole et un objet Maple ; de sorte qu'une bonne lucidité va être nécessaire pour bien comprendre la différence entre les deux notions. Sans un effort significatif en ce sens, attendez-vous à des difficultés sérieuses dans les situations complexes.

Le sujet aurait mieux sa place comme satellite de ce qui avait été dit au sujet de l'environnement, mais, strictement parlant, les alias ne sont pas indispensables, de sorte qu'il est meilleur d'attendre qu'ils deviennent franchement utiles. C'est le cas dans les deux sections suivantes, consacrées aux nombres complexes et aux nombres algébriques, et le sujet est donc maintenant à l'ordre du jour.

Considérons donc qu'un alias est, comme un chaînage, un lien entre symbole et objet[6]. Il est installé par l'intermédiaire de la procédure `alias`. C'est une procédure où chaque argument est un objet **equation**, autrement dit deux objets séparés par le signe «=» ; ces objets sont évalués, et après évaluation, le membre de gauche doit être un symbole, la *source* de l'alias à définir, le membre de droite est un objet quelconque, le *but* de l'alias. Autant d'alias que d'équations seront construits. Dand l'exemple qui suit, les deux membres de l'équation sont à évaluation triviale, et le symbole source est donc `alias_symbol`, l'objet but est l'expression `1+x`.

```
> alias(alias_symbol = 1+x) ;
```

$$I, alias\_symbol$$

Quand un nouvel alias est installé, Maple retourne la suite des symboles aliasés, y compris le dernier ; dans l'environnement initial, seul le symbole `I` est aliasé. Il s'agit maintenant de bien analyser la différence *fondamentale* entre alias et chaînage.

```
> bounded_symbol := 1+y ;
```

$$bounded\_symbol := 1 + y$$

[6] En fait un alias peut admettre une source de type quelconque ; pour les usages les plus fréquents, la source est un symbole et, pour simplifier, nous considérons seulement ce cas.

Ces exemples sont parfaitement artificiels et ne figurent ici que dans un but didactique. Le point clé est le suivant. On a expliqué Section 2.2.2 le modèle **Read-Eval-Print** de Maple. On a compris l'essentiel de la différence entre chaînage et alias quand on a bien vu comment et pourquoi un chaînage n'intervient que pendant la phase **Eval** alors qu'un alias n'intervient que dans les phases **Read** et **Print**.

Pendant la phase **Read**, Maple lit caractère par caractère le texte de l'instruction que vous lui proposez, et construit l'objet Maple, plus généralement (cas d'instructions d'affectation, conditionnelle, itérative...) l'instruction Maple que vous voulez faire évaluer. Si un symbole aliasé figure dans votre texte, ce symbole va être *systématiquement* remplacé par l'objet correspondant. Pour mettre en évidence ce fait, comparez soigneusement ce qui suit.

```
> has('alias_symbol', x) ;
```

true

```
> has('bounded_symbol', y) ;
```

false

L'analyse est la suivante. Le *lecteur* Maple a construit dans le premier cas l'objet `has('1+x', x)`, car il a vu la présence du symbole `alias_symbol` et l'a remplacé par l'objet `1+x`. Puis l'objet construit par le lecteur est passé à l'*évaluateur*. La procédure `has` est standard et les arguments sont évalués avant d'être utilisés, de sorte que le premier argument définitif est la somme `1+x` : les quotes sont «retirés» par l'évaluation, voir Eval-10. Enfin la procédure `has` travaille et voit qu'effectivement le second argument `x` figure dans le premier `1+x`, elle retourne donc *true*. Dans le second cas, le symbole `bounded_symbol` *n'est pas* aliasé et le lecteur va donc passer à l'évaluateur l'objet `has('bounded_symbol', y)` ; rappelons que le lecteur n'est pas concerné par l'environnement. Quand l'évaluateur travaille, le premier argument provisoire `'bounded_symbol'` devient l'argument définitif `bounded_symbol` où *ne figure pas* le symbole `y` et la réponse est donc négative.

Montrons maintenant que le mécanisme alias ne travaille pas pendant une évaluation. Il est assez compliqué de mettre ce fait en évidence, car il faut arriver à faire évaluer le symbole aliasé *sans le montrer* parce qu'il serait aussitôt remplacé par le lecteur par le but de l'alias ! Une façon d'y arriver consiste à obtenir le symbole par résultat d'une procédure de construction. L'opérateur de construction d'un symbole par concaténation, opérateur «.», convient pour ce faire.

```
> i := 1 : sample.i ;
```

*sample*1

D'où la comparaison :

```
> has(alias_sym.bol, x) ;
```

false

```
> has(bounded_sym.bol, y) ;
```

true

Cette fois les objets passés à l'évaluateur sont exactement ceux que vous voyez, car le symbole `alias_symbol` *ne figure pas* dans le texte entré. Le premier argument définitif, construit par l'opérateur «.», est le symbole `alias_symbol`, car ce symbole *n'est pas présent* dans l'environnement, et l'objet retourné par l'opérateur de concaténation est donc l'argument définitif. Au contraire `bounded_symbol` est présent, de sorte que le premier argument définitif dans ce cas est la *valeur* de ce symbole, à savoir `1+y`. C'est une illustration du mécanisme de post-évaluation, cf. Eval-13 : le symbole retourné par l'opérateur de concaténation est à son tour évalué. D'où les réponses respectivement négative et positive.

On note ici que toute tentative de compréhension à la va-vite du style «avec `alias` ça évalue plus (ou moins) qu'avec l'affectation» est en échec. Dans le premier cas, c'était le mécanisme `alias` qui «évaluait plus», dans le second cas c'était le contraire, mais de toute façon cette terminologie pifométrique est impropre et doit être proscrite. Le bon point de vue consiste à avoir les idées claires sur la phase de travail de Maple où le lien symbole-objet est utilisé.

Enfin le mécanisme `alias` reprend du service dans la phase **Print**. Le plus simple pour le mettre en évidence est ceci :

```
> 1 + x, 1 + y ;
```

$$alias\_symbol, 1 + y$$

Maple recherche en effet systématiquement dans les objets affichés des composantes aliasées, et le cas échéant remplace ces occurenses par l'alias. Ici Maple a vu la présence de l'objet `1+x` et l'a donc remplacé *dans l'affichage* par le symbole `alias_symbol`. On voit donc que, contrairement aux chaînages de l'environnement, le lien alias est *bidirectionnel* alors que le lien chaînage est *orienté*. Ce point est mis en évidence par :

```
> alias_symbol, bounded_symbol ;
```

$$alias\_symbol, 1 + y$$

où les deux «évaluations» sont parfaitement différentes. Ici, le lecteur passe à l'évaluateur l'objet «`1+x, bounded_symbol`» ; l'évaluation retourne l'objet «`1+x, 1+y`», mais cet objet est *affiché* «$alias\_symbol, 1+y$» alors qu'en fait le symbole `x` est bien dans cet objet :

```
> has([%], x) ;
```

$$true$$

Ici, il faut encadrer le résultat *suite* entre crochets pour en faire une *liste*, sinon les arguments passés à `has` seraient incorrects ; voir ■■■.

Rien n'interdit d'avoir à la fois le même lien entre symbole et objet, *simultanément* sous forme alias et chaînage environnement. Compte tenu de la pré-évaluation des arguments de la procédure `alias`, ceci peut être fait comme suit.

```
> symb := 1+x ;
```

$$symb := 1 + x$$

```
> alias('symb' = symb)
```

$$I, alias\_symb, symb$$

Selon les exemples donnés précédemment, on vérifie la présence simultanée souhaitée chaînage et alias, par les instructions ésotériques :

```
> has(sy.mb, x) , 1+x ;
```

$$true, symb$$

Ce n'est pas un exercice de style sans motivation ; on verra que les calculs dans les groupes de Galois, Chapitre LLL, sont grandement facilités par cette technique.

3.7 Le type complexcons.

Maple n'a pas de type **real**, mais il a un type **complex** ; de plus le type **complexcons** est aussi défini, et la cohérence de l'ensemble est donc pour le moins assez obscure. On conseille de n'utiliser en fait que le type **complexcons** avec la sémantique que l'on devine. Le type **complex** est à considérer comme un archaïsme survivant de versions antérieures.

Les nombres complexes «simples» sont codés par l'intermédiaire de la constante `I` qui est en fait un alias pour l'expression `sqrt(-1)`. Bien noter que le `I` doit être *majuscule*, cause constante de lapsus clavier ; le plus souvent le symbole `i` minuscule est dédié aux entiers indices ou pilotes d'itération, mais ce n'est qu'un usage. Les rôles ainsi disjoints de `I` et `i` se révèlent à la longue très commodes.

Donc `I` est une abréviation externe, utilisable en entrée, systématiquement utilisée par Maple en sortie, pour l'objet que les mathématiciens, surtout ceux du XVIII et XIX-ièmes siècles, désignaient $\sqrt{-1}$; sans cet alias, on devrait utiliser `sqrt(-1)`, ce qui serait un peu lourd. On peut y jouer un moment en supprimant l'alias `I`.

```
> x := 3+4*I ;
```

$$3 + 4I$$

```
> alias(I = I) ;
> x ;
```

$$3 + 4\sqrt{-1}$$

```
> alias(I = sqrt(-1))
```

$$I$$

```
> x ;
```

$$3 + 4I$$

Comme pour un symbole qu'on veut libérer, la suppression d'un alias est conventionnellement demandée par l'équation «`symb = symb`» si `symb` est le symbole source dont on veut supprimer l'alias. Noter en particulier que les alias *ne sont pas utilisés* pendant l'évaluation des arguments de la procédure `alias`, sans quoi il serait très difficile le plus souvent d'atteindre le symbole dont on veut supprimer ou modifier l'alias ! Ceci a comme conséquence un inconvénient quelquefois assez pervers :

Warning 20 — *On ne peut pas utiliser un alias pour en définir un autre.*

C'est un point qu'il est capital de ne pas oublier quand on travaille par exemple avec des extensions successives de corps, car il est alors bien tentant de définir l'alias pour le générateur de la deuxième extension en utilisant l'alias qui a servi à «aliaser» le générateur de la première. On discutera ce point en détail en temps utile.

Maple est organisé pour traiter de façon cohérente les expressions où figurent des radicaux avec un radicande négatif.

```
> sqrt(-5) ;
```

$$I\sqrt{5}$$

On a expliqué plus haut que la procédure `whattype` ne retourne jamais la réponse `realcons`, de nature arithmétique ; pour la même raison elle ne retournera pas `complexcons`. Comme toujours pour un objet structuré, la procédure `whattype` retourne l'opérateur dominant.

```
> whattype(3+4*I) ;
```

$$+$$

L'opérateur dominant est en effet l'addition et `whattype` n'a pas de raison d'examiner les opérandes qui peuvent être très... complexes. Le type indiqué pour `I` lui-même est ;

```
> whattype(I) ;
```

$$\hat{}$$

car `I` est en fait codé comme $\sqrt{-1} = (-1)^{1/2}$; l'opérateur maître est donc l'exponentiation.

Comme souvent, il est plus approprié de chercher si tel objet est de tel type :

```
> type(I, complexcons) ;
```

$$true$$

```
> type(3+4*I, complexcons) ;
```

$$true$$

```
> restart ; type(a+b*I, complexcons) ;
```

$$false$$

Un objet est de type **complexcons** s'il est une *constante* complexe, donc sans symbole libre. Les réels sont en particulier des complexes :

```
> type(3, complexcons) ;
```

$$true$$

Les considérations de la section précédente au sujet du critère utilisant `evalf` pour caractériser un objet **realcons** s'appliquent aussi bien au type **complexcons**.

D'une façon générale, tout calcul ayant un sens dans le contexte complexe, qu'il

soit numérique ou symbolique, peut être tenté.

```
> int(1/(1+I*x+x^2), x=0..infinity)
```

$$\frac{1}{5}\,\pi\,\sqrt{5}-\frac{2}{5}\,I\,\operatorname{arctanh}\left(\frac{1}{5}\,\sqrt{5}\right)\sqrt{5}$$

```
> evalf(%) ;
```

$$1.404962947-.4304089412\,I$$

Il arrive même assez souvent que les utilisateurs soient surpris par le degré de généralité supposé par Maple ; en particulier Maple *ne suppose pas* par défaut que vos symboles sont à valeurs réelles. D'où l'aventure qui survient inexorablement aux novices quand ils essaient de séparer parties réelle et imaginaire d'un complexe. L'examen de la documentation mène aux procédures `Re` et `Im` qui travaillent comme on pense :

```
> Re(exp(I*Pi/10)), Im(exp(I*Pi/10)) ;
```

$$\frac{1}{4}\,\sqrt{2}\sqrt{5+\sqrt{5}},\ \frac{1}{4}\,\sqrt{5}-\frac{1}{4}$$

C'est le premier essai possible ; mais quelquefois celui qui vient à l'esprit est le suivant ; on crée un complexe «abstrait»

```
> z := x+I*y ;
```

$$z := x+Iy;$$

et on «vérifie» que les procédures `Re` et `Im` travaillent comme on croit avoir compris :

```
> Re(z), Im(z) ;
```

$$\Re(x+Iy),\ \Im(x+Iy)$$

et on a la surprise de voir nos procédures refuser de travailler ! L'explication est simple : il n'a jamais été dit que les symboles `x` et `y` sont à considérer comme à valeurs réelles, hypothèse nécessaire pour la séparation souhaitée. On a expliqué plus haut que le type `real` n'existe pas dans Maple, mais la procédure `assume` vous permet de demander à Maple de prendre en compte l'*hypothèse* qu'un symbole est à valeur réelle, même et surtout si votre symbole est libre :

```
> assume(x, real) ;
> Re(z), Im(z) ;
```

$$x\sim-\Im(y),\ \Re(y)$$

```
> assume(y, real) ;
> Re(z), Im(z) ;
```

$$x\sim,\ y\sim$$

Les variables à propos desquelles une ou des hypothèses ont été faites sont signalées par un tilde suivant leur nom. Ce n'est pas le sujet de cette section, mais le contexte est favorable pour signaler dans le même registre :

```
> restart ;
> sqrt(x^2) ;
```

$$\sqrt{x^2}$$

```
> simplify(%) ;
```

$$\mathrm{csgn}(x)\,x$$

Dans un contexte numérique où son argument est complexe, la procédure `sqrt` retourne la racine carrée de partie réelle positive. Il est dès lors impossible de valider une simplification $\sqrt{x^2} = x$ sans information complémentaire sur x. En particulier si x est négatif, la bonne simplification est $\sqrt{x^2} = -x$. La procédure `csgn` donne le signe de la partie réelle. Si on donne à Maple une information complémentaire, il peut affiner la simplification.

```
> assume(x, negative) ;
> simplify(sqrt(x^2)) ;
```

$$-x\sim$$

La procédure *evalc* tente de présenter son argument sous une forme `xexpr + I*yexpr`.

```
> exp(I*Pi/10) ;
```

$$\mathbf{e}^{\left(\frac{1}{10}\,I\pi\right)}$$

```
> evalc(%) ;
```

$$\frac{1}{4}\,\sqrt{2}\sqrt{5+\sqrt{5}}+I\,\left(\frac{1}{4}\,\sqrt{5}-\frac{1}{4}\right)$$

```
> evalc(exp(I*Pi/11)) ;
```

$$\cos(\frac{1}{11}\,\pi)+I\,\sin(\frac{1}{11}\,\pi)$$

Quand un objet est évalué sous `evalc`, les symboles libres figurant dans l'objet sont supposés réels et l'évaluateur tient compte de cette hypothèse momentanée pour séparer si c'est possible partie réelle et imaginaire. Ceci permet assez souvent de se passer de la procédure `assume` quand les questions de séparations sont très localisées. Exemples typiques après un `restart` nécessaire pour supprimer les hypothèses précédemment installées par `assume` :

```
> restart ;
> Re(x+I*y) ; evalc(Re(x+I*y)) ;
```

$$\Re(x+Iy)$$

$$x$$

```
> Re(exp(x+I*y)) ; evalc(Re(exp(x+I*y))) ;
```

$$\Re(\mathbf{e}^{x+Iy})$$

$$\mathbf{e}^{x}\cos(y)$$

3.8 Nombres algébriques.

Le corps des complexes $\mathbb{C}$, sujet de la section précédente, n'est autre que le corps de décomposition du polynôme X^2+1, un polynôme $\mathbb{R}$-irréductible. Ce mécanisme a une portée théorique très générale et il est particulièrement intéressant dans le cas du corps $\mathbb{Q}$ des rationnels, pour les polynômes à coefficients rationnels irréductibles. Pour les considérations élémentaires sur les extensions algébriques de corps, voir par exemple [6, 4].

Maple contient de ce point de vue un ensemble d'outils puissants, permettant à l'utilisateur de construire des extensions algébriques et d'y *calculer* aussi facilement qu'avec les entiers, rationnels, flottants et complexes, au moins tant que l'obstacle temps de calcul ou espace mémoire n'est pas rencontré.

Le sujet est vaste et on se contentera ici d'une introduction à partir d'un exemple. On va construire le corps de décomposition du polynôme $X^3 + X + 1$, puis calculer un peu dans ce corps. On vérifie d'abord que ce polynôme est bien $\mathbb{Q}$-irréductible.

```
> P3 := X^3 + X + 1 ;
```

$$P3 := X^3 + X + 1$$

```
> irreduc(P3) ;
```

$$true$$

Le *corps* $K_2 = \mathbb{Q}[X]/(P3)$, extension du corps $K_1 = \mathbb{Q}$, est donc mathématiquement défini. Pour pouvoir y travailler commodément on procède dans l'espace de travail Maple exactement comme dans l'espace de travail mathématique. On écrit volontiers en mathématiques que $K_2 = \mathbb{Q}[\alpha]$ où α est *un* nouveau «nombre» vérifiant $\alpha^3 + \alpha + 1 = 0$. Autrement dit K_2 est obtenu à partir de K_1 par adjonction d'un nouvel objet. Les éléments de $\mathbb{Q}[\alpha]$ sont les polynômes $a\alpha^2 + b\alpha + c$ avec a, b et c rationnels ; chaque tel polynôme est à considérer comme le représentant canonique d'une classe d'équivalence de $\mathbb{Q}$-polynômes modulo $P3$; l'irréductibilité de $P3$ assure que K_2 est bien un corps. En particulier α est racine du polynôme $P3$; le corps K_2 contient donc au moins une racine de $P3$ mais pour le moment on ne sait pas s'il en contient d'autres.

Pour ajouter α à notre espace de travail Maple, on crée un *alias*.

```
> alias(alpha = RootOf(P3)) ;
```

$$I, \alpha$$

et désormais α est un objet comme un autre, un nombre algébrique, qui, strictement parlant, n'est rien d'autre que la classe d'équivalence de X dans le quotient $\mathbb{Q}[X]/(P3)$. Un calcul hors contexte ne donne rien de nouveau par rapport à ce qui était précédemment disponible :

```
> (1 + alpha)^6 ;
```

$$(1+\alpha)^6$$

```
> expand(%) ;
```
$$1 + 6\alpha + 15\alpha^2 + 20\alpha^3 + 15\alpha^4 + 6\alpha^5 + \alpha^6$$

et on voit que Maple semble avoir «oublié» la relation $\alpha^3 = -\alpha - 1$. On demande à Maple d'en tenir compte systématiquement en demandant le calcul dans le *contexte algébrique*. De même que la procédure `evalf` demande de privilégier le contexte *flottant*, la procédure `evala` demande l'exécution d'un travail dans le contexte algébrique.

```
> evala((1 + alpha)^6) ;
```
$$-5\alpha^2 - 12 - 21\alpha$$

Ici le résultat est simple, mais dans ce contexte les résultats fleuves sont fréquents et il peut être important pour la lisibilité d'écrire le polynôme résultat ordonné :

```
> sort(%, alpha)
```
$$-5\alpha^2 - 21\alpha - 12$$

Pour voir que le mécanisme d'alias est seulement auxiliaire :

```
> evala(RootOf(X^3+2*X+1)^3) ;
```
$$-1 - 2 * \mathrm{RootOf}(\_Z^3 + 2 * \_Z + 1)$$
```
> evala(RootOf(X^3+X+1)^3) ;
```
$$-\alpha - 1$$

On voit que dans le deuxième cas, même si on n'utilise pas l'alias disponible en entrée, Maple l'utilise si possible en sortie.

Une autre façon de voir que le polynôme initial $P3$ est irréductible, consiste à tenter de le factoriser.

```
> factor(P3) ;
```
$$X^3 + X + 1$$

Dans le corps $K_1 = \mathbb{Q}$, le polynôme $P3$ n'a pas de racine. Par contre il a une racine dans $K_2 = \mathbb{Q}[\alpha]$, ce qui est mis en évidence comme suit :

```
> factor(P3, alpha) ;
```
$$(X^2 + \alpha X + 1 + \alpha^2)\,(X - \alpha)$$

On voit qu'on dispose dans la procédure `factor` d'un argument optionnel complémentaire indiquant que la factorisation doit être réalisée dans une extension de $\mathbb{Q}$, définie par l'intermédiaire d'un objet $\mathfrak{RootOf}$, adjoint à $\mathbb{Q}$.

```
> type(alpha, RootOf) ;
```
$$true$$

Le corps K_2 contient comme il se doit la racine α du polynôme $P3$, mais pas d'autre. Le corps de décomposition de $P3$ n'est pas encore construit. Il faut construire une extension K_3 de degré 2 de K_2 en ajoutant une racine β du facteur

de degré 2 de $P3$. Extrayons ce polynôme. Il s'agit de diviser $P3$ par $(X-\alpha)$; à nouveau le contexte `evala` est indispensable.

```
> P2 := evala(P3/(X-alpha)) ;
```

$$P2 := X^2 + \alpha X + 1 + \alpha^2$$

On ajoute à notre environnement une racine de $P2$[7].

```
> alias(beta = RootOf(P2)) ;
```

$$I, \alpha, \beta$$

Ceci va nous permettre de travailler maintenant dans le corps $K_3 = K_2[\beta] = \mathbb{Q}[\alpha, \beta]$. Notre nouvel élément β est bien racine du polynôme initial :

```
> evala(subs(X = beta, P3)) ;
```

$$0$$

On peut d'ailleurs demander la factorisation de $P3$ dans K_3 :

```
factor(P3, {alpha, beta}) ;
```

$$(X-\beta)\,(X+\alpha+\beta)\,(X-\alpha)$$

Le corps K_3 est donc le corps de décomposition de $P3$. C'est une extension de degré 6 de $\mathbb{Q}$ dont un exercice classique consiste à trouver un *élément primitif*. C'est à la portée de Maple.

```
> evala(Primfield({alpha, beta})) ;
```

$$[[\mathrm{RootOf}(9\_Z^2 + 31 + 6\_Z^4 + \_Z^6) = 2\,\alpha + \beta],$$
$$[\alpha = \frac{2}{9} + \frac{1}{2}\,\mathrm{RootOf}(9\_Z^2 + 31 + 6\_Z^4 + \_Z^6) + \frac{5}{18}\,\mathrm{RootOf}(9\_Z^2 + 31 + 6\_Z^4 + \_Z^6)^2$$
$$+\frac{1}{18}\,\mathrm{RootOf}(9\_Z^2 + 31 + 6\_Z^4 + \_Z^6)^4,$$
$$\beta = -\frac{4}{9} - \frac{5}{9}\,\mathrm{RootOf}(9\_Z^2 + 31 + 6\_Z^4 + \_Z^6)^2 - \frac{1}{9}\,\mathrm{RootOf}(9\_Z^2 + 31 + 6\_Z^4 + \_Z^6)^4]]$$

L'affichage est assez encombrant, mais on repère immédiatement l'occurence répétée d'un nouveau `RootOf` pour lequel on va aussi créer un alias ; on voit que notre `RootOf` est en particulier le *premier* membre de l'équation qui est le *premier* (unique) élément de la *première* liste du résultat ; notre alias peut donc être créé comme suit[8] :

```
> alias(gamma = op([1,1,1], %)) ;
```

$$I, \alpha, \beta, \gamma$$

moyennant quoi on peut *réécrire* le résultat antérieur :

---

[7] Attention, ne pas écrire ici :

```
> alias(beta = RootOf(X^2 + alpha*X + 1+alpha^2))
```

car un alias ne peut pas être défini par une expression où figure *explicitement* un autre alias.

[8] Voir Section 4.3 pourquoi la procédure `op` peut être ainsi utilisée.

```
> %% ;
```

$$[[\gamma = 2\alpha + \beta], [\alpha = \frac{2}{9} + \frac{1}{2}\gamma + \frac{5}{18}\gamma^2 + \frac{1}{18}\gamma^4, \beta = -\frac{4}{9} - \frac{5}{9}\gamma^2 - \frac{1}{9}\gamma^4]]$$

qui est maintenant bien lisible. Que faut-il lire ? Que l'élément primitif γ choisi par Maple est la combinaison $\gamma = 2\alpha + \beta$, et que les deux générateurs α et β s'expriment comme polynômes simples à l'aide de γ. Le polynôme minimal de γ est très particulier ; c'est un polynôme de degré 6 et pourtant son corps de décomposition est aussi de degré 6, alors qu'un polynôme de degré 6 choisi «au hasard» a en général un corps de décomposition de degré 720 sur $\mathbb{Q}$. Extrayons ce polynôme ; la variable système `_Z` est laide et on la remplace par `X` :

```
> P6 := subs(_Z=X, op(gamma)) ;
```

$$P6 := 9X^2 + 31 + 6X^4 + X^6$$

Ce polynôme est certainement irréductible :

```
> irreduc(P6) ;
```

$$true$$

Mais il est entièrement décomposé dans le corps $\mathbb{Q}(\gamma)$:

```
factor(P6, gamma)
```

$$-\frac{1}{1296}\,(6X + 4 + 3\gamma + 5\gamma^2 + \gamma^4)\,(-6X + 4 - 3\gamma + 5\gamma^2 + \gamma^4)\,(6X + 4 - 3\gamma + 5\gamma^2 + \gamma^4)$$
$$(-6X + 4 + 3\gamma + 5\gamma^2 + \gamma^4)\,(X + \gamma)\,(-X + \gamma)$$

On est tout près de la théorie de Galois qu'on réexaminera plus tard dans ce manuel ; disons seulement que le degré du corps de décomposition d'un polynôme de degré inférieur à 8 est directement accessible sous Maple comme suit :

```
> galois(P6)[4] ;
```

$$6$$

à comparer au groupe de Galois du premier polynôme de degré 6 choisi au hasard par Maple ; l'option `terms=7` sert à forcer la présence d'un terme constant.

```
> galois(randpoly(X, degree=6, terms=7))[4]
```

$$720$$

3.9 Un exemple float vs Not(float).

Cette section peut être omise en première lecture ; de nombreuses procédures non encore étudiées y sont utilisées. Ceci dit, une imitation fidèle dans une session Maple de ce qui est montré ici ne devrait poser aucun problème, même pour un lecteur novice ; et elle peut être instructive.

Considérons le problème suivant : on veut construire le polynôme d'interpola-

tion $P(z)$ de degré 7 vérifiant les relations :

$$P(e^{2ik\pi/8}) = k \quad ; \quad 0 \leq k \leq 7.$$

Puisqu'il s'agit d'un problème très classique d'interpolation polynômiale, une procédure Maple est peut-être toute prête pour un tel travail ? La recherche **Topic Search** mène rapidement à la procédure `interp`, très simple d'utilisation. L'exemple suivant suffit pour en comprendre le fonctionnement.

```
> P := interp([1,2,3,4], [81,70,97,63], X) ;
```

$$P := -\frac{33}{2} * X^3 + 118 * X^2 - \frac{499}{2} * X + 229$$

```
> seq(subs(X=k, P), k=1..4) ;
```

$$81, 70, 97, 63$$

On détermine dans l'exemple ci-dessus le polynôme de degré 3 vérifiant $P(1) = 81$, ..., $P(4) = 63$. Puis le résultat obtenu est contrôlé : l'appel de la procédure `seq` demande l'affichage, pour $1 \leq k \leq 4$, de la valeur de $P(k)$ obtenue par substitution $X \mapsto k$ dans l'expression de P.

Un essai dans le même sens avec les racines 8-ièmes de l'unité mérite d'être tenté. Les données sont un peu plus...complexes, et il est meilleur de les préparer d'abord.

```
> xdata := [seq(exp(2*I*k*Pi/8), k=0..7)] ;
```

$$xdata := [1, \frac{1}{2}\sqrt{2} + \frac{1}{2}I\sqrt{2}, I, -\frac{1}{2}\sqrt{2} + \frac{1}{2}I\sqrt{2}, -1, -\frac{1}{2}\sqrt{2} - \frac{1}{2}I\sqrt{2}, -I, \frac{1}{2}\sqrt{2} - \frac{1}{2}I\sqrt{2}]$$

```
> ydata := [seq(k, k=0..7)] ;
```

$$ydata := [0, 1, 2, 3, 4, 5, 6, 7]$$

Maple sait exprimer élémentairement les racines 8-ièmes de l'unité. Il ne reste plus qu'à utiliser ces données :

```
> interp(xdata, ydata, Z) ;
```

$$-\frac{X}{((-\frac{1}{2}\sqrt{2} + \frac{1}{2}I\sqrt{2} - 1)(-\frac{1}{2}\sqrt{2} + \frac{1}{2}I\sqrt{2} - I)(I - \frac{1}{2}\sqrt{2} - \frac{1}{2}I\sqrt{2})(\frac{1}{2}\sqrt{2} + \frac{1}{2}I\sqrt{2} - 1))} + 12 \ldots$$

Le temps de calcul est très bref, 1 seconde sur un PC moyen, mais le résultat[9], dont on n'a montré ici que le début de la première ligne, est fleuve : l'auteur a compté patiemment 379 écrans de formules à base de radicaux. Les tentatives rituelles à base de `simplify` et `radnormal` pour tenter de réduire le résultat ne donnent rien, de sorte que le résultat est pratiquement inexploitable. On trouve ici une situation typique où un calcul *exact* échoue à cause de sa trop grande complexité. Si le but du calcul est le tracé graphique obtenu page 102, des données flottantes sont toujours suffisantes, et il est donc légitime de se rabattre sur un calcul de type flottant. On convertit donc nos racines de l'unité sous forme flottante et on applique le même calcul de polynôme d'interpolation.

[9] L'expérience montre que l'ordre des termes affichés est arbitraire d'une session à l'autre.

```
> xdataf := evalf(xdata) ;
```

$$xdataf := [1., .7071067810 + .7071067810\,I, 1.\,I, -.7071067810 + .7071067810\,I, -1., -.7071067810 - .7071067810\,I, -1.\,I, .7071067810 - .7071067810\,I]$$

```
> Pf := interp(xdataf, ydata, Z) ;
```

$$\begin{aligned} Pf := {} & .2\,10^{-8}I - .500000001Z - .499999999Z^3 + 1.207106778IZ - .499999997Z^2 \\ & + .500000001IZ^2 + .207106785IZ^3 - .207106783IZ^5 - .500000000IZ^6 \\ & - 1.207106783IZ^7 - .500000004Z^4 - .499999999Z^5 - .50000000003Z^6 \\ & + 3.500000000 - .4999999997Z^7 \end{aligned}$$

Le polynôme d'interpolation est cette fois visible d'un coup d'œil ; de plus les coefficients sont remarquablement simples et laissent subodorer des propriétés arithmétiques souterraines dans le calcul d'interpolation en cours. Les coefficients du style 1.207106778 sont moins clairs, mais leur quasi répétition nous incite à élucider ce qui peut expliquer ces phénomènes. Une idée naturelle consiste alors à *isoler* le coefficient de Z dans P ; on sait qu'il est géant et on ne l'affiche pas.

```
> a1 := coeff(P, Z, 1) :
```

puis on normalise par rapport aux radicaux ce coefficient *isolé*. Cette fois la normalisation va rapidement au bout et donne :

```
> a1rn := radnormal(a1) ;
```

$$a1rn := \frac{1}{2}I + \frac{1}{2}I\sqrt{2} - \frac{1}{2}$$

Le résultat est si simple qu'il mérite d'être comparé au coefficient correspondant du polynôme d'interpolation à coefficients flottants.

```
> evalf(a1rn - coeff(Pf, Z, 1)) ;
```

$$.10\,10^{-8} + .30\,10^{-8}\,I$$

Aux erreurs d'arrondi près, les coefficients sont bien les mêmes. On peut faire le même travail pour les autres coefficients, un par un, ce qui peut d'ailleurs être automatisé : on a expliqué plus haut que les tentatives usuelles de simplification échouent, mais il est maintenant clair qu'en traitant les coefficients un par un, on doit aboutir. On oriente normalement le calcul en ce sens comme suit, mais ne le faites pas !

```
> P2 := collect(P, X) ;
```

Il semble en effet qu'un bug[10] Maple soit ici : Maple reste sans réponse, les indicateurs «Time» et «Bytes» en bas de l'écran restent inactifs, et l'utilisateur ne peut reprendre la main. Seule solution : abattre sa session de l'extérieur.

Ceci est assez troublant, et au lieu d'utiliser la procédure `collect`, on peut chercher à traiter soi-même la gestion coefficient par coefficient.

[10]Sous versions 5 et 5.1, sous Windows ou sous Unix.

```
> P2 := add(radnormal(coeff(P, X, i))*X^i, i=0..7)) ;
```

$$P2 := \frac{7}{2} + (\frac{1}{2}\,I\sqrt{2} + \frac{1}{2}\,I - \frac{1}{2})\,X + (-\frac{1}{2} + \frac{1}{2}\,I)\,X^2 + (\frac{1}{2}\,I\sqrt{2} - \frac{1}{2}\,I - \frac{1}{2})\,X^3 - \frac{1}{2}\,X^4 + (-\frac{1}{2}\,I\sqrt{2} - \frac{1}{2} + \frac{1}{2}\,I)\,X^5 + (-\frac{1}{2} - \frac{1}{2}\,I)\,X^6 + (-\frac{1}{2} - \frac{1}{2}\,I\sqrt{2} - \frac{1}{2}\,I)\,X^7$$

Le but est-il atteint ? Si on en juge d'après le résultat obtenu, oui ; ceci dit le dernier calcul demande une quarantaine de secondes[11] et compte tenu de la simplicité du résultat obtenu, il est très tentant de chercher une méthode plus simple permettant de court-circuiter les objets fleuves intermédiaires.

C'est le moment de penser que la solution réputée naïve pour le problème d'interpolation passe par une matrice de Vandermonde. Il s'agit en effet de trouver des coefficients α_i vérifiant les huit équations linéaires :

$$\sum_{j=0}^{7} \alpha_j x_k^j = k \quad \text{avec } x_k = e^{2ik\pi/8}, 0 \leq k \leq 7.$$

La matrice des coefficients de ce système linéaire est la matrice de Vandermonde par rapport aux racines 8-ièmes de l'unité ; elle a des propriétés bien particulières. Pour montrer commodément la matrice en question, créons un alias pour la racine génératrice, qu'on a intérêt à exprimer sous forme *abstraite*[12], comme racine d'un polynôme irréductible :

```
> alias(alpha = RootOf(Z^4+1)) ;
```

$$I, \alpha$$

Réexprimons notre liste de racines :

```
> xdata := [seq(alpha^k, k=0..7)] ;
```

$$xdata := [1, \alpha, \alpha^2, \alpha^3, \alpha^4, \alpha^5, \alpha^6, \alpha^7]$$

Maple ne semble pas savoir que $\alpha^4 = -1$, mais on peut lui souffler d'exploiter ce type de relation par application de la procédure `evala` expliquée page 94.

```
> xdata := evala(xdata) ;
```

$$xdata := [1, \alpha, \alpha^2, \alpha^3, -1, -\alpha, -\alpha^2, -\alpha^3]$$

On va utiliser les procédures spécifiques de calcul matriciel, qui nécessitent le chargement préalable du package `linalg`,

```
> with(linalg) :
```

puis on construit la matrice de Vandermonde engendrée par nos racines :

[11] Sur un PC moyen, on ne le précisera plus.

[12] Plus précisément α est la classe du monôme Z dans le corps $Q[Z]/(Z^4+1)$; c'est un corps cyclotomique et le polynôme y est donc entièrement décomposé.

```
> vdm := vandermonde(xdata) ;
```

$$vdm := \begin{bmatrix} 1 & 1 & 1 & 1 & 1 & 1 & 1 & 1 \\ 1 & \alpha & \alpha^2 & \alpha^3 & \alpha^4 & \alpha^5 & \alpha^6 & \alpha^7 \\ 1 & \alpha^2 & \alpha^4 & \alpha^6 & \alpha^8 & \alpha^{10} & \alpha^{12} & \alpha^{14} \\ 1 & \alpha^3 & \alpha^6 & \alpha^9 & \alpha^{12} & \alpha^{15} & \alpha^{18} & \alpha^{21} \\ 1 & -1 & 1 & -1 & 1 & -1 & 1 & -1 \\ 1 & -\alpha & \alpha^2 & -\alpha^3 & \alpha^4 & -\alpha^5 & \alpha^6 & -\alpha^7 \\ 1 & -\alpha^2 & \alpha^4 & -\alpha^6 & \alpha^8 & -\alpha^{10} & \alpha^{12} & -\alpha^{14} \\ 1 & -\alpha^3 & \alpha^6 & -\alpha^9 & \alpha^{12} & -\alpha^{15} & \alpha^{18} & -\alpha^{21} \end{bmatrix}$$

On sait comment tenir compte de $\alpha^4 = -1$; une évaluation *explicite* du symbole `vdm` est ici nécessaire pour atteindre la matrice pointée par le symbole `vdm` : l'exception signalée pour les valeurs procédures en Eval-16 intervient aussi pour les valeurs matricielles ; voir Eval-▙▟▙. D'où l'appel étrange mais inévitable «`evala(eval(...`». On cache le résultat, évident mais encombrant.

```
> vdm := evala(eval(vdm)) :
```

L'inverse de cette matrice, clé du processus «naïf» d'interpolation, est calculé par la procédure `inverse` du package `linalg`.

```
> ivdm := inverse(vdm) ;
```

$ivdm :=$

$$\begin{bmatrix} \frac{1}{8} & \frac{1}{8} & \frac{1}{8} & \frac{1}{8} & \frac{1}{8} & \frac{1}{8} & \frac{1}{8} & \frac{1}{8} \\ \frac{1}{8} & -\frac{1}{4}\frac{1}{\alpha(-1+\alpha^4)} & \frac{1}{8}\frac{1}{\alpha^2} & \frac{1}{4}\frac{\alpha}{-1+\alpha^4} & -\frac{1}{8} & \frac{1}{4}\frac{1}{\alpha(-1+\alpha^4)} & -\frac{1}{8}\frac{1}{\alpha^2} & -\frac{1}{4}\frac{\alpha}{-1+\alpha^4} \\ \frac{1}{8} & \frac{1}{8}\frac{1}{\alpha^2} & -\frac{1}{8} & -\frac{1}{8}\frac{1}{\alpha^2} & \frac{1}{8} & \frac{1}{8}\frac{1}{\alpha^2} & -\frac{1}{8} & -\frac{1}{8}\frac{1}{\alpha^2} \\ \frac{1}{8} & \frac{1}{4}\frac{\alpha}{-1+\alpha^4} & -\frac{1}{8}\frac{1}{\alpha^2} & -\frac{1}{4}\frac{1}{\alpha(-1+\alpha^4)} & -\frac{1}{8} & -\frac{1}{4}\frac{\alpha}{-1+\alpha^4} & \frac{1}{8}\frac{1}{\alpha^2} & \frac{1}{4}\frac{1}{\alpha(-1+\alpha^4)} \\ \frac{1}{8} & -\frac{1}{8} & \frac{1}{8} & -\frac{1}{8} & \frac{1}{8} & -\frac{1}{8} & \frac{1}{8} & -\frac{1}{8} \\ \frac{1}{8} & \frac{1}{4}\frac{1}{\alpha(-1+\alpha^4)} & \frac{1}{8}\frac{1}{\alpha^2} & -\frac{1}{4}\frac{\alpha}{-1+\alpha^4} & -\frac{1}{8} & -\frac{1}{4}\frac{1}{\alpha(-1+\alpha^4)} & -\frac{1}{8}\frac{1}{\alpha^2} & \frac{1}{4}\frac{\alpha}{-1+\alpha^4} \\ \frac{1}{8} & -\frac{1}{8}\frac{1}{\alpha^2} & -\frac{1}{8} & \frac{1}{8}\frac{1}{\alpha^2} & \frac{1}{8} & -\frac{1}{8}\frac{1}{\alpha^2} & -\frac{1}{8} & \frac{1}{8}\frac{1}{\alpha^2} \\ \frac{1}{8} & -\frac{1}{4}\frac{\alpha}{-1+\alpha^4} & -\frac{1}{8}\frac{1}{\alpha^2} & \frac{1}{4}\frac{1}{\alpha(-1+\alpha^4)} & -\frac{1}{8} & \frac{1}{4}\frac{\alpha}{-1+\alpha^4} & \frac{1}{8}\frac{1}{\alpha^2} & -\frac{1}{4}\frac{1}{\alpha(-1+\alpha^4)} \end{bmatrix}$$

Un appel de `evala` va nous simplifier considérablement cette matrice.

```
> ivdm := evala(eval(ivdm)) ;
```

$$ivdm := \begin{bmatrix} \frac{1}{8} & \frac{1}{8} & \frac{1}{8} & \frac{1}{8} & \frac{1}{8} & \frac{1}{8} & \frac{1}{8} & \frac{1}{8} \\ \frac{1}{8} & -\frac{1}{8}\alpha^3 & -\frac{1}{8}\alpha^2 & -\frac{1}{8}\alpha & -\frac{1}{8} & \frac{1}{8}\alpha^3 & \frac{1}{8}\alpha^2 & \frac{1}{8}\alpha \\ \frac{1}{8} & -\frac{1}{8}\alpha^2 & -\frac{1}{8} & \frac{1}{8}\alpha^2 & \frac{1}{8} & -\frac{1}{8}\alpha^2 & -\frac{1}{8} & \frac{1}{8}\alpha^2 \\ \frac{1}{8} & -\frac{1}{8}\alpha & \frac{1}{8}\alpha^2 & -\frac{1}{8}\alpha^3 & -\frac{1}{8} & \frac{1}{8}\alpha & -\frac{1}{8}\alpha^2 & \frac{1}{8}\alpha^3 \\ \frac{1}{8} & -\frac{1}{8} & \frac{1}{8} & -\frac{1}{8} & \frac{1}{8} & -\frac{1}{8} & \frac{1}{8} & -\frac{1}{8} \\ \frac{1}{8} & \frac{1}{8}\alpha^3 & -\frac{1}{8}\alpha^2 & \frac{1}{8}\alpha & -\frac{1}{8} & -\frac{1}{8}\alpha^3 & \frac{1}{8}\alpha^2 & -\frac{1}{8}\alpha \\ \frac{1}{8} & \frac{1}{8}\alpha^2 & -\frac{1}{8} & -\frac{1}{8}\alpha^2 & \frac{1}{8} & \frac{1}{8}\alpha^2 & -\frac{1}{8} & -\frac{1}{8}\alpha^2 \\ \frac{1}{8} & \frac{1}{8}\alpha & \frac{1}{8}\alpha^2 & \frac{1}{8}\alpha^3 & -\frac{1}{8} & -\frac{1}{8}\alpha & -\frac{1}{8}\alpha^2 & -\frac{1}{8}\alpha^3 \end{bmatrix}$$

Si on a l'œil, on observe que l'inverse de notre matrice de Vandermonde est assez extraordinaire ; c'est l'un des rares cas où *l'inversion du cancre* réussit (presque), celle qui consiste à prendre comme inverse de la matrice la matrice des inverses ! L'inverse du cancre peut être obtenu comme suit :

```
> dunce_ivdm := map(item -> 1/item, vdm) :
  iszero(evala(evalm(8 - dunce_ivdm &* vdm))) ;
```

$$true$$

après quoi on vérifie que le produit de notre candidat inverse par la matrice à inverser est 8 fois la matrice identité ; la procédure `evalm` évalue l'expression matricielle et l'appel de `evala` demande de tenir compte de $\alpha^4 = -1$; enfin la procédure `iszero` examine si la matrice obtenue est nulle.

L'inversion si facile est due aux propriétés particulières de l'ensemble des racines de l'unité : on a retrouvé les propriétés les plus simples de la transformation de Fourier discrète, voir par exemple [5]. Un autre point de vue peut aussi être considéré ; examinons le *conditionnement* de la matrice à inverser. Comparer :

```
> evalf(cond(vdm)) , evalf(cond(vandermonde(ydata))) ;
```

$$8.000000000, .920659911\,10^{8}$$

C'est pourquoi l'interpolation polynômiale par rapport à une suite d'entiers $0, \ldots, n$ est si instable (phénomène de Runge par exemple), alors que, sur un groupe de racines de l'unité, les mêmes instabilités ne sont pas observées.

Le polynôme qu'on avait obtenu très facilement en calcul flottant mais sans comprendre sa nature peut maintenant être obtenu en calcul exact : on multiplie matriciellement le vecteur des valeurs `ydata` par l'inverse `ivdm` ; on obtient ainsi la liste des coefficients du polynôme d'interpolation cherché qu'on construit ensuite par un mécanisme d'addition itérative. On vérifie numériquement que c'est bien,

aux erreurs d'arrondi près, le polynôme flottant déjà obtenu ; l'appel intermédiaire de la procédure `collect` sert à regrouper les coefficients des mêmes puissances de `Z`, nécessaire, compte tenu de l'interférence avec le générateur complexe `I`.

```
> coefficients := evala(evalm(ivdm &* ydata)) ;
```

$$coefficients := [\tfrac{7}{2}, -\tfrac{1}{2}+\tfrac{1}{2}\alpha^3+\tfrac{1}{2}\alpha^2+\tfrac{1}{2}\alpha, -\tfrac{1}{2}+\tfrac{1}{2}\alpha^2, -\tfrac{1}{2}+\tfrac{1}{2}\alpha-\tfrac{1}{2}\alpha^2+\tfrac{1}{2}\alpha^3, -\tfrac{1}{2}, -\tfrac{1}{2}-\tfrac{1}{2}\alpha^3+\tfrac{1}{2}\alpha^2-\tfrac{1}{2}\alpha, -\tfrac{1}{2}-\tfrac{1}{2}\alpha^2, -\tfrac{1}{2}-\tfrac{1}{2}\alpha-\tfrac{1}{2}\alpha^2-\tfrac{1}{2}\alpha^3]$$

```
> exactP := add(coefficients[i]*Z^(i-1), i=1..8) ;
```

$$exactP := \tfrac{7}{2}+(-\tfrac{1}{2}+\tfrac{1}{2}\alpha^3+\tfrac{1}{2}\alpha^2+\tfrac{1}{2}\alpha)Z+(-\tfrac{1}{2}+\tfrac{1}{2}\alpha^2)Z^2+(-\tfrac{1}{2}+\tfrac{1}{2}\alpha-\tfrac{1}{2}\alpha^2+\tfrac{1}{2}\alpha^3)Z^3 -\tfrac{1}{2}Z^4+(-\tfrac{1}{2}-\tfrac{1}{2}\alpha^3+\tfrac{1}{2}\alpha^2-\tfrac{1}{2}\alpha)Z^5+(-\tfrac{1}{2}-\tfrac{1}{2}\alpha^2)Z^6 +(-\tfrac{1}{2}-\tfrac{1}{2}\alpha-\tfrac{1}{2}\alpha^2-\tfrac{1}{2}\alpha^3)Z^7$$

```
> evalf(collect(Pf - subs(alpha = exp(2*I*Pi/8), exactP), Z));
```

$$-.44\,10^{-8}IZ^7+(-.18\,10^{-8}-.12\,10^{-8}I)Z^6+(-.18\,10^{-8}+.20\,10^{-8}I)Z^5 +(.20\,10^{-8}+.1\,10^{-8}I)Z^4+(.18\,10^{-8}-.20\,10^{-8}I)Z^3+(-.40\,10^{-8}+.12\,10^{-8}I)Z^2 +(.28\,10^{-8}+.44\,10^{-8}I)Z+.1\,10^{-8}-.1\,10^{-8}I$$

Notre polynôme d'interpolation est cette fois obtenu par un calcul qui n'a posé aucune difficulté particulière : aucun résultat intermédiaire ingérable à l'écran, 1.3 seconde de temps CPU.

On voulait obtenir une courbe *fermée* du plan complexe joignant les huit points réels de `ydata`. Il suffit de tracer l'image du cercle unité par le polynôme `exactP`. Il faut mieux utiliser la procédure `complexplot` du package `plots` :

```
> with(plots, complexplot) :
```

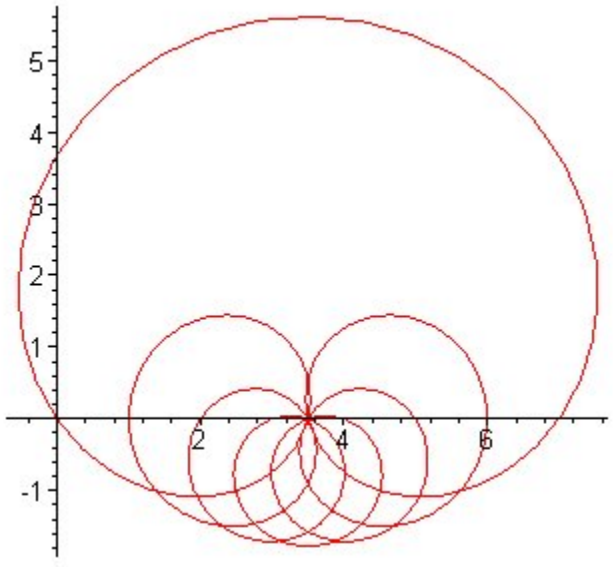
On remplace notre racine «abstraite» α par `exp(2*I*Pi/8)` $= e^{2i\pi/8}$, on remplace aussi `X` par `exp(I*t)` $= e^{it}$ et on se contente des approximations flottantes :

```
> fourierP := evalf(subs({alpha = exp(2*I*Pi/8), X = exp(I*t)}, exactP)) ;
```

$$fourierP := 3.500000000+(-.4999999998+1.207106781\,I)\,\mathbf{e}^{(1.\,I\,t)} +(-.5000000000+.4999999998\,I)\,(\mathbf{e}^{(1.\,I\,t)})^2+(-.4999999998+.2071067810\,I)\,(\mathbf{e}^{(1.\,I\,t)})^3 -.5000000000\,(\mathbf{e}^{(1.\,I\,t)})^4+(-.5000000002-.2071067810\,I)\,(\mathbf{e}^{(1.\,I\,t)})^5 +(-.5000000000-.4999999998\,I)\,(\mathbf{e}^{(1.\,I\,t)})^6+(-.5000000002-1.207106781\,I)\,(\mathbf{e}^{(1.\,I\,t)})^7$$

Le tracé est alors obtenu par la procédure `complexplot` ; l'argument supplémentaire «`scaling = constrained`» force Maple à utiliser la même échelle horizontalement et verticalement.

```
> complexplot(fourierP, t=0..2*Pi, scaling = constrained) ;
```



Une propriété du graphique obtenu est inattendue ; la courbe d'interpolation passe 6 fois au point 3.5. On est ici dans une situation typique où on constate *expérimentalement* une propriété pas du tout évidente a priori. Peut-on obtenir une confirmation autre que visuelle ? Déterminons la suite des points où notre polynôme d'interpolation vaut 3.5 :

```
> solve(Pf = 3.5) ;
```

$$-.9963394407 - 0.08548518823\,I, -.6440712188 + .7649655315\,I, -.5009167252 - .8654954829\,I,$$
$$-.1414213565\,10^{-8} + .5857864416\,10^{-9}\,I, .2577961103 + .9661993401\,I,$$
$$.4381080388 - .8989223241\,I, .9454232373 + .3258449052\,I$$

puis considérons les modules des racines obtenues :

```
> map(abs, [%]) ;
```

$$[.9999999993, .9999999996, .9999999983, .1530733733\,10^{-8},$$
$$.9999999996, .9999999992, .9999999999]$$

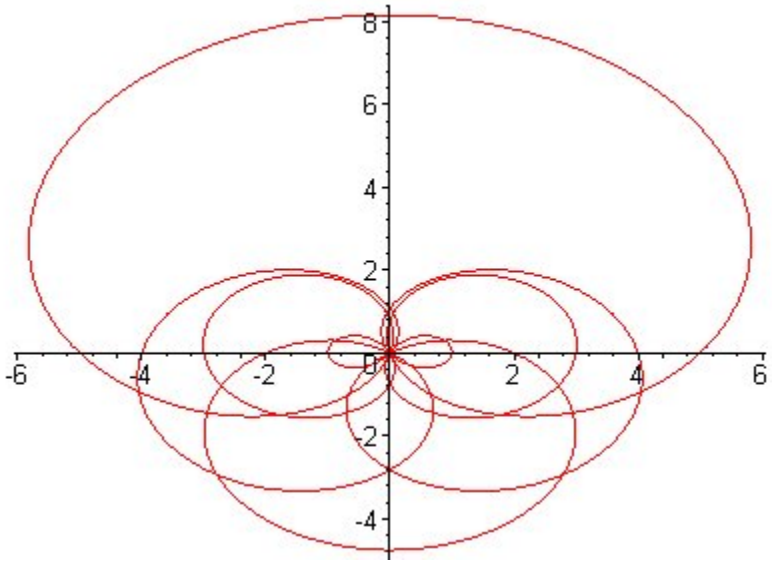
On «voit» ainsi la racine 0, confirmée aussi par la forme exacte du polynôme d'interpolation, et les six autres racines «manifestement» de module 1. Pour confirmer ce point de vue, on peut commencer par pousser la précision ; le calcul avec 30 chiffres significatifs ne prend pratiquement pas plus de temps et «confirme» notre conjecture. Si au lieu de prendre huit points sur le cercle unité du plan complexe, on en prend un nombre pair quelconque, le même résultat est observé ; et un résultat analogue est aussi constaté dans le cas impair. Il reste à obtenir une démonstration mathématique générale de ce fait, mais c'est une autre histoire[13]

[13] **Indications**. Considérons d'abord le cas où n est pair et k un entier $0 \leq k < n/2$. On va s'intéresser aux racines de l'unité $\alpha_\ell = e^{(2\ell-(n-1))i\pi/n}$ pour $0 \leq \ell < n$. Il y a n telles racines, mais disposées par paires, symétriquement par rapport à l'axe réel. On considère alors le polynôme d'interpolation $P_{n,k}$ vérifiant $-P(\alpha_k) = P(\alpha_{n-k-1}) = k$ et $P(\alpha_\ell) = 0$ pour les autres α_ℓ. Ce

C'est un exemple élémentaire mais typique de *Mathématiques expérimentales* : des «coïncidences» troublantes *observées* sur des résultats numériques ou graphiques permettent d'énoncer et de démontrer des propriétés qu'il aurait été autrement difficile de trouver.

-o-o-o-o-o-o-o-o-

polynôme est à coefficients réels ; il a la racine 0 et les autres racines sont les $\alpha_\ell \neq \alpha_k, \alpha_{n-k-1}$; on en déduit sa forme $P_{n,k} = XQ_{n,k}$ où le polynôme $Q_{n,k}$ est à coefficients symétriques ; de plus le polynôme $Q_{n,k}$ est strictement positif sur l'axe réel. Donc toute combinaison *à coefficients positifs* des polynômes $P_{n,k}$ (n fixé) va avoir les mêmes propriétés : une racine 0 et les autres racines sur le cercle unité du plan complexe. Or le polynôme du texte se déduit d'une telle combinaison par une manipulation simple. Une étude analogue aboutit dans le cas n impair. La symétrie assez rudimentaire à l'origine de notre phénomène est ainsi élucidée ; par exemple le polynôme d'interpolation où les valeurs successives sur les 10 racines 10-ièmes de l'unité sont 2, 4, 1, 3, 5, -5, -3, -1, -4 et -2 va présenter la même propriété : la courbe image du cercle unité va passer huit fois par l'origine :



Chapitre 4

Listes, ensembles, suites et tables.

4.1 Introduction.

Ce chapitre est consacré à des objets Maple qui, selon le contexte, pourraient être appelés *composés*, ou *itératifs*, bien qu'aucun de ces qualificatifs ne soit suffisant pour définir notre sujet. En langage C++, une paire d'accolades {..., ...} regroupe plusieurs instructions pour en produire une seule, qu'on appelle alors *instruction composée*. Dans le même esprit, si `obj1`, `obj2`, ..., `objn` sont des objets Maple, alors :

```
[obj1, obj2, ..., objn]
```

désigne un *objet composé*, appelé *liste*, ou encore objet **list**. Mais d'autres objets comme par exemple l'objet **function** :

```
operator(operand1, ..., operandn)
```

est aussi en un sens un objet composé, mais qui n'est pourtant ni une liste, ni un ensemble, ni une suite, ni une table. On verra toutefois que la «suite» des opérandes «`operand1, ..., operandn`» est bien une *suite* au sens précis donné par Maple à cette notion, et de telles suites d'opérandes (ou d'arguments) seront donc considérées dans ce chapitre, notamment Section 4.4.

Un autre point de vue est à considérer ; l'instruction C++ :

```
for {int i = 0 ; i < 10 ; ++i}
   cout << i^3 << endl ;
```

exécute 10 fois la même instruction, à ceci près que l'environnement est différent à chaque itération : la valeur de `i` est modifiée. Dans le même esprit l'instruction Maple :

```
> [seq(i^3, i=0..9)] ;
```

$$[0, 1, 8, 27, 64, 125, 216, 343, 512, 729]$$

construit une liste où les éléments sont tous de même nature, un cube d'entier, mais l'entier en question est variable d'un entier à l'autre. Un tel objet peut donc être appelé naturellement un *objet itératif*, mais d'autres listes ont au contraire des éléments très différents deux à deux et ne peuvent pas raisonnablement entrer dans cette catégorie.

Décidons d'appeler listes, ensembles, suites et tables des *objets multiples*, pourquoi pas. Les objets multiples Maple sont donc de type **list**, **set**, **exprseq** ou **table**. Le type **table** admet le sous-type **array**, lequel admet à son tour les sous-types **matrix** et **vector** ; les deux derniers types sont bien entendus dédiés d'abord au calcul matriciel.

Le type **exprseq** est vraiment l'un des ingrédients essentiels de Maple. Les lispeurs reconnaîtront dans les objets de type **exprseq**, absolument partout dans Maple, la version adaptée des *listes* de Lisp ; rappelons que «Lisp» est une contraction de *List Processing*. De même que les cellules **cons** sont les particules élémentaires des objets Lisp, l'ingrédient de base pour la construction des listes Lisp, la structure de type **exprseq** est le *noyau* de la structuration des objets Maple. C'est peu dire que la compréhension des objets de ce type est la clé indispensable pour entrer dans la confrérie des initiés Maple.

Warning 21 — *Les quatre types* **exprseq** *(suite),* **list** *(liste),* **set** *(ensemble) et* **table** *(table) sont* disjoints deux à deux. *Une* suite *ne peut pas être ni une* liste*, ni un* ensemble*, ni une* table*. De même pour toutes les autres combinaisons deux à deux.*

Rien n'empêche un élément *de suite d'être une liste, un ensemble ou une table ; de même pour les éléments des listes, ensembles et tables. Par contre un élément de suite, liste ou ensemble ne peut* jamais *être une suite. Un élément de table peut lui être absolument de tout type ; il peut en particulier être une suite.*

Les relations d'appartenance possibles sont réunies dans ce tableau.

$\in$	**exprseq**	**list**	**set**	**table**
exprseq	Non	Non	Non	Oui
list	Oui	Oui	Oui	Oui
set	Oui	Oui	Oui	Oui
table	Oui	Oui	Oui	Oui

Ce warning est incompréhensible à ce point ; il n'est énoncé que pour définir un but qui *doit* être atteint : n'espérez pas utiliser Maple pour des applications non triviales si vous n'avez pas compris ce warning. Il s'agit dans les sections suivantes de décrypter ces assertions, pour le moment ésotériques. Ces questions ne sont pas capitales en elles-mêmes, mais elles constituent un excellent thème didactique permettant de vérifier que ces structures sont bien comprises.

4.2 Premier examen des types multiples.

On construit une suite, une liste, un ensemble et une table, tous très simples.

```
> seq4 := a, b, c, d ;
  list4 := [a, b, c, d] ;
  set4 := {a, b, c, d} ;
  table4 := table([1=a, 2=b, 3=c, 4=d]) ;
```

$$seq4 := a, b, c, d$$

$$list4 := [a, b, c, d]$$

$$set4 := \{a, b, c, d\}$$

$$table4 := \mathrm{table}([1 = a, 2 = b, 3 = c, 4 = d])$$

Nos objets multiples ont tous quatre éléments. Pour construire une *suite*, les éléments constituants sont séparés par virgule. De même pour *liste* et *ensemble*, à ceci près que pour une liste il faut prévoir en plus crochets ouvrant et fermant, et pour un ensemble, accolades ouvrante et fermante. Pour une table, on peut utiliser le constructeur `table` avec comme argument une *liste* d'équations «clé = objet».

On voit qu'une liste (resp. ensemble) n'est rien d'autre, typographiquement, qu'une suite encadrée de crochets (resp. accolades). Ceci n'est pas une convention pour les représentations externes ; il en est essentiellement de même en mémoire. Avant de détailler ce point, il faut d'abord expliquer que la virgule *ne doit pas* être comprise comme un procédé syntaxique ; c'est véritablement l'opérateur infixé n-aire permettant de *construire* une suite. Comparer dans ce registre les deux instructions :

```
> a + b + c ;
  a , b , c ;
```

$$a + b + c$$

$$a, b, c$$

où l'analogie est parfaite, et ce n'est pas une analogie, c'est une réalité. L'auteur suit le plus souvent en entrée l'usage typographique : pas de blanc avant la virgule, un blanc après ; mais ceci n'est qu'un usage, sans aucune importance pour le lecteur Maple.

Donc une expression avec n opérandes séparés par $n-1$ virgules est *lue* comme une *suite* de n éléments. Ceci n'a de sens que si $n \geq 2$. En particulier la notion de *suite à un élément* n'a pas de sens ! Par contre la suite vide existe et est accessible en particulier par l'intermédiaire du symbole `NULL`, protégé, qui pointe vers cette suite, invisible, comme on le voit ci-dessous.

```
> whattype(seq4) ;
```
$$exprseq$$
```
> wrong_seq1 := a ;
```
$$wrong\_seq1 := a$$
```
> whattype(wrong_seq1) ;
```
$$symbol$$
```
> right_seq0 := NULL ;
```
$$right\_seq0 :=$$
```
> whattype(right_seq0)
```
$$exprseq$$

On a expliqué qu'une *liste* (resp. *ensemble*) n'est autre qu'une *suite* encadrée par crochets (resp. accolades). C'est si vrai que les liste et ensemble pointés par `list4` et `set4` auraient pu aussi bien être construits comme suit :

```
> list4 := [seq4] ; set4 := {seq4} ;
```
$$list4 := [a, b, c, d]$$
$$set4 := \{a, b, c, d\}$$

et en mémoire il en est de même ; un objet **list** est une entête indiquant qu'il s'agit d'une liste et un unique pointeur vers un objet **exprseq**, à moins que la liste ait un seul élément, auquel cas le pointeur pointe directement vers cet élément, car une suite à un élément n'a pas de sens[1].

Donc listes et ensembles ne sont en quelque sorte que des objets satellites de suites. Une liste n'est autre qu'une suite avec un «chapeau» indiquant qu'il s'agit en fait d'une liste, de même pour un ensemble. Il y a pourtant une différence entre les *constructeurs* de listes et d'ensembles ; une liste n'est vraiment rien d'autre qu'une suite avec un «chapeau», un ensemble aussi, mais le constructeur d'ensemble va *en plus* vérifier que les objets constituants sont différents deux à deux. Le cas échéant, les objets doublons sont écartés. De plus l'ordre des objets affichés est sans importance pour un ensemble alors qu'il l'est pour une liste. En fait Maple trie pour un ensemble selon les adresses machine. Comparer :

```
> seq6 := a, z, e, r, a, z ;
```
$$seq6 := a, z, e, r, a, z$$
```
> list6 := [seq6] ;
```
$$list6 := [a, z, e, r, a, z]$$
```
> set4 := {seq6} ;
```
$$set4 := \{a, r, z, e\}$$

Avant de poursuivre notre examen des types multiples, donnons-nous les moyens d'analyse d'objets offerts par Maple.

[1] Les passionnés d'implémentation machine peuvent examiner ces questions avec les procédures `addressof`, `pointto` et `disassemble`.

4.3 Les procédures op et nops.

Les objets Maple sont constitués d'objets élémentaires assemblés selon des procédures variées riches en possibilités, mais en définitive assez simples. On a déjà expliqué Section 2.2.3 que les types de données étaient récursifs ; ceci permet, avec des procédés de construction assez rudimentaires, de construire des objets fort complexes. Inversement il faut être capable d'analyser indéfiniment comment est structuré un objet quelconque, analyse qui n'est terminée que quand les composantes *irréductibles* sont identifiées, ainsi que leur mode d'assemblage. Les principaux outils d'analyse sont les procédures `op` et `nops`.

L'usage de ces analyseurs est très facile, sauf pour les *suites*. On a expliqué Section 2.4.2 que les objets **function** ont un *opérateur* et des *opérandes* ; dans des cas particuliers, il peut y avoir seulement un opérande, ou même pas d'opérande du tout. Le nombre d'opérandes est obtenu par la procédure `nops`, chaque opérande est obtenu par la procédure `op`, l'opérateur aussi. Pour faire comprendre le mécanisme, considérons l'objet **function** pointé par le symbole `function3` :

```
> function3 := abc(bcd1, bcd2, bcd3) ;
```

$$function3 := \mathrm{abc}(bcd1, bcd2, bcd3)$$

```
> whattype(function3) ;
```

$$function$$

L'objet **function** est construit puis affecté au symbole `function3` ; quand on demande le type de la *valeur* de ce symbole, la procédure `whattype` répond justement `function`. On obtient le nombre d'opérandes de la *valeur* du symbole `function3` à l'aide de la procédure `nops` :

```
> nops(function3) ;
```

$$3$$

Moyennant quoi on sait que la valeur de `function3` est un objet avec trois opérandes. On peut demander par exemple le deuxième opérande :

```
> op(2, function3) ;
```

$$bcd2$$

Par extension, le «0-ième» opérande est l'*opérateur* :

```
> op(0, function3) ;
```

$$abc$$

On a ainsi un procédé très général et très simple pour analyser tout objet. Considérons par exemple cet objet appelé une *intégrale inerte* ; c'est une intégrale où le choix de l'opérateur `Int` commençant par une majuscule, au lieu de l'opérateur d'intégration usuel `int` commençant par une minuscule, demande explicitement à Maple de *ne pas* évaluer l'intégrale et donc de la garder telle quelle. On verra plus tard de nombreuses circonstances où de tels objets «inertes» sont utiles, voire indispensables.

```
> II := Int(x^3, x=1..2) ;
```

$$II := \int_1^2 x^3 \, dx$$

La valeur de `II` semble un peu exotique pour le débutant, mais il ne doit pas se laisser intimider par l'apparence extérieure de cette intégrale non évaluée ; les procédures `nops` et `op` vont lui permettre d'analyser la structure de cet objet.

```
> nops(II) ;
```

$$2$$

```
> op(0, II) ;
```

$$Int$$

```
> op(1, II) ;
```

$$x^3$$

```
> op(2, II) ;
```

$$x = 1..2$$

On voit qu'en fait le format interne correspond exactement à ce qui avait été entré pour la création de l'objet.

Le choix d'une intégrale *inerte* pour cette illustration n'est pas innocent ; l'intégrale inerte, par principe, n'est pas évaluée. Comparer le même exercice avec la procédure `int` ordinaire, celle qui cherche à effectuer l'intégration :

```
> nops(int(x^3, x=1..2)) ;
```

$$2$$

```
> op(0, int(x^3, x=1..2)) ;
```

$$fraction$$

```
> op(1, int(x^3, x=1..2)) ;
```

$$15$$

```
> op(2, int(x^3, x=1..2)) ;
```

$$4$$

Car l'intégrale est évaluée et retourne $15/4$:

```
> int(x^3, x=1..2) ;
```

$$\frac{15}{4}$$

Il faut rester toujours vigilant à ce propos ; malgré leur parfum «système», les procédures `nops` et `op` sont *standard* et évaluent leurs arguments avant de les utiliser, source permanente de pièges dans les debuggings fiévreux.

Les procédures `nops` et `op` fonctionnent en particulier pour les *listes*. Considérons cette liste biscornue :

```
> list4 := [27, x^2+y^2, Pi, Int(x^3, x=1..2)] ;
```

$$list4 := \left[27, x^2 + y^2, \pi, \int_1^2 x^3\, dx\right]$$

```
> nops(list4) ;
```

$$4$$

Notre liste, dans la terminologie Maple, est donc un objet à quatre *opérandes*, correspondant à ce qui est appelé d'habitude les *éléments* de la liste. On peut demander l'opérande 4 :

```
> op(4, list4) ;
```

$$\int_1^2 x^3\, dx$$

et l'opérande 0 :

```
> op(0, list4) ;
```

$$list$$

On peut demander l'opérande 2 de l'opérande 4 :

```
> op(2, op(4, list4)) ;
```

$$x = 1..2$$

ou si on préfère :

```
> op([4, 2], list4) ;
```

$$x = 1..2$$

Noter que les indices *successifs* sont alors à lire de gauche à droite et *doivent* être les constituants d'une *liste*. Et ainsi de suite :

```
> op([4,2,1], list4) ;
```

$$x$$

On voit qu'on dispose bien ainsi du scanner permettant d'identifier chaque composante d'un objet, à quelque profondeur que ce soit. Si on va trop loin, une erreur intervient.

```
> op([4,2,1,2], list4) ;
Error, improper op or subscript selector
```

L'opérande 0 peut être aussi structuré.

```
> op(0, f(g)(h)) ;
```

$$\mathrm{f}(g)$$

```
> op([0,1], f(g)(h)) ;
```

$$g$$

```
> op([0,0], f(g)(h)) ;
```

$$f$$

```
> op([1,0], f(g(h))) ;
```

$$g$$

On pratique aussi assez souvent l'analyse niveau par niveau, en descendant un étage à chaque fois par rapport au résultat précédent atteint par le symbole système «ditto», à savoir «`%`» :

```
> op(4, list4) ;
```

$$\int_1^2 x^3\,dx$$

```
> op(1, %) ;
```

$$x^3$$

```
> op(0, %) ;
```

$$\hat{}$$

Si un numéro d'opérande est *négatif* dans l'analyse à l'aide de la procédure `op`, la numérotation commence par la fin. Une autre façon de retrouver notre intégrale inerte est donc :

```
> op(-1, list4) ;
```

$$\int_1^2 x^3\,dx$$

```
> op([-1,2,-2], list4) ;
```

$$x$$

On peut encore appeler la procédure `op` avec un seul argument, l'objet à analyser ; à défaut de précision sur l'opérande souhaité, la procédure retourne alors la *suite* des opérandes, opérateur exclu.

```
> op(list4) ;
```

$$27, x^2+y^2, \pi, \int_1^2 x^3\,dx$$

Autrement dit on a réalisé la conversion objet **list** → objet **exprseq**, et ceci est la *seule* façon d'y parvenir. Cette conversion est souvent indispensable et l'exemple qui précède doit donc être bien compris. Un exemple typique d'application consiste à chercher l'élément maximum d'une liste ; la procédure `max` est disponible et tout le monde espère d'abord pouvoir l'utiliser ainsi :

```
> list5 := [13, 26, 46, 10, 45] ;
```

$$list5 := [13, 26, 46, 10, 45]$$

```
> max(list5) ;
Error, (in simpl/max) arguments must be of type algebraic
```

Il faut en effet nécessairement «retirer les crochets», c'est-à-dire convertir liste vers suite.

```
> max(op(list5)) ;
```

$$46$$

Plus explicitement :

```
> max([13, 26, 46, 10, 45]) ;
Error, (in simpl/max) arguments must be of type algebraic
> max(13, 26, 46, 10, 45) ;
```

$$46$$

On peut aussi passer comme premier argument à la procédure `op` un objet **range** indiquant qu'on veut comme résultat la *suite* des opérandes couverts par le **range** :

```
> op(2..3, list4) ;
```

$$x^2 + y^2, \pi$$

```
> op(3..2, list4) ;
> op(-3..3,list4) ;
```

$$x^2 + y^2, \pi$$

```
> op(0..nops(list4), list4) ;
```

$$list, 27, x^2 + y^2, \pi, \int_1^2 x^3\,dx$$

La procédure `subsop` est logiquement de la même famille que les procédures `op` et `nops`, car il s'agit de traitement assez fin d'opérandes. Nous préférons toutefois reporter son étude Section 4.5.3 dans un autre contexte, quand il s'agit de *modifier* composante par composante un objet déjà existant.

4.4 Suites d'arguments.

Les suites sont aussi très utilisées dans les appels de procédures, le plus souvent sans que l'utilisateur en ait même conscience. La syntaxe d'un appel de procédure est en effet `proc(arg1, ..., argn)` où `proc` est un objet dont la valeur est une procédure, celle qu'on veut *invoquer* ; les arguments (provisoires) `arg1, ..., argn` sont séparés par virgule et constituent bien *une* suite, qui est en fait l'unique argument vraiment passé à la procédure. On réexaminera cette question Chapitre ■■■. Disons tout de suite que ceci permet des appels sophistiqués, où on prépare patiemment la suite d'arguments nécessaires. Un point à ce propos est important : l'opérateur virgule opère non seulement sur les objets quelconques, mais aussi sur les suites :

```
> seq12 := 1, 2 ; seq456 := 4, 5, 6 ;
```

$$seq12 := 1, 2$$

$$seq456 := 4, 5, 6$$

```
> seq16 := seq12, 3, seq456 ;
```

$$seq16 := 1, 2, 3, 4, 5, 6$$

On voit que chaque symbole pointant vers une suite a été remplacé par la suite en question ; autrement dit l'opérateur «virgule» est en fait un opérateur de *concaténation* entre suites, où, cas le plus fréquent, tout opérande qui n'est pas une suite est remplacé par la suite constituée de cet unique élément. Ceci *semble* contredire le warning expliquant qu'un élément de suite ne peut pas être une suite, mais justement, l'*évaluateur* tient compte de notre warning et en quelque sorte «enlève» les parenthèses implicites que vous croyez peut-être voir ; les concepteurs de Maple auraient aussi bien pu décider ici que la valeur du symbole `seq16` est une suite à trois éléments où le premier et le troisième auraient été des suites, mais ce n'est pas le choix qui a été fait ; ils ont décidé que si un argument de l'opérateur virgule est une suite, alors la suite argument est *concaténée* aux suites adjacentes. C'est souvent commode, mais nécessite d'être bien compris. Aucun parenthésage, même intensif, ne changera ceci :

```
> seq16bis := (((seq12), 3), (((seq456)))) ;
```

$$seq16bis := 1, 2, 3, 4, 5, 6$$

À comparer avec :

```
> result := (((1 + 2) + 3) + (((4 + 5 + 6)))) ;
```

$$result := 21$$

où le parenthésage intensif n'a évidemment pas empêché l'addition de travailler.

C'est une conséquence d'Eval-13 ; quand une expression est parenthésée, elle est évaluée séparément, puis devient un composant de suite ; de sorte que si ce composant est lui-même une suite, la suite en question s'allonge, ou encore se raccourcit si notre composant suite est la suite vide ! Ci-dessus l'évaluation de `(seq12)` retourne la même valeur que celle de `seq12` et le parenthésage pour tenter d'isoler la suite résultat est inopérant.

Considérons maintenant cet exemple :

```
> max(2, 4, 3) ;
```

$$4$$

La procédure `max` travaille sur un nombre quelconque d'arguments, et retourne dans le cas réel ce qu'on pense. D'où :

```
> max(seq12, 3, seq456) ;
```

$$6$$

ce qui dans certains contextes peut être assez commode. Cette technique est par

exemple bien utile quand vous avez décidé une fois pour toutes d'utiliser les procédures de tracé graphique avec les options couleur, épaisseur du tracé et type de tracé ci-dessous :

```
> plot(sin(x), x=-Pi..+Pi, color=blue, thickness=2, linestyle=4) :
```

On ne montre pas ce tracé, hors sujet ici. Mais l'usage répété des mêmes options peut être abrégé par la définition de la *suite* de trois équations affectée au symbole `my_options` :

```
my_options := color=blue, thickness=2, linestyle=4 ;
```

$$my\_options := color = blue, thickness = 2, linestyle = 4$$

Moyennant quoi vous pouvez entrer maintenant plus simplement :

```
> plot(sin(x), x=-Pi..+Pi, my_options) :
```

Ce qui semble être l'unique argument `my_options` est «développé» en trois arguments, chacun d'eux étant un objet **equation**.

On est maintenant enfin capable de comprendre ce qu'on était bien en peine d'expliquer page 19. Reprenons l'expression qui était évaluée :

```
> int((1+x/(1+x^2+x^4), x=1..infinity)) ;
```

$$\infty$$

L'analyse du *lecteur* Maple est la suivante : il s'agit d'évaluer un objet **function** à *un seul* opérande ; l'opérateur est le symbole `int`, et, à cause du parenthésage, l'unique opérande, est la *suite* «`(1+x/(1+x^2+x^4), x=1..infinity)`» ; en effet le parenthésage oblige le lecteur à considérer tout ce texte comme un unique objet, ce qui est possible, l'objet en question étant une suite à deux éléments.

Ces préparatifs étant faits par le lecteur, l'évaluateur est appelé. La procédure `int` est standard et l'unique argument *provisoire* suite est évalué ; son évaluation est triviale et on est donc dans une situation où l'unique argument initial est à remplacer par *deux* arguments définitifs. La procédure `int` est donc finalement invoquée comme suit :

```
> int(1+x/(1+x^2+x^4), x=1..infinity)
```

$$\infty$$

Autrement dit les parenthèses redondantes, fautives pour un profane, ici sont parfaitement légales et n'ont provoqué aucune erreur. Mais la fonction à intégrer, compte tenu de la priorité supérieure de la division sur l'addition, est la fonction :

$$1 + \frac{x}{1 + x^2 + x^4}$$

et l'intégrale est donc divergente ; alors que l'utilisateur souhaitait en fait intégrer :

$$\frac{1 + x}{1 + x^2 + x^4}$$

La première parenthèse fermante trop à droite a impliqué l'erreur de fonction, sans provoquer d'erreur de Maple.

4.4.1 Avantages et inconvénients.

Rien n'est bien compliqué dans les explications qui précèdent, l'ensemble est cohérent et facile à utiliser. Il y a cependant quelques conséquences assez cocasses qu'il faut connaître.

Warning 22 — *Les procédures d'analyse* `op`, `nops` *et* `type` *ne peuvent être utilisées pour analyser les suites.*

Reprenons par exemple notre suite `seq456`, et tentons d'en déterminer le nombre d'opérandes :

```
> seq456 ;
```

$$4, 5, 6$$

```
> nops(seq456) ;
Error, wrong number (or type) of parameters in function nops
```

alors que le profane attend évidemment le résultat 3. Mais réfléchissez aux conséquences de ce qui a été expliqué plus haut ; ce que l'utilisateur croit voir comme un *unique* argument est transformé par le mécanisme de préévaluation Eval-14 en une suite à trois éléments 4, 5 et 6. De sorte que notre appel équivaut à :

```
> nops(4, 5, 6) ;
Error, wrong number (or type) of parameters in function nops
```

qui provoque l'erreur constatée, puisque la procédure `nops` ne doit être appelée qu'avec un seul argument. On aurait préféré ici que notre argument évolue en un *unique* argument suite qui aurait donc pu être traité de façon cohérente par la procédure `nops`, mais on perdrait alors la possibilité montrée plus haut de réunir en un seul argument `my_options` les trois arguments de la procédure `plot` qu'on ne souhaite pas répéter à chaque appel. Comme toujours, l'essentiel est de savoir élucider le cas échéant comment le moteur Maple va en définitive fonctionner dans tel ou tel cas.

Le même phénomène va engendrer des anomalies inévitables avec les procédures `op` et `type`.

```
> op(seq456) ;
Error, wrong number (or type) of parameters in function op
```

Plus vicieux :

```
> op(seq12) ;
```

$$2$$

car :

```
> op(0..1, 2) ;
```

$$integer, 2$$

Il est clair maintenant que la procédure `type` va être sérieusement en difficulté avec les suites :

```
> type(seq456, integer) ;
Error, wrong number (or type) of parameters in function type
```

car ceci équivaut à `type(4, 5, 6, integer)`, or la procédure `type` ne travaille qu'avec deux arguments. Encore plus cocasse :

```
> whattype(seq456) ;
```

$$exprseq$$

```
> type(1, exprseq) ;
Error, type `exprseq` does not exist
```

La procédure `whattype` examine en effet d'abord le nombre d'arguments passés. S'il est différent de 1, elle en déduit qu'après tout la *suite* d'arguments peut être considérée comme... *une* suite. Sinon, elle examine le type de l'unique objet argument. Dans le même esprit, la procédure `type` aurait dû retourner *true* si l'appel a un nombre d'arguments différent de 2 et si le type demandé est **exprseq**. Au lieu de quoi, la procédure `type` retourne une erreur dans un tel cas ; le défaut de cohérence est évident, le plus étonnant consistant à mentir insolemment en déclarant que le type **exprseq** n'existe pas, alors qu'il est d'une certaine façon le constituant essentiel de Maple !

4.5 Listes et ensembles, fin.

Le rôle assez particulier des suites est maintenant couvert. Listes et ensembles ne posent dès lors plus de problème. Ne pas se laisser impressionner par l'apparence exotique des opérateurs liste : $[\ldots]$ et ensemble : $\{\ldots\}$; ce sont des opérateurs à la fois pré-et-post-fixés, c'est tout ; les procédures correspondantes sont standard et évaluent leurs arguments avant de les utiliser :

```
> a := b ; c := d ; list2 := [{a, c}, [a, c]] ;
```

$$a := b$$

$$c := d$$

$$list2 := [\{b, d\}, [b, d]]$$

On voit en particulier qu'une liste ou un ensemble peut être un élément de liste. Supposons qu'on veuille convertir la liste `list2` en un ensemble avec les mêmes éléments ; la méthode style ingénieur système consiste à «retirer» les crochets, autrement dit à extraire les opérandes à l'aide de la procédure `op`, puis à «ajouter» les accolades nécessaires pour construire l'ensmble souhaité.

```
> set2 := {op(list2)} ;
```

$$set2 := \{\{b, d\}, [b, d]\}$$

L'application simple de ce qui a été expliqué précédemment montre que le résultat doit être atteint et c'est ce qu'on observe. Ceci dit il est sensiblement plus lisible d'utiliser la procédure `convert`, capable de beaucoup de conversions, en particulier de liste vers ensemble.

```
> set2bis := convert(list2, set) ;
```

$$set2bis := \{\{b, d\}, [b, d]\}$$

La conversion inverse peut aussi être obtenue de l'une ou l'autre façon. Par contre la procédure `convert` ne sait pas réaliser les conversions vers une suite ou d'une suite. Le second cas est logique puisqu'on buterait à nouveau sur le problème de l'argument suite développé le plus souvent en plus d'un argument. On peut supposer que pour respecter une symétrie tentante, les concepteurs Maple n'ont pas souhaité proposer la conversion inverse, qui ne poserait en fait aucun problème.

```
> convert(list2, exprseq) ;
Error, unable to convert
> convert(op(list2), list) ;
Error, wrong number (or type) of parameters in function convert
```

La deuxième conversion est délicate à gérer en raison du nombre d'arguments variable en fonction de la suite à convertir ; par contre la première est facilement réalisable et peut être ajoutée à votre environnement comme suit :

```
> `convert/exprseq` := proc(obj)
    RETURN(op(obj))
    end :
> convert(list2, exprseq) ;
```

$$\{b, d\}, [b, d]$$

car une procédure peut parfaitement retourner une suite, ce qui est souvent commode. Il est fréquent que des procédures Maple de portée assez générale peuvent être étendues par l'utilisateur pour couvrir tel ou tel cas non prévu par les concepteurs Maple. On avait déjà vu cette technique pour la procédure `type` page 77. Le cas de la procédure `convert` ici est analogue. L'utilisateur peut alors définir une procédure `` `mapleproc/othercase` `` où `mapleproc` est une procédure Maple prédéfinie, et `othercase` est un symbole expliquant que si tel argument d'un appel de `mapleproc` est `othercase`, alors l'appel doit être traité comme expliqué dans le corps de la procédure introduite. Pour le cas ajouté `exprseq` de `convert`, on a simplement expliqué qu'il fallait appliquer à l'objet à convertir la procédure `op`. Noter que la présence du slash «/» dans le nom de la procédure impose les backquotes encadrant le symbole, sous peine d'erreur. Il faut comprendre que le nom de la nouvelle procédure est un peu exotique pour dissuader l'utilisateur d'utiliser *directement* cette procédure ; c'est en fait faisable, mais sans intérêt :

```
> `convert/exprseq`(list2) ;
```

$$\{b, d\}, [b, d]$$

On comprend maintenant pourquoi un *élément* de suite, liste ou ensemble ne peut pas être une suite ; l'exemple qui suit est une variante de ce qui a été fait plus haut et illustre ce point :

```
> list_seq12_3_seq456 := [seq12, 3, seq456] ;
```

$$list\_seq12\_3\_seq456 := [1, 2, 3, 4, 5, 6]$$

On a tenté de construire une liste à trois éléments où le premier et le dernier seraient des *suites* ; mais l'opérateur virgule *concatène* les suites, de sorte que le résultat est en fait une liste dont le corps est *une* suite à six éléments. C'est ce qu'exprime entre autres le Warning-21 : un élément de suite ne peut pas être une suite ; puisque le *corps* d'une liste ou un ensemble est justement une suite, il en est de même aussi pour les listes et ensembles.

4.5.1 Lecture indexée des éléments de suites, listes et ensembles.

Reprenons notre première liste exercice `list4`.

```
> list4 := [a, b, c, d] ;
```

$$list4 := [a, b, c, d]$$

On a vu Section 4.3 comment utiliser les procédures `op` et `nops`, pour extraire un ou plusieurs éléments de listes ou ensembles.

```
> op(2, list4) ;
```

$$b$$

```
> op(2..3, list4) ;
```

$$b, c$$

Le même résultat est obtenu à l'aide d'une notation *indexée*, sensiblement plus lisible :

```
> list4[2] ;
```

$$b$$

```
> list4[2..3] ;
```

$$[b, c]$$

```
> op(2..3, list4) ;
```

$$b, c$$

On note toutefois une différence : si l'indice est un **range**, l'expression indexée retourne une *sous-liste* alors que l'expression utilisant la procédure `op` retourne une *suite*, comme il se doit compte tenu de la portée très générale de cette procédure. Les indices négatifs peuvent aussi être utilisés, la numérotation commençant alors

par la fin ;

```
> list4[-3] ;
```

$$b$$

```
> list4[-3..3] ;
```

$$[b, c]$$

Ce qui est illustré ici sur les listes s'applique exactement de la même façon pour les ensembles. Mais, et ceci est important, contrairement à la procédure `op` qui est inutilisable pour les suites, l'extraction indexée d'éléments de suite travaille comme on le souhaite.

```
> seq4 := a, b, c, d ;
```

$$seq4 := a, b, c, d$$

```
> seq4[2] ;
```

$$b$$

```
> seq4[-3..3] ;
```

$$b, c$$

4.5.2 Ecriture indexée dans une liste.

Dans le même esprit que dans la section précédente, on peut *modifier* un élément de liste par une instruction d'affectation citant la référence de l'élément à modifier.

```
> list4[2] := bb ;
```

$$list4_{2} := bb$$

```
> list4[-2] := cc ;
```

$$list4_{-2} := cc$$

```
> list4 ;
```

$$[a, bb, cc, d]$$

Ceci n'est utilisable que pour les *listes*, ne fonctionne donc pas pour les ensembles et les suites. Ne pas essayer non plus la modification de sous-liste.

```
> list4[2..3] := [bbb, ccc] ;
Error, invalid subscript on left hand side of assignment
```

Il faut toutefois être assez méfiant dans ces instructions de modification de liste. Compte tenu de ce qui vient d'être expliqué, le piège suivant fonctionne presque à coup sûr :

```
> list32 := [[a,aa], [b, bb], [c, cc]] ;
```

$$list32 := [[a, aa], [b, bb], [c, cc]]$$

Il peut se faire qu'au cours d'une session, on ait éprouvé la nécessité d'isoler le deuxième élément, qui est aussi une liste (cf. Warning-21) :

```
> pair2 := list32[2] ;
```
$$pair2 := [b, bb]$$

puis on peut avoir une raison de modifier par exemple le second élément de cette liste :

```
> pair2[2] := bbb ;
```
$$pair2_2 := bbb$$
```
> pair2 ;
```
$$[b, bbb]$$

On a pris soin de vérifier, par examen de la valeur de `pair2` que la modification a bien été réalisée ; oui, mais :

```
> list32 ;
```
$$[[a, aa], [b, bb], [c, cc]]$$

et on voit que la valeur de `list32`, elle, est inchangée ! Dans un exercice d'étudiant comme ici, cette étrangeté est identifiable, mais si un tel évènement se produit pendant la mise au point d'un programme complexe, le bug résultant peut être extrêmement difficile à expliquer ! Une analyse soigneuse mais délicate à coups de procédures système `addressof`, `pointto` et `disassemble`, montre que la modification de l'élément `pair2[2]` *modifie* aussi la valeur, plus précisément l'adresse de la valeur de `pair2` ; une *nouvelle* liste est créée, avec les pointeurs ad hoc, mais comme au contraire rien n'est changé pour la valeur de `list32`, la liste valeur de ce dernier symbole est inchangée !

Comparer avec :

```
> list32 := [[a, aa], [b, bb], [c, cc]] ;
```
$$list32 := [[a, aa], [b, bb], [c, cc]]$$
```
> pair2 := list32[2] ;
```
$$pair2 := [b, bb]$$
```
> list32[2,2] := bbb ;
```
$$list32_{2,2} := bbb$$
```
> list32, pair2 ;
```
$$[[a, aa], [b, bbb], [c, cc]], [b, bb]$$

Cette fois c'est la sous-liste repérée par `pair2` qui n'est pas modifiée.

Warning 23 — *La modification d'un terme de liste ne modifie que* la *liste explicitement indiquée comme devant être modifiée ; les listes apparemment «partagées» ne sont pas modifiées.*

Noter aussi que les deux instructions suivantes sont parfaitement équivalentes, pour la lecture de l'élément de liste comme pour sa modification :

```
> list32[2, 2], list32[2][2] ;
```

$$bbb, bbb$$

Dans le registre *surprises associées aux modifications de listes*, il faut aussi signaler ce qui suit. Construisons une liste de 100 entiers quelconques et une autre de 101 ; on cache le résultat, de peu d'intérêt.

```
 > list100 := [0 $ 100] :
> list101 := [0 $ 101] :
```

Modifions dans l'une et l'autre le premier élément :

```
> list100[1] := 1 ; list101[1] := 1 ;
```

$$list100_1 := 1$$

```
Error, assigning to a long list, please use arrays
```

La modification est acceptée pour la première liste, refusée pour la deuxième. On déduit qu'une liste est réputée *longue* si elle a plus de 100 éléments. Ceci est a priori surprenant, car les circonstances sont nombreuses, en simulation notamment, où on peut souhaiter faire évoluer une liste, longue ou pas, pendant une session. Il a été expliqué un peu plus haut pourquoi ces modifications sont en fait très coûteuses, vu l'implémentation adoptée : *chaque* modification d'*un* élément de liste demande la reconstruction d'une liste *entière* de toutes les adresses d'éléments de listes ! Cette méthode très primitive par rapport à ce qui est usuel en traitement de liste fait craindre aux concepteurs de Maple que des usages répétés intempestifs de cette possibilité saturent rapidement la mémoire allouée à Maple ; les dits concepteurs ont donc prudemment décidé que ceci ne serait autorisé que pour les listes «raisonnablement» courtes ; cette dernière notion est heuristique, et la barre a été placée à 100, pourquoi pas. Voir page 160 un exemple frappant pour la comparaison de ce point de vue entre les techhniques *liste* et *vecteur*.

Warning 24 — *Les modifications de listes Maple sont très coûteuses en mémoire ; elles sont déconseillées dès qu'elles sont fréquentes ou concernent des listes relativement longues ; elles sont de toute façon exclues si la longueur de liste est* > 100. *Le cas échéant, il faut préférer, selon les circonstances, un type de donnée* vector *ou* table.

Si la «liste» évolutive est de longueur constante, la solution vecteur est la plus efficace :

```
> vect100 := vector(100, [0 $ 100]) :
```

Si la liste est de longueur variable mais avec longueur maximale connue d'avance, la même solution est utilisable, avec des conventions appropriées pour marquer les cases en fait vides :

```
 > vect10 := vector(10, [seq(i^2, i=1..5), nil $ 5]) ;
```

$$vect10 := [1, 4, 9, 16, 25, nil, nil, nil, nil, nil]$$

```
> for i from 6 to 8 do
    vect10[i] := i^3
    od :
> vect10 ; eval(vect10) ;
```

$$vect10$$

$$[1, 4, 9, 16, 25, 216, 343, 512, nil, nil]$$

etc. Voir Section 4.9.1 pour les détails, en particulier pour comprendre pourquoi le vecteur est *affiché* comme une liste, mais n'est pourtant en aucune façon une liste. Pour des situations beaucoup plus imprévisibles, il faut utiliser des *tables*, voir Section 4.7.

4.5.3 La procédure subsop.

Puisque la section précédente nous a fait toucher du doigt les difficultés potentielles des problèmes de modification, le moment est approprié d'étudier la procédure `subsop` qui, d'un certain point de vue, est un peu de la même famille, mais d'un autre est assez différente. L'exemple élémentaire suivant explique le principe de base.

```
> expr := a^2 + b^2 + c^2 ;
```

$$expr := a^2 + b^2 + c^2$$

```
> op(expr) ;
```

$$a^2, b^2, c^2$$

```
> subsop(2 = bb^3, expr) ;
```

$$a^2 + bb^3 + c^2$$

La procédure `subsop` utilise deux arguments ; le premier est une équation où le membre de gauche est un numéro d'opérande à *modifier*, et le membre de droite est le (ou les) nouvel(eaux) opérande(s) qu'on veut voir figurer en place de l'opérande désigné par son numéro ; enfin le second argument de `subs` est l'objet où la substitution est souhaitée. Ici, on a remplacé le deuxième opérande (donc la sous-expression `b^2`) par l'expression `bb^3`. Bien noter que ceci décrit un procédé de construction d'un *nouvel* objet, mais que l'ancien est inchangé :

```
> expr ;
```

$$a^2 + b^2 + c^2$$

Dans une telle substitution, le nombre d'opérandes peut être augmenté :

```
> subsop(2 = bb^2 + bbb^2, expr) ;
```

$$a^2 + bb^2 + bbb^2 + c^2$$

Une certaine cohérence peut être nécessaire dans la substitution, sous peine d'erreur :

```
> subsop(2 = (bb^2, bbb^2), expr) ;
Error, invalid types in sum
```

puisqu'un terme d'une somme ne peut pas être une *suite* ; ici le parenthésage de la suite membre de droite de l'équation premier argument évitait toute confusion sur la nature des arguments ; comparer avec :

```
> subsop(2 = bb^2, bbb^2, expr) ;
Error, wrong number (or type) of parameters in function subsop
```

puisque le non-parenthésage implique cette fois trois arguments pour l'appel de `subsop` ; ceci n'est pas illégal en soi, voir la documentation pour diverses variantes de `subsop`, mais ici le second argument est incorrect ; il devrait être de la forme **integer** = ...

Par contre un terme de somme peut très bien être une *liste* :

```
> subsop(2 = [bb^2, bbb^2], expr) ;
```

$$a^2 + [bb^2, bbb^2] + c^2$$

Dans certains contextes, ceci pourrait avoir un sens, par exemple dans un contexte vectoriel.

Revenons au sujet principal de cette section, à savoir les listes et ensembles. La procédure `subsop` peut être utilisée pour des manipulations conduisant à intercaler ou à supprimer un terme de liste ou d'ensemble, car cette fois, l'objet à substituer peut être une *suite* avec un nombre quelconque d'éléments.

```
> list4 := [a, b, c, d] ;
```

$$list4 := [a, b, c, d]$$

```
> subsop(2 = (b, bb, bbb), list4) ;
```

$$[a, b, bb, bbb, c, d]$$

```
> subsop(2 = NULL, list4) ;
```

$$[a, c, d]$$

On verra, voir Section LLL, que la procédure `subs` permet elle aussi de nombreuses *constructions* de même nature, par description de l'objet à construire à partir d'un objet déjà existant modulo quelques changements.

4.6 Les constructeurs seq et $.

Cette section est assez délicate et pour éviter toute interférence fâcheuse avec ce qui précède, il est préférable de remettre les compteurs à zéro.

```
> restart ;
```

L'importance des *suites* dans la structure des objets Maple commence à s'imposer. Il est donc important de disposer de constructeurs commodes pour créer des objets où les suites sont des constituants essentiels ; l'opérateur «virgule» permet de

construire les suites élément par élément, mais il serait fort pénible de construire ainsi par exemple la suite des entiers de 1 à 1000. La procédure `seq` est idéale pour ce faire. Elle utilise une *variable pilote*, un symbole[2]. La procédure `seq` nécessite deux arguments ; le premier est un *terme générique*, un objet Maple quelconque où figure le plus souvent la variable pilote ; le second argument est un objet **equation** dont le membre de gauche doit être la variable pilote, et le membre de droite est le plus souvent un **range**, c'est-à-dire deux expressions qui seront évaluées, séparées par l'opérateur «..». Par exemple pour construire la suite des cubes des entiers de 10 à 20 on peut procéder comme suit :

```
> CubeSeq := seq(i^3, i=10..20) ;
```

$$CubeSeq := 1000, 1331, 1728, 2197, 2744, 3375, 4096, 4913, 5832, 6859, 8000$$

Dans cet appel de la procédure `seq`, la variable pilote est le symbole `i` ; le terme générique est l'objet Maple `i^3` ; le range décrivant le domaine de variation de la variable pilote est l'objet «`10..20`». Si le deuxième membre de l'équation deuxième argument n'est pas un objet **range**, il est interprété d'une façon un peu différente ; dans un tel cas, la variable pilote va parcourir la suite des *opérandes* de ce second membre. Exemple typique :

```
> seq(item^3, item = a^2 + b^2 + c^2) ;
```

$$a^6, b^6, c^6$$

Considérons maintenant l'opérateur «`$`». Il a l'air tès semblable à la procédure `seq` ; par exemple notre construction de suite de cubes peut être obtenue à l'aide de l'opérateur «`$`» comme suit :

```
> CubeSeq2 := i^3 $ i=10..20 ;
```

$$CubeSeq2 := 1000, 1331, 1728, 2197, 2744, 3375, 4096, 4913, 5832, 6859, 8000$$

On voit que les deux appels sont analogues à ceci près que pour la procédure `seq`, le nom de procédure est *préfixe*, précédant les deux arguments parenthésés, alors que pour la procédure «`$`», l'opérateur est *infixé*, c'est-à-dire inséré *entre* les deux opérandes. C'est vrai, mais il y a une autre différence beaucoup plus importante, invisible sur notre exemple simple : le mécanisme d'évaluation des arguments est complètement différent.

Eval 25 — *L'évaluation d'une expression «*`expr1 $ expr2`*» procède comme suit. L'argument* `expr1` *est d'abord évalué ; soit* `expr11` *l'objet résultat de l'évaluation ; l'argument* `expr2` *est ensuite évalué et le résultat de cette évaluation doit être une équation* `symb = n1..n2` *où* `symb` *est un symbole,* `n1` *et* `n2` *sont deux entiers. Alors l'expression* `expr11` *est réévaluée dans un contexte où le symbole* `symb` *pointe successivement vers les entiers* `n1`, `n1+1`, ..., `n2`. *Les* `(n2-n1+1)` *résultats successifs sont les composants de la suite résultat.*

Construisons une procédure espion pour mettre en évidence les différentes

[2] Ou plus généralement un *nom* ; autrement dit une variable indicée peut aussi être utilisée, mais ceci n'est que très rarement utilisé.

étapes d'une évaluation.

```
> spy := proc(arg)
    printf("Evaluating spy(%a)\n", arg) ;
    RETURN(arg)
    end :
```

La procédure `spy`, quand elle est appelée, affiche un message signalant l'appel, permettant aussi de voir quel est l'argument *définitif*[3] qui est passé ; de plus la procédure retourne seulement cet argument.

```
> spy('message') ;
Evaluating spy(message)
```

message

Notre procédure permet donc de *tracer* certaines étapes du fonctionnement de l'évaluateur ; considérons par exemple :

```
> spy('spy(i^2)') $ spy(i)=spy(2)..spy(3) ;
Evaluating spy(spy(i^2))
Evaluating spy(i)
Evaluating spy(2)
Evaluating spy(3)
Evaluating spy(4)
Evaluating spy(9)
```

4, 9

On voit que l'évaluateur a bien travaillé ! Le premier `spy('spy(i^2)')` est d'abord évalué et retourne `spy(i^2)` ; les trois évaluations suivantes construisent le second argument de l'expression «`$`» qui va donc être «`i = 2..3`». On voit ensuite que `spy(i^2)` est évalué deux fois, la première fois dans un environnement où `[i` $\rightarrow$ `2]`, la seconde dans un environnement où `[i` $\rightarrow$ `3]`, de sorte que les arguments définitifs de `spy` sont respectivement 4 et 9. D'où le résultat final. La procédure «`$`» *ne dispose pas* de la même possibilité qui avait été signalée pour la procédure `seq`, permettant de faire décrire à la variable pilote les opérandes d'un objet :

```
> i^2 $ i = a^2+b^2+c^2 ;
Error, wrong number (or type) of parameters in function $
```

Espionnons maintenant la variante `seq` de l'appel sophistiqué ci-dessus de la procédure «`$`» :

```
> seq(spy('spy(i^2)'), spy(i) = spy(2)..spy(3)) ;
Evaluating spy(2)
Evaluating spy(3)
Error, unable to execute seq
```

On voit que Maple procède d'une façon complètement différente. La procédure `seq` *n'évalue pas* d'abord son premier argument et ce n'est donc pas une procédure

[3] Voir page 61.

standard : elle ne respecte pas Eval-14. On verra page ▙▟ comment l'utilisateur peut lui aussi préparer des procédures non standard. La procédure `seq` commence par vérifier que le deuxième argument est une équation, sans quoi elle arrête immédiatement.

```
> seq(spy(a), spy(b)) ;
Error, wrong number (or type) of parameters in function seq
```

Si par contre le second argument est une équation, la procédure `seq` évalue d'abord son second membre, puis vérifie que le premier est un symbole[4], sinon erreur :

```
> seq(spy(a), spy(b) = spy(c)) ;
Evaluating spy(c)
Error, unable to execute seq
```

C'est ce qui avait provoqué l'erreur dans l'exemple initial, car :

```
> type('spy(i)', name) ;
```

false

Noter la nécessité de quoter le premier argument car la procédure `type` au contraire est standard et évalue donc ses arguments avant de les utiliser :

```
> type(spy(i), name) ;
Evaluating spy(i)
```

true

Comme `spy(i)` est évalué, l'argument définitif est cette fois le symbole `i` qui est bien un objet **name** ! L'expérience suivante ne peut être que :

```
> seq(spy('spy(i^2)'), i = spy(2)..spy(3)) ;
Evaluating spy(2)
Evaluating spy(3)
Evaluating spy(spy(i^2))
Evaluating spy(spy(i^2))
```

$$\mathrm{spy}(i^2), \mathrm{spy}(i^2)$$

Encore une surprise, mais facile à expliquer. Une fois que le second argument de `seq` est évalué, le premier est donc maintenant, et maintenant seulement, évalué deux fois, une fois dans un environnement où `[i` $\rightarrow$ `2]`, une autre fois avec `[i` $\rightarrow$ `3]`. Mais puisque l'unique occurence du symbole `i` est entre quotes, ces valeurs de `i` sont inutilisées. L'argument définitif de `spy` est l'expression `spy(i^2)`, qui ne sera plus réévaluée, d'où le résultat sous forme de deux exemplaires de cette expression, un pour chaque valeur de `i` demandée. La dernière expérience est donc :

[4] Plus généralement un objet **name**.

```
> seq(spy(i^2), i = spy(2..3)) ;
Evaluating spy(2..3)
Evaluating spy(4)
Evaluating spy(9)
```

4, 9

qui nous donne la chronologie parfaite des évaluations réalisées.

Eval 26 — *Un appel* `seq(expr1, symb=expr2)` *est évalué comme suit. L'objet* `expr2` *est d'abord évalué, soit* `expr22` *l'objet retourné par l'évaluateur ; le domaine que doit parcourir la* variable pilote `symb` *est la suite des* opérandes *de* `expr22`*, à moins que cet objet soit un objet* **range** *auquel cas le domaine sera l'ensemble des entiers décrit par ce* **range**. *Ensuite l'objet* `expr1` *est évalué autant de fois que nécessaire dans un environnement où le symbole* `symb` *pointe successivement vers les différents éléments du domaine déduit de* `expr22`. *Le résultat de l'évaluation sera la* suite *des résultats d'évaluation obtenus.*

Conseil 27 — *D'une façon générale, la procédure* `seq` *est sensiblement plus adaptée à la construction des suites que l'opérateur «* `$` *».*

Pour illustrer ce conseil, considérons l'exemple prototype suivant. Soit à construire une suite de 10 entiers aléatoires du **range** `1..100`. Pour construire *un* tel entier, on peut appeler la procédure `rand` :

```
> rand(1..100)() ;
```

82

L'appel `rand(1..100)` construit un *générateur* d'entiers aléatoires, sans argument, qu'on *appelle* donc par `rand(1..100)()`.

Puisqu'on s'apprête à appeler de façon répétitive la procédure *construite* par `rand(1..100)`, il est bien préférable de la créer une fois pour toutes et de l'affecter à un symbole approprié. Pour des exercices d'étudiants comme ici, ce point est relativement secondaire, mais si vous comptez un jour faire de vrais calculs avec Maple, notamment de vraies simulations avec de vastes générations de nombres aléatoires, cette précaution peut être *capitale* et il n'est jamais trop tôt pour acquérir le bon réflexe. On refait donc l'exemple précédent comme suit :

```
> rnd100 := rand(1..100) ;
             rnd100 := proc()
                       local t; global _seed;
                       _seed := irem(427419669081 * _seed, 999999999989);
                       t := _seed;
                       irem(t, 100) + 1
                       end
> rnd100() ;
```

71

Pour une fois, on a montré le résultat procédure[5] pour mettre en évidence que la

[5] Habituellement affiché sur une ligne dans une fenêtre 17».

procédure rand retourne effectivement une *procédure*.

Si donc on veut répéter dix fois cet appel pour obtenir une suite de dix entiers aléatoires, avec l'opérateur «`$`», on procède a priori comme suit :

```
> rnd100() $ i=1..10 ;
```

$$64, 64, 64, 64, 64, 64, 64, 64, 64, 64$$

Ce n'est pas du tout le résultat souhaité ! On voit ici que le luxe de détails précédemment donnés dans cette section est loin d'être superflu. L'expression `rnd100()` est *d'abord* évaluée, «choisit» donc *un* nombre aléatoire, ici 64 ; c'est l'objet noté `expr11` dans Eval-25. Puis le deuxième opérande de l'expression «`$`» est traité. Le *résultat* de la première évaluation, donc l'entier 64, est alors évalué autant de fois que nécessaire, donc 10 fois, dans un environnement où le symbole `i` pointe vers 1, puis vers 2, etc. Mais quel que soit l'environnement, le résultat de l'évaluation de 64 est 64, d'où notre résultat. Comparer avec ce qui est obtenu naturellement avec la procédure `seq` :

```
> seq(rnd100(), i=1..10) ;
```

$$77, 39, 86, 69, 22, 10, 56, 64, 58, 61$$

Car cette fois le terme générique n'est évalué qu'une fois pour chacune des valeurs de `i` souhaitées, valeurs qui ne jouent d'ailleurs aucun rôle en elles-mêmes. On obtiendrait un résultat analogue avec l'opérateur «`$`» en *interdisant* la première évaluation :

```
> 'rand(1..100)()' $ 'i'=1..10 ;
```

$$75, 86, 17, 62, 8, 50, 87, 99, 67, 10$$

mais cette technique est à ranger dans les «dirty tricks».

Les seuls cas où l'opérateur «`$`» est franchement commode sont les suivants. Pour la répétition simple d'*un* objet, on n'est pas en fait tenu de prévoir une variable pilote :

```
> a $ 5 ;
```

$$a, a, a, a, a$$

On aurait donc pu créer notre suite d'entiers aléatoires comme suit :

```
> 'rnd100()' $ 10 ;
```

$$74, 82, 75, 67, 74, 43, 92, 94, 1, 12$$

Puisque *chaque* évaluation de `rnd100()` retourne un *nouvel* entier aléatoire, la suite obtenue dans ce cas n'est pas constante, mais c'est justement ce qui est souhaité !

Un autre cas, plus rare, est aussi intéressant :

```
> $ 5..10 ;
```

$$5, 6, 7, 8, 9, 10$$

Noter aussi la difficulté déjà signalée page 58 ; comparer :

```
> for i from 1 to 3 do print(i) od ;
```

$$1$$
$$2$$
$$3$$

```
> i^2 $ i = 1..10 ;
Error, wrong number (or type) of parameters in function \boxtt{\$}
> seq(i^2, i=1..10) ;
```

$$1, 4, 9, 16, 25, 36, 49, 64, 81, 100$$

On avait expliqué page 58 que l'instruction itérative `for` laisse le symbole `i` pointant vers 4, d'où l'erreur dans l'instruction «`$`» car la variable pilote y est évaluée, alors qu'au contraire l'appel de `seq` produit le résultat escompté, car la variable pilote n'y est jamais évaluée. Ceci fonctionne :

```
> 'i^2' $ 'i' = 1..10 ;
```

$$1, 4, 9, 16, 25, 36, 49, 64, 81, 100$$

mais c'est détestable et propice à bien des erreurs dans des situations plus complexes.

4.7 Tables.

La structure d'une table est assez différente de ce qui a été expliqué pour les suites, listes et ensembles. Dans le cas le plus général il s'agit en langage informatique d'un fichier à accès direct ; les *articles* sont des objets Maple quelconques, et chaque article est accessible par une *clé*, un autre objet Maple lui aussi quelconque. D'habitude, on appelle plutôt *indice* ce qui est de fait une clé d'accès. Exemple artificiel :

```
> table4 := table([1=a, 2=b, 3=c, 4=d]) ;
```

table4 := table([

$1 = a$

$2 = b$

$3 = c$

$4 = d$

])

Cette table a quatre articles (*entries* en jargon Maple), à savoir les symboles `a`, `b`, `c` et `d`. Chacun est accessible par sa clé (*indice*), respectivement 1, 2, 3 et 4. Noter que, contrairement à ce qui est fréquemment prévu dans les systèmes de traitement de fichiers, la clé n'est pas une partie de l'article : les objets correspondants, article et clé, sont disjoints, conceptuellement et aussi en machine.

Pour voir l'intégralité par exemple d'une *liste* qui serait pointée par le symbole `list3`, il suffit d'évaluer le symbole `list3` :

```
> list3 := [aa, bb, cc] ;
```

$$list3 := [aa, bb, cc]$$

```
> list3 ;
```

$$[aa, bb, cc]$$

On ne peut demander de la même façon le contenu d'une table pointée par un symbole ; par exemple si on tente de voir le contenu de notre table repérée par le symbole `table4` :

```
> table4 ;
```

$$table4$$

L'explication vient de la règle Eval-16 : un chaînage `[symbol` → `table]` n'est utilisé que dans une évaluation *explicite*, donc seulement si le symbole est en position opérateur, ou encore si on demande explicitement l'évaluation par l'intermédiaire de la procédure `eval` :

```
> eval(table4) ;
table([
   1 = a
   2 = b
   3 = c
   4 = d
   ])
```

Les opérations fondamentales de traitements de fichiers sont :

1. Lecture d'un article, étant donné sa clé ;
2. Modification, on dit plutôt *mise-à-jour*, d'un article, étant donné sa clé et le nouvel article à installer à la place de l'ancien ;
3. Suppression d'un article, étant donné sa clé ;
4. Ajout d'un article étant donné le nouvel article à installer, et la clé qui permettra d'y accéder ; la terminologie standard traitement de fichier appelle ceci une *écriture*.

Il va donc falloir nous intéresser aux méthodes, en fait assez simples, qui permettent de réaliser ces quatre opérations.

4.7.1 Lecture d'un article.

On lit un article de table en indexant la table par la clé de l'article cherché.

```
> table4[2] ;
```

$$b$$

Il est *légal* de citer une clé inexistante, auquel cas l'objet retourné est aussi un objet indexé, type `indexed` :

```
> table4[55] ;
```

$$table4_{55}$$

```
> type(%, indexed) ;
```

$$true$$

Ceci fonctionne même si l'opérateur n'évalue pas vers une table. Exemple artificiel pour expliquer le mécanisme :

```
> a := b ;
```

$$a := b$$

```
> a[a+4] ;
```

$$b_{b+4}$$

```
> eval(b) ;
```

$$b$$

On voit que dans la deuxième instruction, l'opérateur `a` *et* l'indice `a+4` ont été évalués. Le résultat est l'objet de type **indexed** `b[b+4]` (en notation input). Cet objet retourné n'implique pas la création d'une table repérée par `b` comme le démontre la dernière instruction.

Une table peut donc être considérée comme une *extension* de l'environnement. Ainsi la table repérée par le symbole `table4` peut être vue comme un ensemble de quatre chaînages `[table4[1]` → `a]`, `[table4[2]` → `b]`, etc. De ce point de vue, si un indice comme `55` est absent de la table, on peut penser que le nom `table4[55]` est absent de l'environnement, auquel cas le mécanisme d'évaluation triviale expliqué à Eval-7 s'applique, le nom est retourné tel quel.

Comparer avec :

```
> list4 := [a, b, c, d] ;
```

$$[a, b, c, d]$$

```
> list4[55] ;
Error, invalid subscript selector
```

On verra cependant que pour les tableaux, et donc en particulier pour les matrices et les vecteurs, il est nécessaire de choisir *à la création* du tableau l'ensemble des indices valides ; si un indice non valide est ensuite utilisé, il entraîne alors logiquement une erreur. Ce point sera rediscuté Section ▙▟.

4.7.2 Modification ou ajout d'un article.

Ces deux opérations sont traitées de la même façon, par une instruction d'affectation, citant la référence à modifier ou à ajouter. Par exemple on peut modifier l'article associé à la clé 3 :

```
> table4[3] := cc ;
```

$$table4_3 := cc$$

```
> eval(table4) ;
```

table([

$1 = a$

$2 = b$

$3 = cc$

$4 = d$

])

L'affichage des tables entières est un peu envahissant, et on préfèrera désormais examiner plutôt le ou les seuls éléments qui nous intéressent, justement par une opération de lecture :

```
> table4[3] ;
```

$$cc$$

écrivons, c'est-à-dire ajoutons, un élément de clé 5 :

```
> table4[5] := `fifth element` ;
```

$$table4_5 := fifth\ element$$

Le symbole pointant notre table n'a pas changé, mais notre table a maintenant 5 éléments ; l'article ajouté est un symbole «exotique», à cause du caractère espace y figurant, et a nécessité les backquotes d'encadrement. Les «nouvelles» clés ajoutées peuvent être des objets Maple quelconques, idem pour les articles ajoutés.

```
> table4[55] := 5 &+ 11 ;
```

$$table4_{55} := 5\ \&+\ 11$$

Ici l'article ajouté est une expression utilisant un opérateur infixé non prédéfini ; voir à ce sujet la page de documentation intitulée `neutral`.

Rien n'empêche un article d'être une suite :

```
> table4[x^2+y^2] := 1, 2, 3 ;
```

$$table4_{x^2+y^2} := 1, 2, 3$$

Une clé peut aussi être une suite :

```
> table4[1, 2, 3] := {4, 5, 6, 7} ;
```

$$table4_{1,2,3} := \{4, 5, 6, 7\}$$

Notre table est maintenant un objet auquel on serait bien en peine de donner un sens raisonnable, mais ceci est sans importance pour Maple.

```
> eval(table4) ;
```
table([
$x^2 + y^2 = (1, 2, 3)$
$1 = a$
$2 = b$
$3 = cc$
$4 = d$
$5 = \textit{fifth element}$
$55 = 5 \ \&+ \ 11$
$(1, 2, 3) = \{4, 5, 6, 7\}$
])
```
> table4[table4[y^2+x^2]] ;
```
$$\{4, 5, 6, 7\}$$

Bref, on voit que clés et articles peuvent être des objets absolument quelconques, et que les opérations usuelles de comparaison sont utilisées pour identification ; en particulier la référence de clé `x^2+y^2` est synonyme de `y^2+x^2`.

4.7.3 Suppression d'un article.

La suppression d'un article d'une table est apparentée au retrait d'un symbole de l'environnement ; elle est effectuée exactement de la même façon. Deux méthodes sont donc possibles ; la première consiste à demander une «auto-affectation» :

```
> table4[3] := 'table4[3]' ;
```
$$table4_3 := table4_3$$
```
> assigned(table4[3]) ;
```
$$false$$

La deuxième méthode, sensiblement plus lisible, consiste à utiliser la procédure `unassign` ; il faut cependant faire attention à l'évaluation de son argument ; presque toujours il doit être quoté justement pour empêcher cette évaluation.

```
> unassign('table4[1]') ; assigned(table4[1]) ;
```
$$false$$

4.7.4 Questions subtiles d'évaluation autour des objets indexés.

Repartons de zéro et créons une petite table en vue de quelques expériences.

```
> restart ; a := table([11, 22, 33, 44, 55]) ;
```

$$a := \text{table}([$$
$$1 = 11$$
$$2 = 22$$
$$3 = 33$$
$$4 = 44$$
$$5 = 55$$
$$])$$

On observe au passage que si on ne présente pas l'argument de la procédure `table` sous forme d'une liste d'équations, la procédure considère alors que la liste ne contient que les éléments (entries) de la table, et elle affecte d'office des indices entiers en partant de 1. On vous laisse expérimenter ce qui advient si la liste contient des équations et d'autres objets qui ne sont pas des équations.

Il peut arriver alors qu'une variable disons `n` ait pour valeur un entier indice de notre table et qu'on veuille modifier l'élément ayant cet indice. Observons très attentivement l'affectation suivante :

```
> n := 3 ; a[n] := n^2 ;
```

$$n := 3$$
$$a_3 := 9$$

On voit que le comportement de l'évaluateur est ici assez fin : quand il traite l'objet qu'il s'agit d'affecter, il constate que c'est un objet indexé ; il décide alors d'*évaluer l'indice*, transformant ici `n` en `3`, et l'élément de table à affecter est donc en définitive `a[3]` ; bien noter qu'au contraire il *n'évalue pas* `a[3]` qui produirait `33`, mais une affectation à `33`, un entier, n'aurait pas de sens. Dans le même esprit :

```
> a[n^2-n-2] := 444 ; a[4] ;
```

$$a_4 := 444$$
$$444$$

```
> a[a[n+1]/111] := 555 ;
```

$$a_4 := 555$$

Autrement dit, comme il l'avait déjà été observé page 71, il n'y a pas d'évaluation *au niveau supérieur* dans le membre de gauche d'une instruction d'affectation, par contre toute évaluation à un *niveau inférieur*, typiquement d'un indice, est effectuée pour déterminer quel est l'objet source du chaînage à installer ou à modifier. Moyennant quoi vous pouvez comprendre l'exemple artificiel suivant.

```
> b[1] := 11 ; n := 1 ; b[n] := 111 ; b['n'] := nn ; eval(b) ;
```

$$b_1 := 11$$
$$n := 1$$
$$b_1 := 111$$
$$b_n := nn$$

table([
$1 = 111$
$n = nn$
])

Il a fallu en effet quoter le symbole `n` en indice pour empêcher son évaluation afin de créer un élément d'indice `n`.

Dans le même registre la libération d'un élément de table à indice calculé va poser problème. Supposons qu'on veuille libérer l'élément de la table `a` dont l'indice est la valeur de `n`. Ceci ne fonctionne pas :

```
> a[n] := a[n] ;
```

$$a_1 := 11$$

En effet le terme de gauche de l'instruction d'affectation devient comme on le souhaite `a[1]`, car `n` est évalué ; mais le terme de droite est lui *entièrement* évalué produisant la *valeur* de `a[1]`, à savoir `11` et notre affectation ne sert à rien, elle ne fait que répéter la situation déjà existante !

Ceci ne fonctionne pas non plus :

```
> a[n] := 'a[n]' ;
```

$$a_1 := a_n$$

```
> a[1] ;
Error, too many levels of recursion
> eval(a[1], 1) ;
```

$$a_n$$

car on voit qu'on a cette fois «trop» empêché l'évaluation du terme de droite de l'affectation ; ce n'est donc pas une «auto-affectation» qui est demandée, de sorte que Maple installe un chaînage de `a[1]` vers `a[n]`. Provoquant ainsi quand on demande la valeur de `a[1]` une erreur amusante, conséquence curieuse du mécanisme de post-évaluation, voir la règle Eval-13 : la valeur de `a[1]` est `a[n]`, oui, mais `n` a pour valeur `1`, donc l'évaluateur obtient `a[1]`, oui, mais `a[1]` a pour valeur `a[n]`, etc. Comme toujours, `eval(..., 1)` permet de demander une seule phase d'évaluation.

Et on voit qu'on est en panne pour demander l'auto-affectation souhaitée ! Il faut en fait utiliser ici `unassign` ; on semble pourtant buter a priori sur les mêmes difficultés. En particulier `unassign(a[n])` va provoquer la même erreur de récursisivité indéfinie :

```
> unassign(a[n]) ;
Error, too many levels of recursion
```

Et le lecteur lucide pense probablement ici que `unassign('a[n]')` ne fonctionne pas non plus car on demande semble-t-il ainsi la libération de `a[n]` et c'est en fait celle de `a[1]` qui est voulue. Oui, mais ici la procédure `unassign` nous aide bigrement car elle dispose d'un traitement très sophistiqué des objets indicés[6] ; si l'argument *définitif* est un objet indicé, la procédure *évalue encore* l'indice donné avant de réaliser la libération. Preuve :

```
> eval(a[1], 1) ;
```

$$a_n$$

```
> unassign('a[n]') ;
> assigned(a[1]) ;
```

$$false$$

4.7.5 Indices et éléments d'une table.

On peut obtenir la *suite* des clés (resp. articles) actuels d'une table par appel de la procédure `indices` (resp. `entries`) :

```
> restart ; a := table([b, c, d]) ;
```

$a :=$ table([

$1 = b$

$2 = c$

$3 = d$

])

```
> indices(a) ; entries(a) ;
```

$$[1], [2], [3]$$

$$[b], [c], [d]$$

On voit que *chaque* clé, chaque article, est encadré entre crochets. Cette notation assez lourde pour le résultat est nécessaire pour lever l'ambiguïté qui pourrait résulter du Warning-21 : un élement de table, contrairement à ce qui a été expliqué pour les suites, listes et ensembles, *peut être une suite*. Un indice de table peut aussi être une suite, auquel cas il devient essentiellement un *multi-indice*. Procédons aux expériences rituelles pour examiner ces points.

[6] Par l'intermédiaire de la procédure souterraine `unassign/indexed`

```
> a[1,2,3] := e ;
```

$$a_{1,2,3} := e$$

```
> a[f] := 4, 5, 6 ;
```

$$a_f := 4, 5, 6$$

```
> indices(a) ; entries(a) ;
```

$$[1,2,3], [1], [2], [3], [f]$$
$$[e], [b], [c], [d], [4,5,6]$$

On voit que sans les crochets délimitant les multi-indices ou les éléments suites, on serait incapable d'interpréter correctement les résultats.

Dans le même ordre d'idées, si on initialise une table avec des équations `index = entry`, quand un indice est multiple, il doit être parenthésé, pour lever le même genre d'ambiguïté.

```
> restart ; a := table([1, 2 = 3, 4]) ;
```

$a :=$ table([
$1 = 1$
$2 = (2 = 3)$
$3 = 4$
])

Ici la liste a trois arguments dont deux ne sont pas des équations ; Maple décide donc l'affectation automatique des clés ; en particulier l'élément de clé `2` est l'équation «`2 = 3`» ; la coïncidence clé et premier membre de l'article est à considérer comme fortuite. Comparer avec les trois autres cas complémentaires.

```
> restart ; a := table([(1, 2) = 3, 4]) ;
```

$a :=$ table([
$1 = ((1,2) = 3)$
$2 = 4$
])

```
> restart ; a := table([1, 2 = (3, 4)]) ;
```

$a :=$ table([
$1 = 1$
$2 = (2 = (3,4))$
])

```
> restart ; a := table([(1, 2) = (3, 4)]) ;
```

$a :=$ table([
$(1,2) = (3,4)$
])

```
> a[1,2] ;
```

$$3, 4$$

Rien n'empêche dans une table que certains indices soient multiples et d'autres non, comme il l'avait déjà été constaté plus haut quand on examinait les résultats de la procédure `indices`.

Le lecteur peut aussi être troublé s'il compare les évaluations :

```
> 1, 2 = 3, 4, 5 ;
```
$$1, 2 = 3, 4, 5$$
```
> 1, 2 = (3, 4), 5 ;
```
$$1, 2 = (3, 4), 5$$
```
> 1, 2 = 3, (4, 5) ;
```
$$1, 2 = 3, 4, 5$$
```
> nops([%%]), nops([%]) ;
```
$$3, 4$$

Si on est imprécis, on peut ne pas comprendre pourquoi l'évaluateur garde bien la *suite* «`(3, 4)`» dans le deuxième cas, et qu'au contraire il ne garde pas la suite «`(4, 5)`» dans le troisième. Il faut ici penser au travail du *lecteur* Maple. Dans le deuxième cas, quand le lecteur entame la lecture du deuxième élément de liste, il lit successivement l'entier `2`, le signe «`=`» et l'objet est donc une équation, car l'opérateur-sic «`=`» est en fait un constructeur, qui construit une équation qui restera certainement telle, et enfin la lecture du second membre commence par une parenthèse ; Maple sait qu'il n'aura terminé le traitement du second membre que quand il aura trouvé la parenthèse fermante correspondante ; en définitive le second élément de liste est donc une équation dont le second membre est une liste, et le traitement du deuxième argument est ainsi définitivement terminé.

Dans le dernier cas, ce qui semble être le troisième argument est développé en une suite qui est donc *concaténée* «par» la deuxième virgule aux deux arguments précédents, produisant ainsi une suite de quatre arguments ; la liste construite est donc de longueur 4. Noter qu'on ne peut pas demander directement la longueur d'une suite, d'où la nécessité des crochets encadrant les symboles `%%` et `%` pour déterminer la longueur des deux derniers résultats.

4.8 Tableaux.

Les tableaux sont des tables particulières. Elles ont deux particularités :

1. L'ensemble des indices *possibles* pour leurs éléments est défini une fois pour toutes à la *création* du tableau ;
2. L'ensemble des indices possibles doit être un produit cartésien d'intervalles d'entiers :
$$[i_1..j_1] \times \ldots \times [i_d..j_d].$$

L'entier d est la *dimension* du tableau ; c'est un entier strictement positif. *Chaque* indice est donc *une* suite d'entiers «`k1, ..., kd`» qu'on considère habituellement plutôt comme un multi-indice ; mais en plusieurs circonstances, le seul point de vue qui, strictement parlant, soit *correct* est selui-ci : «l'*unique* indice est une *suite* de d entiers».

Soit par exemple à créer un tableau où la suite d'indices possible est le produit $[0..4] \times [0..4]$. La procédure `array` peut être utilisée comme suit, et le tableau

résultat est affecté au symbole `m5`.

```
> m5 := array(0..4, 0..4) ;
```

$$m5 := \mathrm{array}(0..4, 0..4, [\,])$$

Le tableau étant ainsi créé, ses éléments peuvent maintenant être affectés, modifiés ou supprimés comme pour les tables, à la condition expresse de ne citer qu'un indice autorisé, donc ici une suite de deux entiers du domaine `0..4`. On peut par exemple souhaiter installer dans `m5[i,j]` le produit modulo 5 des deux indices `i` et `j`, pour obtenir la table de multiplication des entiers modulo 5 :

```
> for i from 0 to 4 do
    for j from 0 to 4 do
      m5[i, j] := (i * j) mod 5
      od
    od ;
```

Aucun affichage n'apparaît à l'écran ; ceci est dû au fait que *deux* instructions `for` sont emboîtées, et Maple juge que la profondeur de travail est alors trop importante pour justifier un affichage qui risquerait d'être un peu envahissant[7]. La table `m5` peut maintenant être utilisée pour obtenir un produit modulo 5 particulier.

```
> m5[2,4] ;
```

$$3$$

Quand un élément de tableau n'est pas défini, le mécanisme d'évaluation triviale s'applique comme toujours. Supposons qu'on construise une table analogue à la précédente modulo 6, mais qu'on oublie de corriger les domaines de variation de `i` et `j` :

```
> m6 := array(0..5, 0..5) ;
```

$$m6 := \mathrm{array}(0..5, 0..5, [\,])$$

```
> for i from 0 to 4 do
    for j from 0 to 4 do
      m6[i, j] := (i * j) mod 6
      od
    od ;
```

L'utilisation de notre table pour «retrouver» le produit modulo 6 de 3 et 5 donne alors le résultat suivant :

```
> m6[3,4], m6[3,5] ;
```

$$0, m6_{3,5}$$

Mais toute tentative de citer un indice illégal sera sanctionnée par un message d'erreur assez clair.

[7] Ce point peut être modifié par l'utilisateur à l'aide de la variable `printlevel`, voir la page de documentation correspondante.

```
> m6[3, 6] ;
Error, 2nd index, 6, larger than upper array bound 5
```

La notion d'indice illégal peut d'ailleurs être assez subtile ; examinez ceci :

```
> expr := m6[3,t] ;
```

$$m6_{3,t}$$

```
> t := 4 : expr ;
```

$$0$$

```
> t := 6 : expr ;
Error, 2nd index, 6, larger than upper array bound 5
```

Le bon point de vue est le suivant : l'indice t peut être ou non «légal» selon sa valeur, et dans le doute, Maple accepte l'indice. Si par la suite, on affecte au symbole `t` la valeur 4 (légale) ou 6 (illégale), l'évaluation de `expr` est correcte ou non.

4.8.1 Création-Initialisation.

On peut initialiser directement tout ou partie des éléments d'un tableau à la création en prévoyant en plus une liste d'initialisation. Soit à créer un tableau des cubes des entiers compris entre 10 et 20 :

```
> cubes := array(10..20, [seq(i^3, i=10..20)]) :
> cubes[15] ;
```

$$3375$$

On voit ici que le mécanisme permettant de créer facilement, à l'aide de la procédure `seq`, une suite d'éléments *itérative* permet une initialisation confortable et lisible. On peut critiquer le choix du symbole pluriel `cubes`, à l'origine de l'appel `cubes[15]`, incohérent, puisqu'on ne demande qu'un seul cube ! mais si on choisit le singulier, c'est la désignation de la table elle-même avec un singulier qui est alors critiquable. Le perfectionniste en choix de symboles peut demander :

```
> cube := cubes ;
```

$$cube := cubes$$

```
> cube[16] ;
```

$$4096$$

Le profane peut penser ici que l'affectation du symbole `cubes` au symbole `cube` a transmis aussi de façon «souterraine» la table sous-jacente valeur de `cubes`, comme semble l'indiquer l'évaluation qui suit. Cette interprétation est incorrecte ; quand Maple lit l'expression à évaluer, il trouve une expression indexée, où, avant de faire travailler le mécanisme de lecture de table, il *évalue* l'expression racine, ici `cube` ; il trouve le symbole `cubes` et c'est donc à partir de ce symbole qu'il cherchera l'élément de table demandé. D'une certaine façon, l'expression `cube[16]` est une expression à *deux* arguments, le premier le symbole `cube`, le second l'entier `16` ;

l'évaluation *implicite* de `cube` produit `cubes` et puisqu'il s'agit d'une expression indexée, Maple va examiner si la valeur du symbole `cubes` est une table, d'où le résultat. Voir aussi :

```
> cubes := cubes ;
```

$$cubes := cubes$$

```
> cube[16] ;
```

$$cubes_{16}$$

C'est ici un cas très curieux qui semble contredire ce qui avait été expliqué page 53. Dans l'instruction d'affectation «`cubes := cubes`», l'évaluation du second membre est *implicite* et la valeur du symbole `cubes` étant une table, c'est seulement le symbole `cubes` qui est retourné, à cause de la règle Eval-16. Ensuite, l'affectation de ce symbole à lui-même est interprétée par Maple comme une demande de *libération* du symbole, voir page 53. Donc ici le symbole `cubes` est libéré. Comme `cube` chaîne toujours vers `cubes`, l'évaluation de `cube[16]` provoque l'évaluation de `cube` vers `cubes`, mais puisque cette fois le symbole est libre, le mécanisme d'évaluation triviale s'applique et l'expression `cubes[16]` est en définitive retournée.

On rediscutera ces questions quand on examinera les problèmes assez subtils liés aux échanges de valeurs entre procédures et monde «extérieur». Notez dès à présent que des bugs subtils et très coriaces à élucider en sont souvent la conséquence si on ne procède pas de façon précise en la matière.

Pour en terminer avec l'initialisation des tableaux, il faut aussi signaler que si un tableau a une dimension multiple, il faut l'initialiser avec une liste de listes de listes... avec la profondeur ad hoc.

```
> sample := array(0..1, 0..2, [[4, 5, 6], [7, 8, 9]]) :
> sample[0, 2], sample[1, 1] ;
```

$$6, 8$$

4.9 Vecteurs et matrices.

Les vecteurs et matrices sont des tableaux particuliers, et puisque les tableaux sont des tables particulières, les vecteurs et les matrices sont donc aussi des tables particulières. Les subtilités à base d'évaluations implicites et explicites vont dont être rencontrées ; elles demandent la plus grande précision, sous peine de bugs assez difficiles.

4.9.1 Vecteurs.

Un vecteur est un tableau de dimension 1 où l'ensemble des indices est un **range** `1..`n pour un entier positif n :

```
> A := array(0..2, [4,5,6]) ; V := array(1..3, [4,5,6]) ;
```

$A :=$ array(0 .. 2[
(0) = 4
(1) = 5
(2) = 6
])

$$V := [4, 5, 6]$$

Les formes *internes* des deux objets construits sont pratiquement les mêmes ; pour `A`, l'indice varie de 0 à 2, alors que pour `V` il varie de 1 à 3. Mais ceci suffit à faire de la valeur de `V` un vecteur, alors que la valeur de `A` n'est pas un vecteur :

```
> type(A, vector), type(V, vector) ;
```

$$false, true$$

Le lecteur (très-très) attentif doit ici être surpris qu'on n'ait pas préféré :

```
> type(eval(A), vector), type(eval(V), vector) ;
```

$$false, true$$

car en principe une évaluation *explicite* est nécessaire pour atteindre les valeurs tableaux ; oui, mais la procédure `type` applique elle-même l'évaluation explicite au premier argument si le type désigné au second demande une telle évaluation pour atteindre l'un de ses éléments, ce qui nous dispense dans ce cas, *et dans ce cas seulement*, de l'application de la procédure `eval`. Comparer avec :

```
> type(A, symbol), type(V, symbol) ;
```

$$true, true$$

où cette fois, puisque le type proposé est `symbol`, seule une évaluation *implicite* est effectuée !

Un vecteur est affiché *comme* une liste mais n'est en aucune façon une liste :

```
> L := [4,5,6] ;
```

$$[4, 5, 6]$$

```
> type(L, list), type(L, symbol), type(L, vector),
  type(V, list), type(eval(V), list) ;
```

$$true, false, false, false, false$$

Cette fois, pour une liste, même l'évaluation implicite du symbole `L` a retourné la liste, et c'est pourquoi `type(L, symbol)` répond négativement ; l'évaluation implicite ou explicite de `V` ne donne pas une liste, elle retourne respectivement un symbole ou un vecteur.

4.9.2 Matrices.

La situation est très analogue pour les matrices. Une matrice est un tableau de dimension 2 dont les domaines de variation des indices commencent par 1.

```
> T1 := array(0..4, 1..6) ;
  M := array(1..4, 1..6) ;
  T2 := array(1..4, 2..6) ;
```

$$T1 := \text{array}(0..4, 1..6, [])$$
$$M := \text{array}(1..4, 1..6, [])$$
$$T2 := \text{array}(1..4, 2..6, [])$$

De ces trois tableaux du même style, seul le deuxième est une matrice.

```
> type(T1, matrix), type(M, matrix), type(T2, matrix) ;
```

$$false, true, false$$

De nombreuses méthodes[8] adaptées aux nombreux cas de figure de définitions de matrices sont disponibles pour initialiser les matrices. On les examinera plus tard dans ce texte au gré des besoins pour nos nombreux exemples de calculs matriciels. Juste pour donner une petite idée de ce qui est possible, soit à construire la matrice 3×3 dont la première ligne est 1, 2, 3, la seconde 4, 5, 6, et la dernière 7, 8, 9. Le plus commode pour un tel cas consiste à initialiser la matrice à construire à l'aide d'une seule liste, constitué des éléments de matrice lus à l'occidentale, de gauche à droite puis de haut en bas.

```
> M := matrix(3,3, [seq(i, i=1..9)]) ;
```

$$M := \begin{bmatrix} 1 & 2 & 3 \\ 4 & 5 & 6 \\ 7 & 8 & 9 \end{bmatrix}$$

Là aussi, Maple vous facilite la vie en choisissant un affichage beaucoup plus clair, celui généralement utilisé par les mathématiciens pour les matrices. Comparer les deux tableaux :

```
> T1 := array(0..1, 1..2, [[1, 2], [3, 4]]) ;
  T2 := array(1..2, 1..2, [[1, 2], [3, 4]]) ;
```

$T1 :=$ table([
$(0,1) = 1$
$(0,2) = 2$
$(1,1) = 3$
$(1,2) = 4$
])

$$T2 := \begin{bmatrix} 1 & 2 \\ 3 & 4 \end{bmatrix}$$

`T2` est une matrice et est affichée comme telle, alors que le domaine de variation du premier indice de `T1` exclut que `T1` soit une matrice ; l'affichage de `T1` est donc

[8] Voir notamment `matrix`, `entermatrix` et `randmatrix`.

l'affichage très général, moins confortable, des tables.

Un package assez vaste appelé `linalg` (linear algebra) donne à l'utilisateur les moyens les plus évidents de calcul matriciel. Chargeons-le dans notre environnement.

```
> with(linalg)
Warning, new definition for norm
Warning, new definition for trace
[BlockDiagonal, GramSchmidt, ...
... ... ...
vecpotent, vectdim, vector, wronskian]
```

On n'a pas montré ici la liste complète des 114 procédures constituant ce package. Tous les grands sujets classiques du calcul matriciel y sont traités. Par exemple le déterminant de notre matrice M est nul :

```
> det(M) ;
```

$$0$$

Cette matrice est donc singulière, mais quel est son rang ?

```
> rank(M) ;
```

$$2$$

Le noyau de cette matrice est donc de dimension 1 ; comment en avoir un générateur ?

```
> K := kernel(M) ;
```

$$K := \{[1, -2, 1])\}$$

Sceptique ? Il faut alors évaluer le produit de la matrice `M` par le vecteur qui vient d'être montré ; pour extraire ce vecteur de l'*ensemble* de vecteurs constituant la *base* du noyau, ici réduite à un élément, on peut utiliser l'indexation `[1]` ; un calcul matriciel doit être effectué «sous» `evalm` et l'opérateur de multiplication matricielle est «`&*`».

```
> evalm(M &* K[1]) ;
```

$$[0, 0, 0]$$

Ce calcul très simple montre qu'on peut travailler *interactivement* avec des données matricielles quelconques. On en verra de nombreux exemples, voir notamment les chapitres ■■■. On avait déjà vu un exemple relativement sophistiqué page 99 et suivantes à propos de problèmes d'interpolation et de matrices de Vandermonde.

-o-o-o-o-o-o-o-

Chapitre 5

Programmation.

5.1 Introduction.

Maple n'est pas seulement un système de *calcul formel* ; c'est aussi un *langage de programmation*. Il est de ce point de vue *théoriquement* équivalent à tous les autres langages, bien que, comme chacun sait, selon le contexte, tel langage est commode ou au contraire délicat à utiliser.

Le point de vue évident à ce propos des concepteurs de Maple était, et reste, le suivant. Le contexte *calcul symbolique* est relativement délicat, et pour éviter toute surcharge de complexité, un langage de programmation qu'on peut qualifier de *rustique* a été adopté.

Il a en particulier l'avantage d'être facile à apprendre, car conceptuellement simple tout en étant logiquement bien équilibré. Sa faiblesse rapidement perçue comme la plus évidente est dans le *typage* des objets en général, des variables en particulier. Avec une situation manifestement un peu déséquilibrée de ce point de vue : d'une part, d'innombrables types prédéfinis sont disponibles, et l'utilisateur a de grandes facilités pour en créer de nouveaux à sa guise. Mais il n'est pas possible de déclarer les variables locales[1] de procédures, ou même les variables globales. Ceci donne un style de programmation généralement assez «sauvage», proche de la programmation en Lisp des premiers temps, où il arrive souvent de voir par erreur une variable prévue pour localiser une matrice recevoir une... intégrale comme valeur sans que Maple signale quoi que ce soit ! La procédure `ASSERT` permet de mettre en œuvre quelques précautions dans ce registre, mais le plus souvent assez rudimentaires.

5.2 Instructions conditionnelles.

Les canons de la programmation dite *structurée* réduisent tout algorithme à un assemblage plus ou moins complexe d'instructions *conditionnelles* et d'instructions

[1] Ceci est maintenant possible dans Maple 6.

itératives. La forme la plus générale d'une instruction Maple conditionnelle est :

if
  $condition_1$
  then
    $instruction_{1,1}$;
    . . . . . ;
    $instruction_{1,n_1}$
elif
  $condition_2$
  then
    $instruction_{2,1}$;
    . . . . . ;
    $instruction_{2,n_2}$
elif
    . . . . .
    . . . . .
elif
  $condition_k$
  then
    $instruction_{k,1}$;
    . . . . . ;
    $instruction_{k,n_k}$
else
    $instruction_{k+1,1}$;
    . . . . . ;
    $instruction_{k+1,n_{k+1}}$
fi ;

Il est rare et surtout franchement déconseillé d'utiliser le maximum de généralité affiché ici ! Presque toujours, un minimum de réflexion sur le sujet traité permet de se dispenser des instructions conditionnelles à multiples branches, de plus en général assez peu lisibles. Ceci dit, qui peut le plus peut le moins et inversement toute limitation sur le degré de généralité montré ici, commun à presque tous les langages de programmation, serait malcommode dans tel ou tel cas particulier.

Seule relative originalité de Maple : la possibilité «automatique» d'insérer dans chaque branche un nombre arbitraire d'instructions. Avec un inconvénient : cette possibilité nécessite un indicateur de fermeture, `fi` en fin d'instruction, ou bien un séparateur, `elif` ou `else` indiquant le passage à la branche suivante. Mais ceci a dispensé les concepteurs de Maple de prévoir un format d'*instruction composée*, comparable au {. . . . . .} de C. Il résulte de ceci que la faute d'écriture consistant à oublier le `fi` terminal est très fréquente, et surtout n'est pas détectable *en place*, puisque le point-virgule suivant indique que la branche de `then` ou `else` continue. Si ceci se produit dans le texte d'une procédure, la faute n'est le plus souvent détectable qu'au `end` terminal, situation qui laisse l'utilisateur dans le vide à chercher

si c'est une parenthèse, ou un crochet, ou une accolade, ou un `if`, ou encore un `do` qui n'est pas refermé. Seule solution : soigneusement indenter son texte pour en vérifier facilement la structure, comme il l'est toujours montré dans nos exemples.

On ne peut que regretter amèrement le caractère franchement primitif de l'éditeur Maple de ce point de cue ; il devrait, par un jeu quelconque d'édition, indiquer dans le texte du groupe d'exécution en cours quelle est l'entité la plus profonde qui est la première à refermer. Étant donné la sophistication d'autres éléments de cet éditeur, on reste un peu surpris que ceci ne soit pas encore proposé.

Passons en revue les cas d'utilisation les plus courants d'instructions conditionnelles.

5.2.1 if – then – fi.

C'est le cas le plus simple. Il intervient très fréquemment en début de procédure pour traiter un ou plusieurs cas particuliers, permettant de dévier la procédure sur un autre cas. Considérons par exemple l'exercice d'étudiant consistant à mettre en place une procédure de calcul numérique du logarithme *de base 10*, celle fournie par le «système» étant supposée en panne.

Un algorithme très commode et de plus assez efficace pour ce faire consiste à n'utiliser que les «quatre opérations» classiques. Comme c'est souvent le cas, on a intérêt à être un peu plus général et à installer un algorithme capable de calculer $\phi(\alpha, \beta, \gamma) = \alpha + \beta \log(\gamma)$, pour $\gamma > 0$, le cas du logarithme seul étant le cas particulier $\alpha = 0$ et $\beta = 1$. Cette organisation permet en effet de «reporter» par opérations élémentaires le logarithme vers α jusqu'à ce que la précision souhaitée soit atteinte ; cette précision est définie par un paramètre ε, par exemple donné comme $\varepsilon = 10^{-10}$. On procède de façon itérative alors comme suit :

1. Si $\gamma = 1$, alors $\alpha + \beta \log(\gamma) = \alpha$;
2. Si $\gamma < 1$, alors $\alpha + \beta \log(\gamma) = \alpha - \beta \log(1/\gamma)$;
3. Si $\gamma \geq 10$, alors $\alpha + \beta \log(\gamma) = \alpha + \beta + \beta \log(\gamma/10)$;
4. Si les conditions précédentes ne sont pas satisfaites, c'est que $1 < \gamma < 10$ et donc $0 < \log(\gamma) < 1$; dans ces conditions, si α est non nul et si $|\beta/\alpha| < \varepsilon$, l'exigence sur la précision relative est satisfaite en se contentant de α ;
5. Sinon on utilise $\alpha + \beta \log(\gamma) = \alpha + (\beta/2) \log(\gamma^2)$.

Il est facile de voir que l'évolution de cet algorithme fait tendre β/α vers 0, et la quatrième condition termine donc l'algorithme. La traduction en Maple, uniquement à coups de «**if** – **then** – **fi**» s'écrit alors :

```
> phi := proc(alpha::realcons, beta::realcons, gamma::positive)
   global epsilon ;
   if gamma=1. then RETURN(alpha) fi ;
   if gamma<1. then RETURN(phi(alpha, -beta, 1/gamma)) fi ;
   if gamma>=10. then RETURN(phi(alpha+beta, beta, gamma/10.)) fi ;
   if alpha<>0. and abs(beta/alpha)<epsilon then RETURN(alpha) fi ;
   RETURN(phi(alpha, beta/2., gamma^2))
 end :
```

La procédure `phi` est le moteur de la procédure cherchée :

```
> mylog10 := proc(x::positive)
   global epsilon ;
   epsilon := 1e-10 ;
   RETURN(phi(0., 1., convert(x,float)))
 end :
```

Noter la déclaration globale de `epsilon` dans les deux procédures, indispensable pour la communication à ce propos, sinon il faudrait précoir un quatrième argument pour la procédure `phi`, ce qui alourdirait son fonctionnement récursif. Ceci dit cette méthode est *dangereuse*, à cause des risques d'interférence avec un autre usage global d'`epsilon` ailleurs. On verra dans le prochain chapitre comment mieux solutionner ce problème, voir page ■■■.

La conversion en flottant dans `mylog10` est nécessaire pour éviter le clash surprenant :

```
> if 1 = 1. then equal else ‘not equal‘ fi ;
                                        not equal
```

car contrairement à ce que croient les mathématiciens, en informatique, l'entier 1 et le flottant 1. ne sont pas égaux ! Ce qui oblige à choisir entre 1 et 1. pour le cas exceptionnel du logarithme trivial. Ces préparatifs étant faits, notre logarithme est maintenant opérationnel :

```
> mylog10(2) ;
                                        .3010299956
```

qu'on peut comparer au logarithme «officiel» soudainement redevenu disponible :

```
> log[10](2.);
                                        .3010299957
```

Des tests faciles à mener — utiliser la procédure `time` — permettent de constater que notre procédure est aussi efficace que la procédure Maple d'origine. Mais la procédure Maple traite aussi le cas du logarithme complexe, non couvert par la nôtre. Cette dernière peut facilement être adaptée pour traiter d'autres bases de logarithmes, et pour atteindre des précisions supérieures, en jouant sur les variables `epsilon` et `Digits`.

5.2.2 if – then – else – fi.

Un exemple était montré ci-dessus, pour tester l'égalité potentielle de l'entier «1» et du flottant «1.». La procédure `evalb` est appelée implicitement pour la comparaison :

```
> evalb(1 = 1.) ;
```

false

Comparer avec la procédure `is` :

```
> is(1 = 1.) ;
```

true

On avait signalé la difficulté provoquée par la comparaison du paramètre γ avec 1 (ou 1.) pour traiter le cas, indispensable sous peine d'erreur occasionnelle, du logarithme trivial dans la procédure `phi` ; cette difficulté avait été surmontée par la conversion en flottant de l'argument `x` à l'appel de `phi` dans la procédure `mylog10`. Ce qui est observé ici donne une autre solution : remplacer le début d'instruction conditionnelle «`if gamma=1. ...`» par «`if is(gamma = 1.) ...`». On observe que ces questions mathématiquement élémentaires demandent une grande lucidité pour être traitées ici correctement. Ainsi :

```
> if is(1 = 1.) then equal else ‘not equal‘ fi ;
```

equal

Comme usage un tout petit peu moins trivial de «**if – then – else – fi**», observons que la procédure `mylog10` aurait pu être terminée d'une façon que beaucoup préfèrent :

```
    if alpha<>0. and abs(beta/alpha)<epsilon
      then RETURN(alpha)
      else(RETURN(phi(alpha, beta/2., gamma^2)))
  end :
```

bien que les deux méthodes soient ici parfaitement équivalentes. Cette équivalence est due à un phénomène particulier qu'il faut bien avoir identifié : ici la présence d'une instruction `RETURN` dans une branche implique que de toute façon la procédure est interrompue, que la condition testée — comparaison avec ε — soit satisfaite ou non.

Pour montrer une situation différente considérons le problème «$n/2$ ou $3n+1$». Soit n un entier positif quelconque. On applique de façon itérative le processus suivant : si n est pair, on le divise par 2, sinon on le remplace par $3n+1$. On veut une procédure qui retourne le nombre d'étapes nécessaires pour atteindre 1. Il est conjecturé que ce nombre est toujours défini, autrement dit que le processus décrit finit par atteindre 1, mais cette conjecture, très ancienne, reste non démontrée quand ces lignes sont écrites. Version utilisant «**if – then – else – fi**» :

```
 > nsteps := proc(n::posint)
    local nsteps, nn ;
    nsteps := 0 ; nn := n ;
    while nn <> 1 do
      nsteps := nsteps+1 ;
      if nn mod 2 = 0
        then nn := nn/2
        else nn := 3*nn+1
      fi ;
    od ;
    RETURN(nsteps)
  end :
> nsteps(10^67-1);
                                2076
```

La procédure `nsteps` est vraiment programmée «comme on pense». Seule coquetterie, l'usage du même nom pour la procédure et pour la variable locale «compteur» ; certains n'aiment pas ce style, d'autres trouvent au contraire commode de désigner ainsi implicitement quelle est la variable «clé» de la procédure. Noter aussi la nécessité d'une variable locale `nn` initialisée à l'argument `n`, car il n'est pas possible sous Maple de modifier cet argument, à moins d'utiliser une technique qu'on examinera en temps utile, page ■■■, mais en fait ici inappropriée.

Comparer à la version suivante où on a «joué» à utiliser seulement la version «**if** – **then** – **fi**» de l'instruction conditionnelle :

```
> nsteps := proc(n::posint)
    local nsteps, nn ;
    nsteps := 0 ;
    nn := n ;
    while nn <> 1 do
      nsteps := nsteps+1 ;
      if nn mod 2 = 0 then nn := nn/2 fi ;
      if nn mod 2 = 1 then nn := 3*nn+1 fi
    od ;
    RETURN(nsteps)
  end :
```

La traduction dans ce nouveau style semble évidente. Vraiment ? Essayez le résultat et vous aurez une surprise ! C'est un excellent exercice de debugging d'identifier l'origine de cette surprise ; plusieurs solutions sont possibles pour la surmonter, bien sûr en respectant notre règle : la clause **else** reste interdite ! La solution ci-dessous n'est pas sans intérêt, car elle montre une solution d'un style assez différent, où cette fois chaque clause **then** comporte *deux* instructions séparées par *un* point-virgule ; noter dans ce registre que les mots-clés **elif**, **then** et **else** sont du point de vue syntaxique des *séparateurs* et qu'il est donc inutile, bien que toléré, de faire précéder **elif**, **else** et **fi** d'un point-virgule, autre séparateur plus profond dans la hiérarchie.

```
> nsteps := proc(n::posint)
    local nsteps, nn ;
    nsteps := 0 ; nn := n ;
    while nn <> 1 do
      if nn mod 2 = 1 then nn := 3*nn+1 ; nsteps := nsteps+1 fi ;
      if nn mod 2 = 0 then nn := nn/2 ; nsteps := nsteps+1 fi
    od ;
    RETURN(nsteps)
  end :
```

5.2.3 if – elif ...

La procédure `phi` qui nous a servi pour le calcul du logarithme de base 10 est le plus souvent écrite par les programmeurs :

```
 > phi := proc(alpha::realcons, beta::realcons, gamma::positive)
    global epsilon ;
    if gamma = 1. then RETURN(alpha)
    elif gamma<1. then RETURN(phi(alpha, -beta, 1/gamma))
    elif gamma>=10. then RETURN(phi(alpha+beta, beta, gamma/10.))
    elif alpha<>0. and abs(beta/alpha)<epsilon
      then RETURN(alpha)
    else   RETURN(phi(alpha, beta/2., gamma^2))
    fi
  end :
```

On voit en particulier que la lisiblité est excellente.

5.3 Instructions itératives.

De nombreux exemples ont été déjà montrés, qui nous permettront de consacrer ce paragraphe plutôt à des compléments techniques par rapport aux usages les plus simples déjà vus, et aussi à des considérations de style.

5.3.1 do.

Encore appelé `do` «forever», car utilisée directement sans préambule, cette instruction est indéfiniment répétée ; elle ne peut alors être interrompue que par une instruction `break` ou `RETURN`, ou encore, plus piteusement, par un click sur le bouton stop :  . Pour montrer un usage plausible de ce `do`, transformons nos procédures `mylog10` et `phi` (cf. pages 149 et 152) de la forme récursive précédemment utilisée vers une forme *itérative*, un peu plus technique, mais indispensable en haute précision. Une seule procédure va maintenant suffire.

```
> mylog10 := proc(x::positive)
    local alpha, beta, gamma, epsilon ;
    epsilon := 1e-10 ;
    alpha, beta, gamma := 0., 1., convert(x, float) ;
    do
      if gamma = 1. then RETURN(alpha)
      elif gamma<1. then beta, gamma := -beta, 1/gamma
      elif gamma>=10. then alpha, gamma := alpha+beta, gamma/10.
      elif alpha<>0. and abs(beta/alpha)<epsilon
        then RETURN(alpha)
        else beta, gamma := beta/2., gamma^2
      fi
    od
  end :
> mylog10(2) ;
```

$$.3010299956$$

La nouvelle procédure est véritablement une *traduction* en style itératif des deux précédentes. Noter l'usage intensif des *affectations simultanées*, très commodes et très lisibles. L'affectation simultanée n'est pas seulement un raccourci d'écriture ; le fait que les affectations demandées soient effectuées «en parallèle» permet des raccourcis efficaces et bien lisibles. L'exemple prototype est l'échange entre deux valeurs de variables. Sans affectation simultanée, une variable «auxiliaire» est nécessaire :

```
> a := 2 ; b := 3 ;
```

$$a := 2$$
$$b := 3$$

```
> tmp := a ; a := b ; b := tmp ;
```

$$tmp := 2$$
$$a := 3$$
$$b := 2$$

alors que par affectation simultanée :

```
> a, b := b, a ; a, b ;
```

$$a, b := 2, 3$$
$$2, 3$$

Dans le cours Maple de niveau *perfectionnement*, on cherche à démontrer que cette affectation prétendue parallèle ne l'est pas tant que ça.

```
> a, b := 2, 3 ;
```

$$a, b := 2, 3$$

```
> a, b := [b, assign('a' = b)][1], a ;
```

$$a, b := 3, 3$$

Cette expérience assez ésotérique *démontre* que la première composante a en fait bien été évaluée *avant* la seconde ; le parallélisme classiquement prétendu est donc

tout relatif ! En fait les expressions du second membre de l'affectation simultanée sont calculées *séquentiellement*, mais chaque affectation demandée n'est effectuée qu'*après* évaluation complète de toutes ces expressions.

5.3.2 for et while.

Il est souvent commode de gérer l'initialisation et la terminaison d'une itération à l'aide de clauses `for` et/ou `while`. L'usage en est si classique qu'il y a peu à dire. Considérons l'exercice consistant à déterminer la première puissance d'un réel `a` vérifiant $a > 1$ qui est plus grande qu'un autre réel `b`. Le logarithme permet une réponse simple, mais il est à nouveau en panne... Avec `do` seul, on peut programmer la procédure :

```
> FirstPower := proc(a::Range(1,infinity), b::realcons)
    local FP, power ;
    FP := 0 ; power := 1. ;
    do
      if power >= b
        then RETURN(FP)
        else FP := FP+1 ; power := power*a
      fi
    od
  end :
> FirstPower(1.1, 10) ;
                                    25
```

C'est en fait très lisible, mais la progression régulière du compteur suggère plutôt d'utiliser une clause `for` :

```
> FirstPower := proc(a::Range(1,infinity), b::realcons)
    local FP, power ;
    power := 1. ;
    for FP from 0 do
      if power >= b
        then RETURN(FP)
        else power := power*a
      fi
    od
  end :
> FirstPower(1.1, 10) ;
                                    25
```

En effet le compteur est incrémenté par défaut d'une unité à chaque tour — il faut dire à chaque *itération* — de sorte que la clause `for` gère automatiquement la progression de la variable `FP` que la plupart des programmeurs appelleraient simplement `i`. Si pour une raison qui nous échappe, le cahier des charges demande la première puissance *paire* satisfaisant la même condition, il suffit d'utiliser la sous-clause `by` :

```
> FirstEvenPower := proc(a::Range(1,infinity), b::realcons)
    local FP, power ;
    power := 1. ;
    for FP from 0 by 2 do
      if power >= b
        then RETURN(FP)
        else power := power*a^2
      fi
    od
  end :
> FirstEvenPower(1.1, 10) ;
                                    26
```

Il serait bien sûr plus rentable, dans le cas où chaque micro-seconde serait précieuse, de calculer une fois pour toutes le carré de a. Mais la terminaison elle-même peut être *simultanément* gérée par une clause `while`, ce qui donne un style beaucoup plus compact.

```
> FirstEvenPower := proc(a::Range(1,infinity), b::realcons)
    local sqa, FP, power ;
    sqa := a^2 ; power := 1. ;
    for FP from 0 by 2 while power < b do
      power := power*sqa
    od ;
    RETURN(FP)
  end :
> FirstEvenPower(1.1, 10) ;
                                    26
```

Il se pourrait aussi qu'on craigne un temps de calcul trop long si a était trop proche de 1 et b trop grand ; on voudrait donc limiter à tout prix à 1000 le nombre d'itérations.

```
> FirstEvenPower := proc(a::Range(1,infinity), b::realcons)
    local sqa, FP, power ;
    sqa := a^2 ; power := 1. ;
    for FP from 0 by 2 to 1998 while power < b do
      power := power*sqa
    od ;
    if FP = 2000
      then ERROR(‘Temps calcul trop long, exécution arrêtée.‘)
      else RETURN(FP)
    fi
  end :
> FirstEvenPower(1.1, 10) ;
                                    26
> FirstEvenPower(1.001, 1e20) ;
Error, (in FirstEvenPower) Temps calcul trop long, exécution arrêtée.
```

Reprenons la procédure `nsteps` de la page 151 ; on peut ainsi la libérer de l'initialisation et de l'entretien de la variable `nsteps` à l'aide de la clause `for` :

```
> nsteps := proc(n::posint)
    local nsteps, nn ;
    nn := n ;
    for nsteps from 0 while nn <> 1 do
      if nn mod 2 = 0 then nn := nn/2
                      else nn := 3*nn+1
      fi ;
    od ;
    RETURN(nsteps)
  end :
```

Il arrive aussi qu'une clause `while` suffise à elle seule. Par exemple si, dans notre procédure `FirstPower`, on est intéressé seulement par la *valeur* de la puissance et non par l'exposant, on peut l'écrire :

```
> FirstPower := proc(a::Range(1,infinity), b::realcons)
    local power ; power := 1. ;
    while power < b do power := power * a od ;
    RETURN(power)
  end :
> FirstPower(1.1, 10) ;
                                  10.83470595
```

5.3.3 next et break.

Deux mots-clés sont quelquefois utiles dans un contexte itératif. L'instruction `next` demande l'interruption de l'itération en cours *et* le passage immédiat à l'itération suivante, alors que l'instruction `break` demande l'interruption *complète* de l'itération. Exemple artificiel :

```
> for i from 0 do
    if i mod 3 = 0 then next fi ;
    if i = 5 then break fi ;
    print(i)
  od ;
                                  1
                                  2
                                  4
```

Il n'est pas difficile de démontrer qu'on peut toujours se passer de ces instructions, ceci dit il est fréquent que leur usage facilite l'écriture et améliore la lisibilité des programmes itératifs un peu compliqués.

5.4 Tris.

La méthode standard pour vérifier que l'essentiel est compris en programmation conditionnelle et itérative consiste à programmer les algorithmes élémentaires de

tri ; ils nécessitent en effet du programmeur une gymnastique instructive. Considérons par exemple l'un des tris les plus simples, le tri par *insertion*. On «range» les objets à trier les uns après les autres, *insérant* chaque objet à la bonne place, qu'il faut donc déterminer par rapport aux objets déjà rangés. La liste des objets triés augmente donc régulièrement, et les objets au-delà de l'objet à insérer doivent être déplacés pour laisser une place au nouveau venu. Plusieurs solutions techniques sont possibles qu'il est intéressant de comparer.

Mais avant de trier, il faut un objet raisonnable à trier. Dans les exercices de tri, ce sont classiquement des listes d'entiers. Soit donc à construire une liste de 20 entiers «aléatoires» compris entre 1 et 1000. On construit d'abord une procédure de génération, qu'on essaie par exemple cinq fois.

```
> rnd1000 := rand(1..1000) :
> seq(rnd1000(), i=1..5) ;
```

$$82, 271, 698, 564, 977$$

Moyennant quoi notre liste exercice peut être construite :

```
> list1 := [seq(rnd1000(), i=1..20)] ;
```

$$\mathit{list1} := [539, 86, 769, 922, 410, 656, 164, 458, 161, 675, 86, 917, 462, 308, 350, 87, 199, 467, 710, 974]$$

Pour avoir les idées plus claires, on va commencer par trier en créant un nouvel objet, une liste qu'il faut initialiser à la bonne longueur, en la remplissant par exemple par des zéros, comme suit.

```
> list2 := [0 $ 20] ;
```

$$\mathit{list2} := [0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0]$$

On va prendre ensuite le premier élément, 539, le ranger dans la première case, puis le second, 86 ; son insertion demande de pousser 539 à la deuxième case, etc. On voit assez rapidement qu'on a intérêt à «descendre» dans la liste résultat pour chercher la bonne place du nouvel élément, tout en décalant les éléments lus pour préparer la place du nouveau venu. D'où cette programmation :

```
> insertion1 := proc(list1::list(integer))
    local list2, i, j ;
    list2 := [0 $ nops(list1)] ;      # initialisation liste résultat
    for i from 1 to nops(list1) do    # pour chaque élément liste entrée
      for j from i-1 to 1 by -1 do    # pour chaque élément déjà trié
        if list1[i] < list2[j]        # comparaison
          then list2[j+1] := list2[j] # si < décaler
          else break                  # si >= stopper l'itération intérieure
        fi
      od ;
      list2[j+1] := list1[i]          # insérer
    od ;
    RETURN(list2)
  end :
```

```
> insertion1(list1) ;
```
$$[86, 86, 87, 161, 164, 199, 308, 350, 410, 458, 462, 467, 539, 656, 675, 710, 769, 917, 922, 974]$$

Les textes après `#` sur chaque ligne sont des commentaires.

Cette solution n'est déjà pas si facile à obtenir. On peut par exemple oublier la sous-clause «`by -1`» dans l'itération interne, car le fait que la valeur initiale de `j` soit plus grande que la valeur finale peut faire croire que Maple «devinera» qu'il faut faire décroître `j` ; il n'en est rien ; comparer :

```
> for j from 2 to 1 do print(coucou) od ;
> for j from 2 to 1 by -1 do print(coucou) od ;
```
coucou

coucou

Si on oublie «`by -1`» dans la programmation de `insertion1`, le résultat est cocasse, essayez et cherchez à comprendre la nature du message d'erreur. Autre subtilité, il est plus instinctif dans le `if` de faire l'insertion avant le `break` quand la réponse est négative ; mais ceci interdit toute insertion si la réponse est *toujours* positive, ce qui arrive quand l'élément à insérer est plus petit que tous ceux déjà triés. On voit que si l'insertion est réalisée à l'extérieur après terminaison de l'itération la plus intérieure, alors elle fonctionne toujours correctement. C'est d'ailleurs un cas, peu fréquent, où la persistance de la valeur de `j` après l'itération est bigrement utile : la coïncidence de correction de l'insertion, qu'il y ait eu `break` ou non, est assez frappante, indice de bonne programmation.

Ceci dit ce programme a encore beaucoup de défauts. On voit d'abord qu'une clause `while` aurait permis un code plus compact, sensiblement plus élégant. Pourtant la correction de clauses simultanées `for` et `while` n'est jamais facile à tester, et il est souvent plus sûr de commencer sagement par un code plus «rustre». Certains s'imposent dans ce registre de commencer la programmation d'une itération `do ... od` sans clause `for` ou `while`, en prévoyant banalement après le `do` initial la progression éventuelle d'une variable pilote et/ou le test d'arrêt. Une fois que le programme dans ce style est au point, ils réfléchissent alors *seulement à ce moment* s'ils pourraient astucieusement utiliser une clause `for` ou `while`, ou même les deux, pour préfixer le `do`.

```
> insertion2 := proc(list1::list(integer))
    local list2, i, j ;
    list2 := [0 $ nops(list1)] ;     # initialisation liste résultat
    for i from 1 to nops(list1) do   # pour chaque élément liste entrée
      for j from i-1 to 1 by -1      # pour chaque élément déjà trié
        while list1[i] < list2[j] do # tant que trop petit
          list2[j+1] := list2[j]     # décaler
      od ;
      list2[j+1] := list1[i]         # insérer
    od ;
    RETURN(list2)
  end :
```

Un défaut d'une autre nature est assez important. Ici, pour un exercice d'étudiant, il est secondaire, mais pour un vrai tri sur de grands ensembles, il aurait de sérieux inconvénients. On observe que `list1[i]` est utilisé pour chaque comparaison, lecture dans une liste bien plus coûteuse que le laisse penser le texte `list1[i]`, si court. Alors que cet objet aurait pu être *lu* une seule fois et réutilisé ensuite ; il faut pour ce faire une nouvelle variable qu'on appelle `fllw` (following).

```
> insertion3 := proc(list1::list(integer))
    local list2, i, j, fllw ;
    list2 := [0 $ nops(list1)] ;     # initialisation liste résultat
    for i from 1 to nops(list1) do   # pour chaque élément liste entrée
      fllw := list1[i] ;             # élément suivant à insérer
      for j from i-1 to 1 by -1      # pour chaque élément déjà trié
        while fllw < list2[j] do     # tant que fllw trop petit
          list2[j+1] := list2[j]     # décaler
      od ;
      list2[j+1] := fllw             # insérer
    od ;
    RETURN(list2)
  end :
```

Comparons `insertion2` et `insertion3` pour une liste de 100 éléments. La procédure `time`, utilisée sans argument, retourne le temps «machine» en secondes depuis le début de la session ; elle permet donc de chronométrer une exécution. On répète chaque tri 100 fois pour une comparaison significative.

```
> list11 := [seq(rnd1000(), i=1..100)] :
> st := time() :
  for i from 1 to 100 do insertion2(list11) od :
  time() - st ;
                                    8.940
> st := time() :
  for i from 1 to 100 do insertion3(list11) od :
  time() - st ;
                                    8.680
```

Le gain est réel mais peu important pour des listes de si petite taille, sans aucune comparaison avec ce qui est fait usuellement sur machine. On ne peut faire plus ici pour une autre raison, qu'il est encore bien plus important de comprendre. Essayons d'utiliser notre tri pour une liste de 200 éléments.

```
> list11 := [seq(rnd1000(), i=1..200)] :
> insertion3(list11) ;
Error, (in insertion3) assigning to a long list, please use arrays
```

Ici, on avait oublié ce qui était expliqué page 122 : on ne peut pas modifier une liste de plus de 100 éléments ! Pour les tris conséquents, les seuls dignes d'intérêt, il faut donc *obligatoirement* utiliser une technique tableau (array), autrement dit «vecteur» (vector) dans le cas d'un seul indice.

```
> insertion4 := proc(list1::list(integer))
    local vct2, i, j, fllw ;
    vct2 := vector(nops(list1)) ;    # initialisation vecteur résultat
    for i from 1 to nops(list1) do   # pour chaque élément liste entrée
      fllw := list1[i] ;             # élément suivant à insérer
      for j from i-1 to 1 by -1      # pour chaque élément déjà trié
        while fllw < vct2[j] do      # tant que fllw trop petit
          vct2[j+1] := vct2[j]       # décaler
      od ;
      vct2[j+1] := fllw              # insérer
    od ;
    RETURN(convert(vct2, list))
  end :
```

On voit que le vecteur sert d'outil annexe, et que, en supposant que le cahier des charges demande le résultat sous forme de liste, on convertit le vecteur en liste en fin d'exécution. Refaisons le même test temps :

```
 > st := time() :
  for i from 1 to 100 do insertion4(list11) od :
  time() - st ;
                                    3.490
```

Le gain est très conséquent et on comprend bien maintenant pourquoi Maple vous interdit de modifier des listes longues : ces modifications sont véritablement *calamiteuses* en optimisation temps et espace mémoire, et pour éviter des défauts de rendement aux programmeurs inexpérimentés, Maple interdit purement et simplement de telles modifications sauf pour de petites listes. Même pour celles-ci l'exemple vu ici montre que ce peut être très coûteux si le même phénomène est répété intensivement.

Rien n'empêche maintenant de trier de longues listes. Une méthode naïve comme la nôtre a un coût temps de tri grossièrement proportionnel au carré du nombre d'léments à trier.

```
> list11 := [seq(rnd1000(), i=1..1000)] :
> time(insertion4(list11)) ;
                                    4.754
> list11 := [seq(rnd1000(), i=1..10000)] :
> time(insertion4(list11)) ;
                                   520.791
```

de sorte que les temps de tri deviennent vite importants. Les méthodes plus rusées à base d'arborescence sont beaucoup plus efficaces. Voir par exemple, pour un beau traitement didactique du sujet, le chapitre 2 du livre de Wirth [7]. Comparer aussi ce qui vient d'être fait avec la procédure de tri proposée par Maple lui-même :

```
> time(sort(list11)) ;
                                     0.45
```

Le temps de calcul est donc divisé par 10000 ! On ne saurait mieux montrer que Maple n'est pas vraiment fait pour la programmation de base. La programmation interne de `sort` est en effet écrite en C, avec une efficacité sans commune mesure, d'une part parce que la technique de tri est autrement sophistiquée, d'autre part parce que C est beaucoup plus proche du langage machine : notre programmation est en effet *interprétée* et l'efficacité par rapport à ce qui est en fait disponible sur machine est dérisoire.

-o-o-o-o-o-o-

Chapitre 6

Procédures.

6.1 Introduction.

On a vu dans le chapitre précédent un type de programmation itérative consistant à répéter un certain nombre de fois une séquence de programmation dans un environnement légèrement variable d'une fois à l'autre. La séquence en question doit alors être encapsulée dans une instruction «`do ... od`». Le présent chapitre est consacré à une autre forme de programmation «itérative», en un sens très différent, consistant à construire un objet **procedure**, appelé plus simplement *procédure*, puis à l'utiliser de temps à autre, quelquefois un grand nombre de fois, avec là aussi un environnement un peu différent d'une fois à l'autre.

La principale différence entre les deux cas est la suivante. Une suite d'instructions encapsulée dans un «`do ... od`» est exécutée de façon répétitive un certain nombre de fois puis «définitivement» abandonnée ; les guillemets de la qualification définitive sont nécessaires, parce que, pour d'autres raisons, cette itération peut en fait être à nouveau exécutée ; cette instruction `do` peut être par exemple à son tour encapsulée dans un autre «`do ... od`» comme on l'a vu dans le chapitre précédent pour notre programmation du tri-insertion, voir Section 5.4. Une instruction `do` peut aussi figurer dans une procédure, on a vu de nombreux exemples, de sorte que la réexécution de la procédure impliquera en particulier une nouvelle exécution de l'instruction `do`.

Le caractère itératif d'une procédure est différent. Il s'agit d'un ensemble d'instructions qu'on veut pouvoir exécuter «au besoin», à n'importe quel moment. Une terminologie bien définie doit être utilisée en la matière, terminologie qu'on présente dans cette introduction, dont la signification sera détaillée dans ce chapitre. Quand une procédure est *construite*, elle ne sera utilisable que si elle est installée dans l'*environnement*, accessible le plus souvent par l'intermédiare d'un symbole appelé alors, mais c'est un abus de langage, le *nom* de la procédure. Le cas des procédures *anonymes* est aussi important : une telle procédure est alors une composante d'un objet plus ou moins complexe, de nature quelconque, suite, liste, ensemble, matrice, etc. Le cas d'une procédure figurant comme élément d'une suite d'arguments d'un appel d'une autre procédure est fréquent.

Quand une procédure est disponible, elle peut être *appelée* de diverses façons. On dit alors qu'une *invocation* de la procédure est *activée*. Cette invocation peut être active un temps très court, moins d'une micro-seconde, ou un temps très long, plusieurs jours, avec bien sûr tous les intermédiaires possibles. Très important : *plusieurs invocations* de la même procédure peuvent être actives au même moment, mais au plus une est *en cours d'exécution* ; les autres, bien qu'actives, sont provisoirement *au repos*, en attente de la suite de leur exécution ; c'est le point clé de la *programmation récursive* dont un exemple typique est la programmation de la procédure `phi` page 149. Quand une invocation de procédure *termine*, cette invocation *retourne* toujours une valeur, quel que soit le cas de figure. Les appels de procédures qui «semblent» ne rien retourner retournent en fait la suite vide, celle qui est repérée par le symbole `NULL`, et ceci doit alors presque toujours être explicitement prévu dans la définition de la procédure. Une procédure peut aussi retourner plusieurs objets, en fait *une* suite d'objets de longueur arbitraire. Le mécanisme d'affectation multiple est alors souvent utilisé pour dispatcher les composantes du résultat vers divers symboles.

Des symboles dits *locaux* sont souvent *ajoutés* à l'environnement lors de l'activation d'une procédure. Les autres symboles utilisés dans la procédure sont *globaux*. Par ailleurs des symboles locaux très particuliers appelés *paramètres* servent à particulariser le travail d'une invocation particulière de la procédure. Quand la procédure est appelée, des *arguments* sont *passés* à la procédure, et les paramètres de la procédure sont *initialisés* en fonction de ces arguments ; on verra que cette description simplifiée est en fait assez incorrecte, mais chaque chose en son temps. On a déjà discuté pour une autre raison la différence entre ce que nous avons appelé *argument provisoire* et *argument définitif*, voir la règle Eval-14.

La partie *locale* de l'environnement peut à son tour contenir des procédures dont les règles de visibilité sont alors particulières au contexte. Un tel objet, en première approximation local à une procédure, peut néanmoins être *retourné* à l'extérieur de la procédure, tout en gardant ses règles de visibilité. Ces objets très étranges mènent à la *programmation fonctionnelle*, inaccessible aux utilisateurs des langages ordinaires, Pascal, C, C++ ou Java. C'est un outil très puissant, ouvrant un monde de programmation complètement nouveau, auquel le chapitre  sera entièrement consacré.

6.2 Exemple élémentaire.

Un utilisateur doit utiliser de façon fréquente une expression $\sqrt{x^2+y^2}$ pour des objets variés. Plutôt que de taper à chaque fois `sqrt(???^2 +???^2)`, il préfère *construire* une procédure appropriée, qu'il affecte au symbole `rho`.

```
> rho := proc(x, y)
    RETURN(sqrt(x^2 + y^2))
  end :
```

Notre utilisateur peut désormais appeler sa procédure, elle lui retournera le

résultat souhaité.

```
> rho(20,21) ;
```

$$29$$

```
> rho(a, b+c) ;
```

$$a^2 + (b+c)^2$$

Exploitons cette procédure, simple mais déjà utile, pour expliquer les notions les plus élémentaires.

6.2.1 Une procédure est un objet.

Une procédure Maple est un objet Maple comme un autre, de type **procedure**. Le plus souvent cet objet est accessible par l'intermédiaire d'un symbole, dans notre exemple par le symbole `rho`. Ici le symbole `rho` est présent dans l'environnement, et sa valeur est l'objet procédure qui nous intéresse. L'analogie est parfaite entre notre `rho` et le `sigma` ci-dessous.

```
> sigma := 123.456 ;
```

$$123.456$$

```
> eval(sigma, 1) ; eval(rho, 1) ;
```

$$123.456$$

$$\mathbf{proc}(x, y)\ \mathrm{RETURN}(\mathrm{sqrt}(x\hat{}2 + y\hat{}2))\ \mathbf{end}$$

Le symbole `sigma` est aussi présent dans l'environnement et sa valeur est l'objet **float** 123.456.

La forme output d'une procédure est un texte utilisant diverses polices et indentations pour tenter d'en faciliter la lecture. Notre procédure *valeur* de `rho` n'est accessible que par évaluation *explicite*, voir à ce sujet la règle Eval-16.

```
> rho ;
```

$$\rho$$

```
> eval(rho) ;
```

$$\mathbf{proc}(x, y)\ \mathrm{RETURN}(\mathrm{sqrt}(x\hat{}2 + y\hat{}2))\ \mathbf{end}$$

```
> whattype(rho), whattype(eval(rho)) ;
```

$$symbol, procedure$$

Un objet composé, comme une liste, une table, une matrice peut parfaitement avoir comme composante une procédure. Ne pas croire qu'une telle situation est artificielle, elle ouvre des possibilités de *programmation fonctionnelle* très intéressantes qu'on examinera plus tard, élargissant considérablement le domaine de la programmation. Par exemple notre procédure repérée par le symbole `rho` peut être un élément de liste :

```
> lst := [a, eval(rho), b] ;
```

$$lst := [a, \mathbf{proc}(x, y)\ \mathrm{RETURN}(\mathrm{sqrt}(x\hat{}2 + y\hat{}2))\ \mathbf{end}, b]$$

La procédure installée ainsi bizarrement en deuxième position de la liste `lst` est

néanmoins utilisable telle quelle.

```
> lst[2](20, 21) ;
                                    29
```

Dans le même registre, considérer l'instruction :

```
> proc(x,y) RETURN(sqrt(x^2+y^2)) end (20,21) ;
                                    29
```

L'expression encadrée par les mots-clés `proc` et `end`, à considérer comme parenthèses ouvrante et fermante, est un *constructeur de procédure* pouvant figurer en toute place où, *après évaluation*, un objet procédure peut figurer. Une fois cette procédure construite, Maple observe qu'elle doit être appliquée à la paire d'arguments 20 et 21, et le fait. Le résultat est donc une nouvelle fois notre hypoténuse 29.

6.2.2 Nom d'une procédure.

Dans l'environnement en cours, la *valeur* du symbole `rho` est un objet **procedure**, celui qui sait calculer les hypoténuses. Dans une telle situation, *par abus de langage*, on dit que le *nom de la procédure* est `rho` ; de façon encore plus expéditive, et strictement parlant c'est encore plus incorrect, on dit «la procédure `rho`». Ceci dit, ces raccourcis sont commodes, rarement vraiment gênants, et ils ont été déjà très souvent utilisés dans ce texte. Ces approximations supposent implicitement que chaque objet procédure est repéré exactement par un symbole, mais il est facile de mettre en défaut cette assertion. Par exemple on peut affecter la même procédure aux dix symboles `psi1`, ..., `psi10` :

```
> psi.(1..10) := seq(eval(rho), i=1..10) :
> eval(psi3) ;
                    proc(x, y) RETURN(sqrt(x^2 + y^2)) end
> psi6(20,21) ;
                                    29
```

On a donc maintenant dans notre environnement onze symboles ayant pour valeur le même objet procédure, et il n'est donc pas correct de dire que «son» nom est l'un ou l'autre de ces symboles. On méditera en particulier les comparaisons suivantes :

```
> evalb(psi3 = psi6) ;
                                   false
> evalb(eval(psi3) = eval(psi6)) ;
                                   true
```

Inversement une procédure peut être *anonyme*, autrement dit sans nom. On verra plus tard que l'instruction suivante crée 100 procédures créées l'une après l'autre par la procédure prédéfinie `unapply`, chacune élevant son argument à une certaine puissance :

```
> ProcedureList := [seq(unapply(x^i, x), i=1..100)] :
```

On montre la 99-ième, puis on essaie la dixième sur 2 et la vingtième sur 3 :

```
> ProcedureList[99] ;
```

$$(x \to x)^{99}$$

```
> ProcedureList[10](2) ;
```

$$1024$$

```
> ProcedureList[20](3) ;
```

$$3486784401$$

Notre environnement contient donc maintenant cent procédures supplémentaires, toutes *anonymes*. Elles sont néanmoins accessibles par l'intermédiaire du symbole `ProcedureList`, qui pointe sur la liste de ces procédures. On peut faire travailler tel élément de la liste sur tel entier comme montré sur les exemples, où 2^{10} et 3^{20} sont calculés.

6.2.3 Appel d'une procédure.

Un appel d'une procédure est une expression :

```
obj0(obj1, obj2, ..., objn)
```

De façon plus précise la présence *contiguë* d'un objet et d'une parenthèse ouvrante, sans séparateur, note un objet de type **function**, même si aucune procédure n'est en piste :

```
> whattype(truc(chose, machin)), whattype(truc()), whattype(truc) ;
```

function, *function*, *symbol*

Les objets figurant entre parenthèses sont les arguments (provisoires) ; il peut ne pas y en avoir, mais les parenthèses restent indispensables pour «mériter» le titre **function** ; comparer ci-dessus les types de `truc()`, avec parenthèses, et de `truc`, sans parenthèses.

De la même façon, et c'est relativement fréquent, une procédure peut être *sans paramètre* ; quand elle sera appelée, il ne faudra donc donner aucun argument, mais les parenthèses doivent néanmoins figurer dans l'appel. Un exemple de procédure sans argument est celle qui est *retournée* par la procédure `rand`, qu'on a déjà utilisée plusieurs fois, voir par exemple page 40 pour initialiser une matrice «aléatoire». Supposons que le contexte nécessite fréquemment le «tirage au sort» d'un entier entre -5 et +5. La procédure `rand` est disponible pour *construire* et *retourner* une procédure capable d'effectuer ces tirages ; une telle procédure est appelée un *générateur aléatoire*. Le plus simple consiste alors à affecter la procédure ainsi construite à un symbole, par exemple `rnd5` :

```
> rnd5 := rand(-5..+5) ;
```

$$rnd5 := \mathbf{proc}\boxed{()}$$
$$\mathbf{local}\, t; \mathbf{global}\, \_seed;$$
$$\_seed := \mathrm{irem}(427419669081 * \_seed, 999999999989);$$
$$t := \_seed;$$
$$\mathrm{irem}(t, 11) - 5$$
$$\mathbf{end}$$

On a encadré sur le listing ci-dessus, les parenthèses ouvrante et fermante qui se succèdent sans paramètre intermédiaire dans la forme externe de l'objet **procedure** construit et retourné par `rand`. La procédure «de nom `rnd5`» est donc une procédure sans paramètre qu'on peut, ou plutôt qu'on *doit*, appeler comme suit :

```
> rnd5() ;
```

4

à comparer avec :

```
> rnd5 ; eval(rnd5) ;
```

$$rnd5$$
$$rnd5 := \mathbf{proc}()$$
$$\mathbf{local}\, t; \mathbf{global}\, \_seed;$$
$$\_seed := \mathrm{irem}(427419669081 * \_seed, 999999999989);$$
$$t := \_seed;$$
$$\mathrm{irem}(t, 11) - 5$$
$$\mathbf{end}$$

où l'absence de parenthèse *ouvrante* derrière `rnd5` exclut un appel de la procédure repérée par ce symbole, alors qu'au contraire la procédure `eval` a travaillé.

6.3 Invocations multiples et programmation *récursive.*

L'exemple prototype de la *programmation récursive* est la programmation de la fonction factorielle

```
> fact := proc(n::nonnegint)
    if n = 0 then RETURN(1)
             else RETURN(n*fact(n-1))
    fi
  end :
> fact(4) ;
```

24

Ici, l'évaluateur appelle la procédure `fact` en lui passant l'argument `4` ; puisque cet argument n'est pas nul, c'est la branche négative de l'instruction `if` qui est exécutée, à savoir `RETURN(n*fact(n-1))` ; l'exécution de cette instruction nécessite en particulier l'évaluation de `fact(3)` et l'évaluateur va donc *ré*-appeler la procédure `fact`, cette fois avec l'argument `3` ; l'*invocation* initiale de `fact` est mise au repos,

et une nouvelle invocation est activée, mais où cette fois l'argument passé est `3`. Le processus continue de la sorte jusqu'à ce que cinq invocations soient actives, appelées respectivement avec les arguments `4`, `3`, `2`, `1` et `0`. Pour cette dernière invocation c'est au contraire la branche positive de l'instruction `if` qui va être exécutée et l'instruction `RETURN(1)` est donc exécutée. Cette exécution met fin à la dernière invocation de `fact`, invocation qui *retourne* `1`. La branche négative de l'invocation précédente, essentiellement `RETURN(1*fact(0))` peut alors être terminée ; l'appel `fact(0)` a retourné `1` et cette «avant-dernière» invocation de `fact` va donc aussi retourner `1`. La troisième invocation est réactivée, qui était arrêtée par l'instruction `RETURN(2*fact(1))` ; la quatrième invocation de `fact` vient justement de retourner `1`, et la troisième va donc elle aussi terminer son exécution et retourner `2`. Selon le même processus, la seconde invocation de `fact` va retourner `6` et enfin la première va retourner `24`.

On peut mettre en évidence ce dispositif «à tiroirs» en demandant d'exécuter la procédure `fact` en mode `debug`.

```
> debug(fact) ;
```

fact

```
> fact(4)
{--> enter fact, args = 4
{--> enter fact, args = 3
{--> enter fact, args = 2
{--> enter fact, args = 1
{--> enter fact, args = 0
<-- exit fact (now in fact) = 1}
<-- exit fact (now in fact) = 1}
<-- exit fact (now in fact) = 2}
<-- exit fact (now in fact) = 4}
<-- exit fact (now at top level) = 24}
```

24

Si une procédure est en mode `debug`, chaque invocation de cette procédure va être signalée, et les arguments passés sont affichés. Inversement, quand une invocation termine, l'objet retourné par cette invocation est aussi affiché, et le debugger vous indique de plus quelle est la procédure qui reprend son activité, ou le «`top level`» si aucune procédure n'est plus active.

La variante suivante, artificielle, montre le cas de procédures *mutuellement récursives*.

```
> evenfact := proc(n::And(nonnegint, even))
    if n = 0 then RETURN(1)
             else RETURN(n*oddfact(n-1))
    fi
  end :
  oddfact := proc(n::And(posint, odd))
    RETURN(n*evenfact(n-1))
  end :
  debug(evenfact, oddfact) ;
```

evenfact, *oddfact*

```
> evenfact(4)
{--> enter evenfact, args = 4
{--> enter oddfact, args = 3
{--> enter evenfact, args = 2
{--> enter oddfact, args = 1
{--> enter evenfact, args = 0
<-- exit evenfact (now in oddfact) = 1}
<-- exit oddfact (now in evenfact) = 1}
<-- exit evenfact (now in oddfact) = 2}
<-- exit oddfact (now in evenfact) = 6}
<-- exit evenfact (now at top level) = 24}
```

$$24$$

Il est presque toujours nécessaire, quand les expériences souhaitées sont terminées, de *désactiver* le mode `debug` d'une procédure. Il faut alors utiliser la procédure `undebug` en donnant comme argument(s) le(s) nom(s) de procédure(s) en question.

```
> undebug(fact)
```

$$fact$$

```
> fact(5) ;
```

$$120$$

La programmation récursive d'une procédure est souvent commode, mais on démontre qu'elle n'est jamais indispensable : une version «purement itérative» est toujours possible. Ainsi la fonction factorielle peut aussi être programmée :

```
> itfact := proc(n::nonnegint)
    local i, result ;
    result := 1 ;
    for i from 1 to n do
      result := i*result
    od ;
    RETURN(result)
  end :
> seq(itfact(i), i=0..10) ;
```

$$1, 1, 2, 6, 24, 120, 720, 5040, 40320, 362880, 3628800$$

En particulier, et c'est un élément à prendre souvent en compte, une procédure programmée récursivement peut nécessiter des invocations si nombreuses qu'elles peuvent finir par mettre en échec les ressources du système. Comparer.

```
> floor(log[10](fact(600))) + 1 ;
Error, (in fact) too many levels of recursion
> floor(log[10](itfact(600))) + 1 ;
```

$$1409$$

On compare nos deux programmations «sensées» de la fonction factorielle, à l'aide de la procédure `fact`, méthode récursive, ou de la procédure `itfact`, méthode itérative. Pour mener à bien cette comparaison, on essaie le calcul de `600!`. C'est un nombre un peu encombrant : 1409 chiffres décimaux, et le résultat supposé obtenu, on ajoute ce qu'il faut pour ne faire apparaître justement que ce nombre

de chiffres. D'une façon générale, le nombre de chiffres c «avant la virgule» d'un nombre décimal x est obtenu par la formule mathématique $c = E(\log_{10}(x)) + 1$ où E est la fonction *partie entière*, `floor` en langage Maple :

```
> floor(log[10](100.000)) + 1 ;
                                    3
> floor(log[10](99.999)) + 1 ;
                                    2
> floor(log[10](0.00123)) + 1 ;
                                   -2
```

On voit ainsi que la programmation *récursive* de la fonction factorielle échoue. Ne pas croire inversement que le style récursif est toujours «déconseillé». Il est très fréquent que la programmation récursive d'une procédure est particulièrement commode et/ou lisible. Ainsi ci-dessus, il faut déjà un peu de lucidité pour comprendre que l'itération dans le calcul de `result` doit *impérativement* commencer à 1. Voir aussi la programmation récursive de la procédure `phi` page 149, commode, alors que sa programmation itérative est plus difficile et moins lisible, voir page 152.

Bien que ce ne soit jamais indispensable, les programmeurs expérimentés savent qu'il faut toujours mieux penser d'abord, quand on programme récursivement, au(x) cas qui va (vont) *terminer* la récursivité, c'est-à-dire celui (ceux) qui *commence(nt)* la définition récursive de la fonction à programmer. C'est le style qui a été systématiquement adopté ici.

6.4 Paramètres et arguments.

C'est l'une des questions les plus difficiles de la programmation. Les solutions adoptées par les différents langages sont en général très différentes. S'il est le plus souvent facile de passer d'un langage de programmation à un autre pour ce qui concerne les programmations itératives et conditionnelles, il n'en est pas de même pour la programmation procédurale ; les styles sont en effet en la matière si différents que des bugs apparemment incompréhensibles sanctionnent très vite ceux qui se contentent d'une étude superficelle de la question quand ils changent de langage.

Ceci est dû à ce que beaucoup de solutions sont possibles pour établir la correspondance entre *paramètres* et *arguments* ; l'espace de travail est si vaste qu'il est impossible d'inclure dans un langage «toutes» les solutions possibles ; en inclure beaucoup rend d'ailleurs le langage techniquement difficile, de sorte que les concepteurs, toujours angoissés à l'idée que leur langage soit jugé rébarbatif et donc peu utilisé, hésitent à travailler en ce sens.

Maple est assez particulier de ce point de vue et un minimum de travail doit être prévu pour qu'un programmeur Maple domine suffisamment le sujet pour un usage disons professionnel. Trois points méritent en effet d'être soigneusement considérés.

1. Le passage des arguments s'effectue «normalement» par *valeur*, notion qu'il faut donc avoir comprise, mais la transmission de la valeur elle-même se fait par *substitution* et non par... valeur ; le lecteur est sans doute perplexe à la lecture de ces technicités, mais quand il aura compris, il saura identifier certains bugs pas si rares et franchement difficiles à élucider.
2. La règle Eval-14 implique l'évaluation préalable des arguments, voir Section 2.4.2. Ceci dit, ceci est modifiable par l'utilisateur, et il peut être utile de connaître cette possibilité. C'est par ce biais que Maple permet le passage éventuel *par référence*.
3. Maple permet aussi des appels de procédure avec un nombre *variable* d'arguments. La technique en est assez primitive, mais peut néanmoins être considérée pour permettre des appels de procédures «à géométrie variable».

6.4.1 Le passage standard d'argument.

Si un *paramètre* de procédure est déclaré sans type, ou bien avec un type «ordinaire», le mécanisme de passage d'argument est le suivant. L'argument *provisoire* correspondant est évalué et le résultat de l'évaluation est appelé l'argument *définitif*. Quand la procédure est invoquée, un travail préalable de *substitution* est effectué : toute *occurence* du paramètre dans le *texte* de la procédure est remplacée par l'argument définitif.

L'illustration la plus simple va comme suit :

```
> rho := proc(x, y)
    RETURN(sqrt(x^2 + y^2))
  end :
> a, b, c := 3, 4, 5 ;
```

$$a, b, c := 3, 4, 5$$

```
> rho(a, b+c) ;
```

$$3\sqrt{10}$$

Ici, `x` et `y` sont des paramètres sans spécification de type, et le passage d'argument par valeur et substitution va donc s'appliquer. Quand la procédure `rho` est appelée, deux arguments *provisoires* sont prévus, le symbole `a` et l'expression `b+c`. L'évaluateur de Maple est donc mis d'abord deux fois au travail pour produire les arguments définitifs, donc dans ce cas, vu les affectations préalables aux symboles `a`, `b` et `c`, les arguments définitifs vont être `3` et `9`.

Cette phase préalable étant réalisée, *avant* d'appeler la procédure `rho`, Maple fait encore un autre travail : il *substitue* dans le texte de la procédure toute occurence de `x` par `3` et toute occurence de `y` par `9`. Le *texte* de la procédure pour la nouvelle invocation devient donc :

```
  RETURN(sqrt(3^2 + 9^2))
```

Ce texte-là est alors invoqué, et la procédure retourne donc l'objet `3*sqrt(10)` affiché en output $3\sqrt{10}$.

Il est important ici de comprendre qu'il y a bien eu un mécanisme de substitution de l'argument au paramètre, et non pas de mise-à-jour de l'environnement à l'aide des arguments, comme il est si tentant de le croire. Beaucoup de langages, C et Lisp notamment, modifient l'environnement visible de l'intérieur de la procédure pour y ajouter dans un tel cas de figure de nouveaux chaînages `[x` $\rightarrow$ `3]` et `[y` $\rightarrow$ `9]`, cachant éventuellement des chaînages externes de même source. Maple est de ce point de vue *très différent*. Pour s'en convaincre, il faut faire deux expériences.

La première est à l'origine des plus mémorables crises de nerfs des utilisateurs de Maple. Les calculs très élémentaires suivants peuvent être effectués «hors procédure» :

```
> a := 5 ; a := a+a ;
```

$$a := 5$$
$$a := 10$$

```
> a := b ; a := a + a ;
```

$$a := b$$
$$a := 2b$$

Ces calculs sont si élémentaires que le lecteur s'interroge probablement sur leur intérêt ; mais considérez la version suivante des mêmes calculs à l'intérieur d'une procédure.

```
> restart :
  exp1 := proc(a)
    a := a + a ;
    RETURN(a)
  end :
> exp1(5) ;
Error, (in exp1) illegal use of a formal parameter
> exp1(b) ;
```

$$b$$

```
> b ;
Error, too many levels of recursion
```

Si vous n'avez pas d'explication, vous aurez probablement les plus grandes peines à comprendre ces résultats. Celui qui «croit» à un passage par valeur et affectation à une variable `a` interne à la procédure est bien incapable d'expliquer ces phénomènes si étranges. Il faut absolument interdire cette interprétation, et revenir à la seule valide, le mécanisme de substitution. Quand la procédure `exp1` a été appelée avec l'argument `5`, le *texte* de la procédure est devenu :

```
  5 := 5 + 5 ;
  RETURN(5)
```

de sorte que l'instruction d'affectation est devenue erronée : le calcul de `5 + 5` ne pose pas de problème, par contre l'affectation à l'entier `5` n'a pas de sens, ce qui explique le premier message d'erreur. Noter que Maple sait vous indiquer que l'erreur provient de l'usage de ce qui est appelé dans la documentation Maple *formal*

parameter, que nous appelons simplement *paramètre*. Cet usage n'est pourtant pas si «illégal» que ça, comme on va le voir immédiatement.

Quand `exp1(b)` est appelé, comme `b` est alors un symbole libre, l'argument définitif est aussi le symbole `b`, de sorte que le *texte* de la procédure invoquée devient :

```
b := b + b ;
RETURN(b)
```

Cette fois l'instruction d'affectation est tout-à-fait légale et est véritablement exécutée comme prévu ; en particulier un chaînage `[b` → `2*b]` est installé dans l'environnement. On s'attend donc à ce que `RETURN(b)` retourne `2*b` ? Non, c'est seulement `b` qui est retourné ! Ici s'intercale une autre règle de fonctionnement dans les procédures : les résultats de substitution installés à l'invocation de la procédure *ne sont pas* réévalués à l'exécution. C'est pourquoi c'est `b` qui est «brutalement» retourné et non pas `2*b` comme s'y attend le profane. De toute façon il avait tort, car l'évaluation ultérieure de `b` *hors procédure* a provoqué une erreur de récursion : en effet la valeur de `b` est `2*b` *qui est réévaluable*, voir la règle Eval-7, donnant `4*b`, donnant `8*b`, ..., finissant par donner une erreur de récursion non terminable.

Conseil 28 — *Bien que ce ne soit pas toujours* erroné*, au sens strict du terme, ne cherchez pas à modifier la «valeur» d'un paramètre par une instruction d'affectation, notion qui est en fait* non sens*. Si vous avez vraiment besoin de faire varier cette pseudo-valeur, doublez ce paramètre d'une variable locale et commencez par initialiser cette variable comme il se doit.*

Soit à construire une procédure permettant d'échanger les valeurs de deux symboles. On avait vu page 5.3.1 que, *hors procédure*, la méthode la plus simple consiste à utiliser une affectation multiple.

```
> a, b := 2, 3 ;
```
$$a, b := 2, 3$$
```
> a, b := b, a ;
```
$$a, b := 3, 2$$
```
> a, b ;
```
$$3, 2$$

Écrire une procédure `swap` réalisant la même chose d'une façon sensiblement plus lisible est une idée naturelle.

```
> a, b := 2, 3 ;
```
$$a, b := 2, 3$$
```
> swap(a, b) ;
> a, b ;
```
$$3, 2$$

Reste à écrire la procédure `swap`. L'application «directe» de la méthode hors procédure échoue pour la raison expliquée.

```
> swap := proc(x, y)
    x, y := y, x ;
    RETURN(NULL)
  end :
> a, b := 2, 3 ;
```

$$a, b := 2, 3$$

```
> swap(a,b) ;
Error, (in swap) illegal use of a formal parameter
```

Notre conseil suivi sans réfléchir ne donne pas la solution.

```
> swap := proc(x, y)
    local xx, yy ;
    xx, yy := x, y ;
    xx, yy := yy, xx ;
    RETURN(NULL)
  end :
> swap(a,b) ;
> a, b ;
```

$$2, 3$$

Cette fois l'erreur est évitée, mais la procédure est sans effet sur les affectations de `a` et `b` ; seules les affectations *internes* à la procédure ont été modifiées. Il faut donc *nécessairement* s'organiser pour passer les symboles eux-mêmes et non pas leur valeur. C'est un peu audacieux, car le conseil ci-dessus n'est plus respecté. De plus l'appel de `swap` doit citer les symboles *quotés*, car leur évaluation doit à tout prix être empêchée. D'où cette (tentative de) solution.

```
> swap := proc(x::symbol, y::symbol)
    x, y := y, x ;
    RETURN(NULL)
  end :
> swap(a, b) ;
Error, swap expects its 1st argument, a, to be of type symbol, but received 2
> swap('a', 'b') ;
> a, b ;
Error, too many levels of recursion
```

Il est judicieux ici de déclarer les paramètres `symbol`[1] de façon à intercepter l'erreur de l'utilisateur qui oublie de quoter les symboles dans l'appel de la procédure `swap`, comme montré dans le premier appel de cette nouvelle forme de la procédure. Dans le deuxième appel les quotes ne sont pas oubliés, et l'appel apparemment réussit, mais la vérification engendre une nouvelle erreur, parce que, compte tenu du fait que les arguments substitués *ne sont pas réévalués*, on a en définitive installé les chaînages `[a` $\rightarrow$ `b]` et `[b` $\rightarrow$ `a]` dans l'environnement qui provoquent l'erreur de récursion déjà observée page 2.3.2 :

```
> eval([a,b], 1) ;
```

$$[b, a]$$

[1] Une déclaration `name` est plus logique et a une portée plus générale.

Une «bonne» (?) solution est donc la suivante :

```
> a, b := 2, 3 ;
```

$$a, b := 2, 3$$

```
> swap := proc(x::symbol, y::symbol)
    x, y := eval(y), eval(x) ;
    RETURN(NULL)
  end :
> swap('a', 'b') ;
> a, b ;
```

$$3, 2$$

Mais cette «solution» est peu élégante : les quotes toujours nécessaires dans l'appel sont franchement pénibles. C'est précisément dans une telle situation qu'il faut songer à l'autre méthode disponible de passages d'arguments, qu'on examinera un peu plus loin.

6.4.2 Piège à base de mots réservés.

On examine dans cette section un type de bug assez pervers dû au problème, présent dans tous les langages, des *mots réservés*, mais qui induit sous Maple, à cause du mécanisme de *substitution* paramètre-argument, des bugs très étranges. Typiquement, le symbole `list` est réservé, toute tentative d'affectation à ce symbole est refusée :

```
> list := [1,2,3,4] ;
Error, attempting to assign to `list` which is protected
```

ce qui oblige l'utilisateur en panne d'inspiration de choisir des variantes comme `list_` (le blanc souligné est légal), `lst`, etc. C'est un tout petit peu désagréable, de sorte que quand on découvre qu'au contraire, pour les paramètres ou les symboles locaux de procédures, ces symboles redeviennent possibles, on est ravi de cette liberté retrouvée.

```
> list_list := proc(list)
    RETURN([op(list), op(list)])
  end :
> list_list([1,2,3]) ;
```

$$[1, 2, 3, 1, 2, 3]$$

Cette procédure exercice met bout à bout deux exemplaires de la liste argument. En mal d'inspiration pour le nom du paramètre, pourquoi ne pas choisir simplement `list` ? Or ceci est tout-à-fait légal pour un nom de paramètre, comme on peut le voir. Les utilisateurs prennent ainsi rapidement l'habitude d'appeler `set` l'unique paramètre ensemble, `matrix` l'unique paramètre matrice, etc. Mais considérons le petit exercice suivant. Soit à écrire une procédure éliminant les doublons d'une liste d'entiers, et triant la liste d'entiers résultante.

```
> clean := proc(list)
    RETURN(sort(convert(convert(list, set), list)))
  end :
```

La conversion en ensemble (`set`) élimine automatiquement les éventuels doublons, voir page 108 ; ceci fait, on reconvertit l'ensemble obtenu en liste, qu'enfin on trie. Essai :

```
> clean([2,1,2,3]) ;
Error, (in clean) wrong number (or type) of parameters in function convert
```

La procédure `convert` est en délicatesse. Un examen soigneux montre immédiatement la confusion : le deuxième argument `list` du `convert` externe est évalué avant d'être utilisé, car `convert` est une procédure standard. L'erreur est facile à corriger : pour empêcher cette évaluation intempestive, il suffit de quoter cette occurence-là du symbole `list` :

```
> clean := proc(list)
    RETURN(sort(convert(convert(list, set), 'list')))
  end :
> clean([2,1,2,3]) ;
Error, (in clean) wrong number (or type) of parameters in function convert
```

et on a la mauvaise surprise de constater que l'erreur est exactement la même, de quoi rester perplexe ! C'est le moment où jamais de penser à un moyen de debugging franchement «rustique», mais qu'il ne faut jamais oublier, consistant à exécuter soi-même, hors procédure, une à une, les instructions fautives ; mais ici c'est impossible, car il n'est pas légal, hors procédure, d'affecter une valeur au symbole `list` ; à moins de lever la protection sur le symbole `list`, ce qui donne l'essai suivant.

```
> list := [2,1,2,3] ;
Error, attempting to assign to `list` which is protected
> unprotect(list) ;
> list := [2,1,2,3] ;
```

$$list := [2, 1, 2, 3]$$

```
> sort(convert(convert(list, set), 'list')) ;
```

$$[1, 2, 3]$$

et on constate que cette fois l'erreur ne se produit pas. Nous sommes là décidément dans un cas de debugging plus que sévère ! Mais que peut bien expliquer tout cela ?

L'erreur commise ici résulte d'une incompréhension fondamentale du mécanisme de transmission argument vers paramètre. Il est *erroné* de penser que le paramètre est un *symbole* ayant, lors d'une évaluation, telle *valeur* déduite de l'argument définitif. Ce n'est pas le bon point de vue, et c'est cette erreur de «point de vue» qui engendre ces birarreries en série. En fait le paramètre est une composante de texte, et l'argument est *substitué* à cette composante de texte paramètre à l'invocation de la procédure. Pour provoquer *hors procédure* la même erreur, il

faut procéder comme suit :

```
> eval(subs('list'=list,
       'sort(convert(convert(list, set), 'list'))')) ;
Error, wrong number (or type) of parameters in function convert
```

ou, en style plus décomposé :

```
> subs('list'=list,
       'sort(convert(convert(list, set), 'list'))') ;
```

$$\mathrm{sort}(\mathrm{convert}(\mathrm{convert}([2,1,2,3], set),' [2,1,2,3]'))$$

```
> eval(%) ;
Error, wrong number (or type) of parameters in function convert
```

et on observe là une propriété logique mais franchement perverse du mécanisme de substitution : la substitution est appliquée *même dans les expressions quotées*, ce qui simule vaguement une «évaluation à tout prix», même «sous» les quotes ! Ci-dessus la substitution a remplacé `'list'` par `'[2, 1, 2, 3]'`, donnant «l'impression» que `list` a été évalué.

Pour mettre en évidence ce phénomène «sous» procédure, considérons l'exemple artificiel mais décisif :

```
> test := proc(parameter)
    local var ;
    var := 'parameter' ;
    RETURN('parameter')
  end :
> debug(test) ;
```

$$test$$

```
> test(argument) ;
{--> enter test, args = argument
```

$$var := argument$$

```
<-- exit test (now at top level) = argument}
```

$$argument$$

On voit ici que les quotes n'ont pas empêché la substitution de `parameter` par `argument`.

Revenons à notre exemple initial. La seule façon de le debugger si on veut garder le même style de programmation consiste à *ne pas utiliser* le symbole `list` comme paramètre. On commence d'abord par remettre le symbole `list` dans son état normal, sans valeur et protégé.

```
> unassign('list') ; protect(list) ;
> clean := proc(lst)
    RETURN(sort(convert(convert(lst, set), list)))
  end :
> clean([2,1,2,3]) ;
```

$$[1,2,3]$$

Autre méthode bien amusante, mais qu'on ne peut pas décemment conseiller :

```
> clean := proc(list)
    RETURN(sort(convert(convert(list, set), li.st)))
  end :
> clean([2,1,2,3]) ;
```

$$[1, 2, 3]$$

On a réussi ainsi à *cacher* le symbole `list` par l'intermédiaire de l'expression `li.st` ; Maple ne voit pas là en effet dans le *texte* de la procédure une occurence du symbole `list`, mais une expression de concaténation sur les deux symboles `li` et `st` ; quand, *plus tard*, cette expression est évaluée, Maple a «oublié» la liaison de `list` vers `[2,1,2,3]`, car cette liaison, plus précisément la substitution paramètre-argument, n'est utilisée qu'à l'*invocation* de la procédure ; elle n'est plus utilisée ensuite.

Ces considérations semblent sans doute exagérément minutieuses, mais elles sont inévitables dans un langage où les symboles jouent un rôle double : d'une part, comme dans les langages ordinaires, repérer telle ou telle expression *valeur* de ce symbole, d'autre part servir d'*étiquette* pour donner telle ou telle indication, notamment une indication de variable «formelle» sans valeur, ou encore une indication de type. Vouloir faire jouer simultanément les deux rôles au symbole `list` est donc assez casse-cou, et c'est ce coup d'audace qui est à l'origine de ces bugs assez difficiles.

6.4.3 Exclure l'évaluation d'un argument.

Le langage Pascal prévoit la possibilité, option `var`, de passer à une procédure la *référence* d'une variable et non pas sa valeur ; ceci permet de modifier, de l'intérieur de la procédure, la *valeur*[2] de la variable passée. Par le jeu des pointeurs et références, les langages C et Java offrent des possibilités encore plus larges. Le langage Lisp sépare soigneusement les appels de fonctions des appels de macros, avec une structure beaucoup plus étudiée, beaucoup plus puissante aussi, mais nécessitant souvent quelque syntaxe ésotérique dans l'écriture des fonctions et macros, sans parler des macros génératrices de macros, notion hors de portée des autres langages. Maple est aussi plus ou moins de cette nature, en fait imposée par la nature «dynamique» des symboles : un symbole en Maple, comme en Matlab, comme en Lisp, reste «vivant» à l'exécution, sa valeur n'étant accessible que par un mécanisme d'*évaluation*, modélisant en fait une indirection pointeur.

C'est par l'intermédiaire d'un contrôle soigneux du mécanisme d'évaluation que l'utilisateur Maple pourra manipuler dans une procédure des chaînages variés *passés* en argument par l'intermédiaire de leur source. Le principe de base est assez simple : si un paramètre est déclaré `uneval`, l'argument correspondant à l'appel de la procédure ne sera pas évalué. La procédure disposera donc de la source potentielle d'un chaînage et pourra le modifier à sa guise. L'exemple prototype est bien sûr celui de la procédure `swap` dont la version présentée maintenant est idéale.

[2] En fait le contenu de la zone mémoire réservée pour cette variable.

```
> restart :
> swap := proc(x::uneval, y::uneval)
    x, y := eval(y), eval(x) ;
    RETURN(NULL)
  end :
> a, b := 2, 3 ;
```

$$a, b := 2, 3$$

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

$$3, 2$$

L'appel de la procédure `swap` est maintenant confortable : il n'est plus nécessaire de quoter les arguments, compte tenu de la déclaration `uneval` pour les paramètres de la procédure. Noter toutefois qu'un utilisateur appliquant le rituel *principe de précaution*, autrement dit quotant «à tout hasard» les arguments, a maintenant une erreur.

```
> swap('a', 'b') ;
Error, (in swap) illegal use of a formal parameter
```

car le mécanisme de *substitution*, combiné avec les déclarations `uneval`, engendre l'instrution suivante :

```
> 'a', 'b' := eval('a'), eval('b') ;
```

qui est évidemment erronée. L'exercice suivant pour le perfectionniste consiste donc à «cocooner» l'utilisateur de la procédure `swap` de telle façon que si jamais il quote à tort la variable à échanger, alors l'erreur est détectée et corrigée de l'intérieur de la procédure. Comment obtenir un tel résultat ? Il faut alors savoir *reconnaître* si l'argument est quoté ou non. L'objet quoté est de type `uneval` :

```
> whattype(''expr'') ;
```

uneval

Noter qu'il a été indispensable de quoter deux fois pour qu'il reste une expression quotée *après évaluation de l'argument*, car `whattype` est une procédure standard ; comparer avec :

```
> whattype('expr') ;
```

symbol

La nouvelle procédure `swap` est assez technique, typique de ce type de manipulation de chaînage, et va aussi mettre en évidence un caractère particulier aux symboles locaux.

```
> swap := proc(x::uneval, y::uneval)
    local xx, yy, xval, yval ;
    xx , yy := x, y ;
    if type(xx, uneval) then xx := op(1, xx) fi ;
    if type(yy, uneval) then yy := op(1, yy) fi ;
    if not type(xx, name) then ERROR(`%a should be a name`, xx) fi ;
    if not type(yy, name) then ERROR(`%a should be a name`, yy) fi ;
    xval, yval := eval(xx), eval(yy) ;
    print('arg1' = x, 'arg2' = y,
          'xx' = xx, 'yy' = yy,
          'xval' = xval, 'yval' = yval) ;
    assign(xx = yval, yy = xval) ;
    RETURN(NULL)
  end :
```

Pour bien faire comprendre le fonctionnement de cette nouvelle mouture de notre procédure, on a ajouté, juste avant que l'échange se réalise (instruction `assign`), un affichage de toutes les variables et paramètres en jeu. Exemples typiques de fonctionnement :

```
> a, b := 2, 3 ;
```

$$a, b := 2, 3$$

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

$$arg1 = a, arg2 = b, xx = a, yy = b, xval = 2, yval = 3$$

$$3, 2$$

```
> swap('a', 'b') ; a, b ;
```

$$arg1 = 'a', arg2 = 'b', xx = a, yy = b, xval = 3, yval = 2$$

$$2, 3$$

```
> swap(a, a+b) ;
Error, (in swap) a+b should be a name.
> lst := [[1,2], [3,4]] ;
```

$$lst := [[1, 2], [3, 4]]$$

```
> swap(lst[1,1], lst[2,2]) ; lst ;
```

$$arg1 = lst[1, 1], arg2 = lst[2, 2], xx = lst[1, 1], yy = lst[2, 2], xval = 1, yval = 4$$

$$[[4, 2], [3, 1]]$$

D'abord il a fallu utiliser des variables locales `xx` et `yy` pour pouvoir manipuler les arguments, surtout supprimer les quotes éventuels : ils sont repérés par `type(..., uneval)` et le cas échéant, ils sont supprimés par `xx := op(1, xx)`, qui extrait l'unique opérande, donc l'objet entre quotes. Demander `xx := xx` est insuffisant, parce que pour un symbole local, *un seul niveau d'évaluation est effectué*, de sorte que la valeur de `xx` resterait en fait inchangée.

Eval 29 — *Un symbole local à une procédure ne suit pas la règle Eval-7 ; quand il est évalué, le résultat est seulement le but du chaînage correspondant ; en particulier ce but n'est pas* réévalué.

Considérez l'exemple suivant, très exotique, pour mettre en évidence cette particularité des symboles locaux par rapport à l'évaluateur.

```
> restart ;
> c := d ;
```

$$c := d$$

```
> test := proc(x::uneval)
    global a ;
    local b ;
    a := b ;
    b := x ;
    print(a, b, eval(b))
  end :
> test(c) ;
```

$$d, c, d$$

Plusieurs chaînages de différents types sont installés. D'abord un chaînage banal entre symboles externes `[c → d]`. Puis le symbole `a` est déclaré global et a donc le statut d'un symbole externe mais visible de l'intérieur de la procédure, alors que le symbole `b` est «local» à la procédure ; les guillemets sont là pour préparer le lecteur à une nouveauté : on va néanmoins réussir à faire «survivre» ce symbole à l'extérieur de la procédure ! Dans la procédure, des chaînages `[a → b]` et `[b → c]` sont installés, car à l'appel de `test(c)`, le texte `x` est remplacé par le symbole `c`. Puis on regarde par curiosité les valeurs de `a`, `b` et `eval(b)`. On a la stupéfaction d'observer que dans le traitement des chaînages successifs de `a` vers `b` puis `c` puis `d`, si on «part» de `a` on va jusqu'à `d`, alors que si on part de `b` on s'arrête à `c` ! On voit néanmoins que `eval(b)` atteint `d`.

Pourquoi ? Dans la procédure, `b` est local et son évaluation arrête donc au but du chaînage correspondant ; c'est une conséquence de la règle Eval-29. Au contraire `a` est *global* et c'est la règle Eval-7 qui est appliquée : son évaluation va donc aussi loin que possible. Tout se passe comme si ce n'était pas le même évaluateur qui travaillait pour les symboles globaux et les symboles locaux ! Considérez encore ces tests ultérieurs, encore plus exotiques :

```
> a ;
```

$$d$$

```
> eval(a,1), eval(a,2), eval(a,3) ;
```

$$b, c, d$$

```
> evalb(b = b), evalb(b = eval(a,1)) ;
```

$$true, false$$

```
> {b, eval(a, 1)} ;
```

$$\{b, b\}$$

```
> addressof(%[1]), addressof(%[2]) ;
```

$$8857168, 8857184$$

Le symbole `a` avait été déclaré global et continue à vivre en dehors de la procédure, valeur conservée. Les `eval(a,-)` permettent de suivre la chaîne de... chaînages partant de `a`. En particulier le symbole `b`, en principe local à la procédure `test`, reste accessible de l'extérieur, par l'intermédiaire de `a`, alors que l'invocation de la procédure qui avait *créé* ce symbole est achevée ! Si on compare ce symbole `b` là à un symbole `b` «banal», on constate qu'ils sont *différents* ! Notre environnement

a maintenant *deux* symboles de même nom, dont l'un est accessible seulement par l'intermédiaire de `a`. On constate d'ailleurs que l'*ensemble* constitué de ces deux symboles a bien deux éléments : pas d'élimination de doublon puisqu'il n'y a pas de doublon ! On vérifie enfin que les adresses machines de ces symboles sont bien différentes.

Ces questions ne sont en aucune façon artificielle. Fréquemment quand une procédure retourne par exemple une matrice, on cite seulement le symbole repérant cette matrice, et en raison de la règle Eval-16, le symbole seul est retourné. À nouveau, des crises de nerfs sont proches si on compare ces deux exemples.

```
> testm := proc()
    local A ;
    A := matrix([[1,2],[3,4]]) ;
    RETURN(A)
  end :
> testm() ;
```

$$A$$

```
> evalm(A) ;
```

$$A$$

Dans ce premier essai, l'utilisation de `evalm` ne permet pas d'atteindre la matrice qui avait été affectée à `A`. Le point, c'est que le symbole retourné, local, est *différent* du symbole `A` argument de `evalm` ! Comparez avec cet autre essai :

```
> B := testm() ;
```

$$B := A$$

```
> evalm(B) ;
```

$$\begin{bmatrix} 1 & 2 \\ 3 & 4 \end{bmatrix}$$

Cette fois la valeur de `B` est le *bon* symbole `A`, de sorte que `evalm` atteint le but recherché. Avez-vous bien compris toutes ces choses ? Si oui, vous ne résisterez pas à tenter ceci :

```
> A := testm() ;
```

$$A := A$$

```
> eval(A,1) ;
```

$$A$$

```
> eval(A,2) ;
```

$$\begin{bmatrix} 1 & 2 \\ 3 & 4 \end{bmatrix}$$

où on joue à atteindre la matrice en partant du `A` global, en passant par le `A` local.

6.4.4 Nombre variable d'arguments.

Comme déjà expliqué, le mécanisme est rustique mais rend l'utilisateur absolument libre pour toute organisation. Pour expliquer rapidement ce dont il s'agit, rappelons-nous, voir page 112, que la procédure prédéfinie Maple `max` admet comme

argument une *suite* de réels ; l'utilisateur a l'impression, pas complètement justifiée, que cette procédure admet un nombre quelconque d'argument*s*. Programmons nous-même cette procédure pour montrer le principe.

```
> my_max := proc()
    local i, result ;
    print(nargs) ;
    print(args) ;
    result := -infinity ;
    for i from 1 to nargs do
      if args[i] > result then result := args[i] fi
    od ;
    RETURN(result)
  end :
> my_max() ;
```

$$0$$

$$-\infty$$

```
> my_max(4,6,5) ;
```

$$3$$

$$4, 6, 5$$

$$6$$

Le principe est fort simple. Deux variables *locales* sont disponibles dans toute procédure, `nargs` et `args`. Elles sont automatiquement initialisées par Maple à chaque invocation de la procédure. La première, `nargs`, a pour valeur le nombre d'arguments passés à l'appel de cette invocation, ou plus précisément la longueur de la *suite* qui est en fait l'unique argument véritable, voir ce qui a déjà été expliqué à propos des suites d'arguments Section 4.4.

Bien comprendre que ce mécanisme est *disjoint* de l'utilisation standard des arguments, à l'aide de la liste de paramètres `proc(prm1, prm2, ...)`. Ici, dans la procédure `my_max`, on n'a même prévu aucun paramètre, la liste d'arguments étant *entièrement* gérée par l'intermédiaire des variables `nargs` et `args`. Noter la programmation *complète* conssistant à considérer comme *légal* un appel de la procédure `my_max` sans argument, ce qui est d'ailleurs traité de la même façon par la procédure officielle `max`, essayez.

Pour rendre plus visible ce mécanisme, la programmation donnée ici de `my_max` affiche au dévut de l'activation de la procédure les valeurs des variables `nargs` et `args`.

Rien n'empêche de *combiner* le mécanisme banal à base de symboles paramètres et les informations disponibles par l'intermédiaire des variables implicites `nargs` et `args`.

Commençons par un exemple artificiel mais didactique.

```
> test := proc(prm1, prm2)
    local i ;
    printf("Cet appel de la procédure test a %a arguments.\n", nargs) ;
    printf("Arg1 = %a.\n", prm1) ;
    printf("Arg2 = %a.\n", prm2) ;
    for i from 3 to nargs do
      printf("Arg%a = %a.\n", i, args[i])
    od  ;
    printf("The end.")
   end :
> test(foo1, foo2, foo3, foo4) ;
Cet appel de la procédure test a 4 arguments.
Arg1 = foo1.
Arg2 = foo2.
Arg3 = foo3.
Arg4 = foo4.
The end.
```

On a passé quatre arguments à la procédure, dont deux ont été lus par l'intermédiaire des paramètres `prm1` et `prm2`, et les autres par celui de la variable `args` ; leur nombre a été déduit de `nargs`.

Comme application significative de ce mécanisme, reprenons le problème de la programmation récursive de la procédure `mylog10` de la page 149. On avait noté le danger, très réel, d'une variable *globale* `epsilon` définie par la procédure maîtresse `mylog10`, puis utilisée par la procédure moteur `phi` : si l'utilisateur a pour d'autres raisons une variable déjà baptisée `epsilon`, l'ancienne valeur de cette variable serait perdue ! Le nom de procédure `phi` n'est pas non plus bien fantastique et risque d'interférer avec un autre symbole. On pourrait utiliser ici pour mettre en évidence la dépendance de la seconde procédure un nom plus explicite à ce propos comme `‘mylog10/phi‘` ; Maple utilise souvent lui-même cette technique pour ses propres procédures. Autre problème : la précision associée à la variable `epsilon` n'est pas modifiable, alors qu'il serait bien plus intéressant de laisser à l'utilisateur la possibilité *éventuelle* de préférer une autre précision.

On va solutionner l'ensemble de ces problèmes comme suit :

```
> mylog10 := proc(x::positive)
    local phi, epsilon ;

    phi := proc(alpha::realcons, beta::realcons, gamma::positive)
      if gamma=1. then RETURN(alpha) fi ;
      if gamma<1. then RETURN(phi(alpha, -beta, 1/gamma)) fi ;
      if gamma>=10. then RETURN(phi(alpha+beta, beta, gamma/10.)) fi ;
      if alpha<>0. and abs(beta/alpha)<epsilon then RETURN(alpha) fi ;
      RETURN(phi(alpha, beta/2., gamma^2))
    end :
```

```
    if nargs = 2 then epsilon := args[2]
                 else epsilon := 1e-10
    fi ;
    RETURN(phi(0., 1., convert(x,float)))
    end :
> mylog10(2) ;
```

$$.3010299956$$

```
> mylog10(2, 1e-5) ;
```

$$.3010292053$$

```
> Digits := 20 :
> mylog10(2, 1e-20) ;
```

$$.30102999566398119522$$

L'utilisateur a maintenant la possibilité de donner un deuxième argument pour refuser la valeur par défaut de `epsilon`. La présence de l'argument optionnel est détectée par examen de `nargs`. Par ailleurs le fait d'avoir défini la procédure `phi` à l'intérieur de la procédure `mylog10` la rend invisible de l'extérieur ; la variable `epsilon` est maintenant locale mais néanmoins *visible* de la procédure `phi` ; tout usage global de `phi` et `epsilon` peut maintenant coexister sans interférence avec les usages *internes* à `mylog10` de ces symboles.

```
> Digits := 10 :
> phi, epsilon := 56, 0.01 ; mylog10(3) ; phi, epsilon ;
```

$$\phi, \varepsilon := 56, .01$$

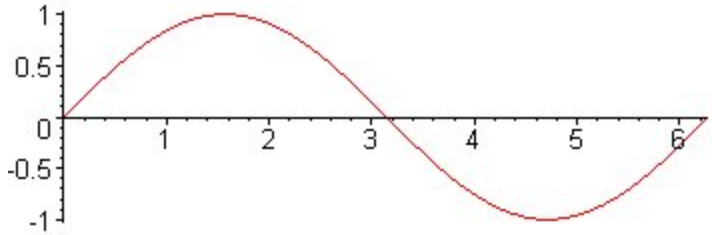
$$.4771212547$$

$$56, .01$$

6.4.5 Arguments à mots-clés.

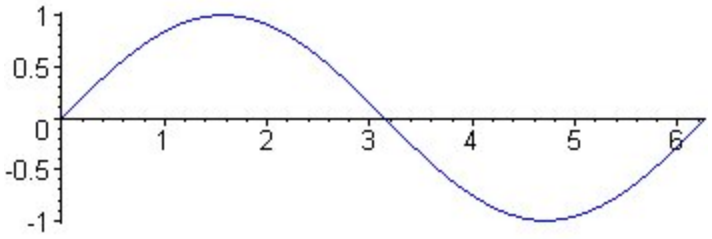
La variable `args` permet en particulier d'utiliser un mécanisme d'arguments à mots-clés, très utilisé par Maple lui-même dans ses propres procédures. Typiquement, quand vous utilisez la procédure `plot` pour tracer une courbe, la couleur par défaut de la courbe est rouge, mais vous pouvez modifier cette couleur par un argument à *mot-clé* `color`. Exemple de tracé sans couleur indiquée :

```
> plot(sin, 0..2*Pi) ;
```



Pour une raison ou une autre, cette couleur peut être jugée non satisfaisante ; Maple vous donne la possibilité d'un troisième argument, optionnel, permettant de demander une autre couleur.

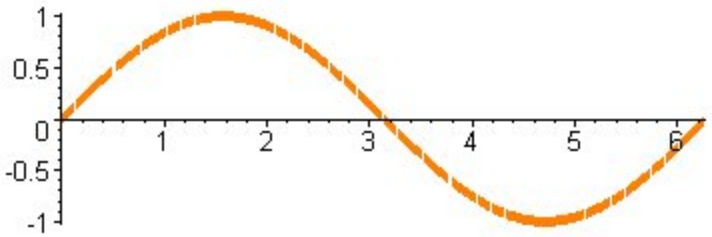
```
> plot(sin, 0..2*Pi, color=blue) ;
```



On a utilisé ici l'argument *à mot-clé* `color`. Noter que l'*unique* argument supplémentaire est l'objet de type **equation** `color=blue`, donc *un seul* objet à opérateur '`=`', et à opérande*s* `color` et `blue`. C'est la procédure `plot` qui analyse cet argument, en déduit notamment qu'il s'agit de modifier la couleur par défaut, et fait le nécessaire pour tenir compte de l'indication `blue` qui est donnée.

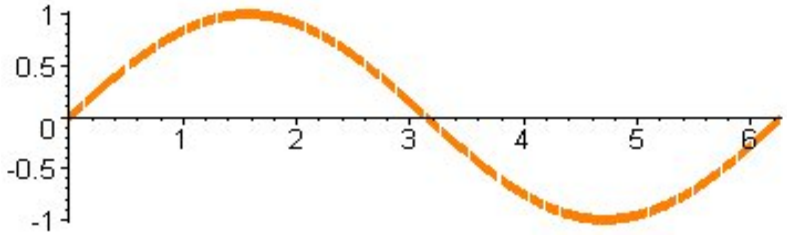
Un grand nombre d'arguments à mot-clé sont disponibles pour les procédures graphiques, donnant beaucoup de possibilités à l'utilisateur pour obtenir telle ou telle particularité dans le résultat. Par exemple on peut en plus souhaiter un tracé plus épais (argument à mot-clé `thickness`) et en trait non continu (argument à mot-clé `linestyle`).

```
> plot(sin, 0..2*Pi, thickness=5, color=coral, linestyle=3) ;
```



Voir la page de documentation «`plot,options`» pour avoir une idée des diverses possibilités proposées, assez vastes. Ce qui est très commode dans ces arguments *à mot-clé*, c'est le fait qu'ils sont utilisables en nombre arbitraire et dans un ordre arbitraire. Comparer le tracé précédent avec le suivant.

```
> plot(sin, 0..2*Pi, linestyle=3, color=coral, thickness=5) ;
```



Il serait en effet franchement désagréable de contraindre l'utilisateur à préciser *tous* les arguments optionnels et avec un ordre imposé ; on préfère définir ces arguments *par défaut*, et laisser à l'utilisateur la possibilité d'en modifier un nombre quelconque en général assez modeste. Comment obtenir un tel résultat dans l'écriture d'une procédure ?

Prenons par exemple le cas d'une procédure de traitement de liste assez souvent utile retournant la *position* de tel élément, premier argument, dans telle liste, deuxième argument. La programmation la plus simple peut-être la suivante.

```
> position := proc(element::anything, list::list)
    local i ;
    for i from 1 to nops(list) do
      if element = list[i] then RETURN(i) fi
      od ;
    RETURN('not_found')
  end :
> position(4, [3,6,4,7,9,4]) ;
                                        3
> position(44, [3,6,4,7,9,4]) ;
                                    not_found
```

Ceci dit un certain nombre d'options données à l'utilisateur pourraient augmenter sensiblement la portée de cette procédure. En particulier on peut donner la possibilité de préciser une fonction de comparaison différente de l'égalité usuelle pour la comparaison entre l'élément «à chercher» et les éléments de liste à examiner. On pourrait donc comme déjà fait utiliser un troisième argument optionnel ; si l'argument ne figure pas, on utilise la comparaison standard, sinon la comparaison proposée.

```
> position2 := proc(element::anything, list::list)
    local i, comp_func ;
    if nargs = 2 then comp_func := `=`
                 else comp_func := args[3]
    fi ;
    for i from 1 to nops(list) do
       if comp_func(element, list[i]) then RETURN(i) fi
    od ;
    RETURN('not_found')
  end :
> position2(4, [3,6,4,7,9,4]) ;
                                        3
> position2(4, [3,6,4,7,9,4],
            proc(x,y) (y-x) mod 2 = 0 end) ;
                                        2
```

Mais d'autres options sont tentantes pour le développeur. Par exemple on peut aussi demander de ne considérer qu'une sous-liste de la liste argument, considérant que le «reste» de la liste n'est pas concerné par la recherche de position. On peut aussi considérer plus intéressant dans certains cas de commencer par la fin. On voit que les combinaisons d'options peuvent être très variées et c'est justement le cas où il faut penser *arguments à mots-clés*.

On va ici prévoir quatre arguments à mot-clé :

– `cmpr` : pour une fonction de comparaison différente de la comparaison standard.

– `strt` : pour commencer l'examen à partir d'une position différente de 1.
– `endd` : pour finir l'examen avant la fin de la liste ; `end` est réservé et il faut un autre mot-clé.
– `rvrs` : pour préciser qu'il faut commencer l'examen par la fin.

Comment gérer cette variété de cas ? Une procédure auxiliaire de recherche de *valeur* éventuelle d'argument à mot-clé va nous aider. C'est une procédure à trois arguments, le mot-clé, la liste où la recherche est faite et la valeur par défaut si l'argument n'est pas trouvé.

```
> kw_arg := proc(keyword::symbol, list::list, default::anything)
    local i ;
    for i from 1 to nops(list) do
      if type(list[i], equation) and keyword = lhs(list[i])
        then RETURN(rhs(list[i]))
      fi
    od ;
    RETURN(default)
  end :
> kw_arg(strt, [endd=34, rvrs=true], 1) ;
                                        1
> kw_arg(strt, [strt=3, endd=34, rvrs=true], 1) ;
                                        3
```

Pour chaque élément de la liste à examiner, on regarde si cet élément est un objet **equation** et, si oui, si le membre de gauche (`lhs` = lefthand side) est le «bon» mot-clé. Le cas échéant, le membre de droite de l'équation est retourné. Sinon, si la liste est épuisée sans succès, on retourne la valeur par défaut indiquée en argument.

Maintenant la programmation de la procédure `position` est divisée en deux étapes : d'une part le traitement des options, d'autre part la programmation proprement dite. Une programmation *professionnelle* devrait prévoir tous les cas plausibles d'erreurs d'arguments, ce qui n'est pas notre sujet et n'a pas été fait ici. Noter d'ailleurs que le perfectionnisme en la matière est souvent dangereux dans un langage à la structure aussi floue que celui de Maple, le statut *véritable* des symboles échappant presque toujours à l'utilisateur ordinaire.

```
> position3 := proc(element::anything, list::list)
    local i, opts, lcmpr, lstrt, lendd, lrvrs, step ;
    opts := [args[3..-1]] ;
    lcmpr := kw_arg('cmpr', opts, `=`) ;
    lstrt := kw_arg('strt', opts, 1) ;
    lendd := kw_arg('endd', opts, nops(list)) ;
    lrvrs := kw_arg('rvrs', opts, false) ;
    step := +1 ;
    if lrvrs then lstrt, lendd, step := lendd, lstrt, -1 fi ;
    for i from lstrt to lendd by step do
      if lcmpr(element, list[i]) then RETURN(i) fi
    od ;
    RETURN(`not_found`)
  end :
> position3(4, [3,6,4,7,9,4]) ;
                                        3
> position3(4, [3,6,4,7,9,4],
            cmpr = ((x,y) -> (x-y) mod 2 = 0)) ;
                                        2
> position3(4, [3,6,4,7,9,4], strt=4) ;
                                        6
> position3(4, [3,6,4,7,9,4], strt=7) ;
                                    not_found
> position3(4, [3,6,4,7,9,4], strt=4, endd=5) ;
                                    not_found
> position3(4, [3,6,4,7,9,4], rvrs=true) ;
                                        6
> position3(4, [3,6,4,7,9,4], endd=5, rvrs=true) ;
                                        3
> strt := 7 ;
                                     strt := 7
> position3(4, [3,6,4,7,9,4], strt=7) ;
                                        3
> position3(4, [3,6,4,7,9,4], 'strt'=7) ;
                                    not_found
```

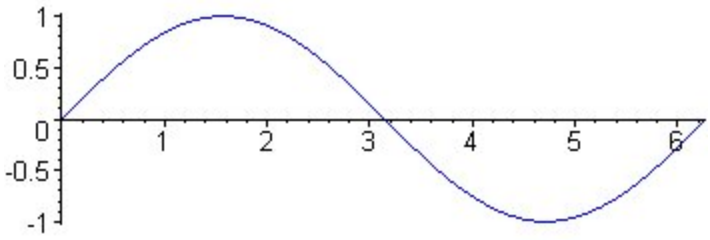
On note l'influence désastreuse d'une éventuelle valeur pour un symbole mot-clé ; si, par «mégarde», le mot-clé `strt` a une valeur, ici 7, alors l'argument à mot-clé est inopérant ; c'est une source permanente de bugs quelquefois assez pervers sous Maple, qu'on retrouve ici sous une forme un peu différente. Comparer avec :

```
> color := 56 ;
                                    color := 56
> plot(sin, 0..2*Pi, color=blue) ;
Error, (in plot) invalid arguments
```

Le point de vue de Maple est en effet un peu différent ; il considère que *tous* les arguments à partir du troisième sont des arguments à mot-clé, mais il ne reconnaît pas dans l'entier 56, valeur du symbole `color`, un mot-clé «légal» enregistré et

signale une erreur de cohérence. Maple signale qu'on peut éviter ce type d'erreur en empêchant l'évaluation du symbole `color` ; c'est exact mais reste assez lourd[3].

```
> plot(sin, 0..2*Pi, 'color'=blue) ;
```



-o-o-o-o-o-o-

[3]Les langages plus évolués commme Common Lisp ont un espace particulier de symboles-mots-clés, notion trop délicate techniquement pour le «grand public» Maple...

Chapitre 7

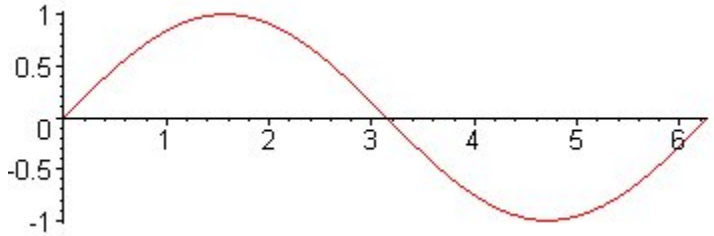
Graphiques.

7.1 Introduction.

Comme (presque) tout logiciel mathématique «grand public», Maple permet la réalisation assez commode de *graphiques mathématiques* de toutes sortes. Le mécanisme sous-jacent est astucieux et élégant et donne des moyens assez puissants aux développeurs. Le principe est le suivant ; les procédures graphiques Maple produisent des objets (pseudo-) fonctions particuliers dont l'opérateur maître est le symbole `PLOT`. Ceci fait, si jamais l'utilisateur Maple, «inconsciemment» (cas le plus fréquent) ou consciemment, demande l'impression de l'objet en question, Maple affiche le graphique probablement souhaité par l'utilisateur.

Pour mettre en évidence cette tehnique, le «truc» consiste à affecter à un symbole expérience la *valeur* d'une instruction graphique ; en première approximation, cette instruction apparaît comme telle, mais en fait il s'agit d'une procédure banale retournant un objet de type **function** où l'opérateur maître est le symbole `PLOT`. Reprenons le tracé fétiche de la fonction sinus de 0 à 2π :

```
> plot(sin, 0..2*Pi) ;
```



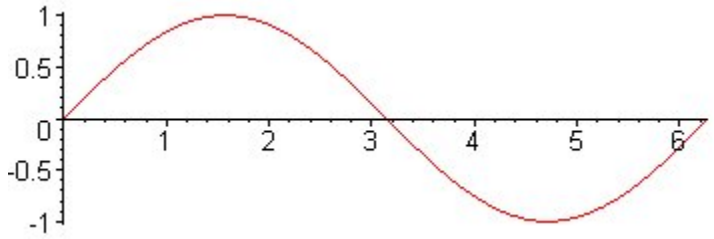
L'utilisateur a ici l'impression d'utiliser la *procédure graphique* `plot` ; mais organisons-nous d'une façon un peu différente :

```
> result := plot(sin, 0..2*Pi) ;
```

$$result := PLOT(CURVES([[0,0],[.1369555585245651,.1365278179668514],$$
$$[.2561198576993970,.2533288954318780],[.3901325751264872,.3803110317172750],$$
$$[.5250336158355156,.5012420929778901],[.6592934954461389,.6125585658925692],$$
$$\ldots\ \ldots$$
$$[6.146664884637999,-.1360967435821896],[6.283185294620000,-.1255958650482642e{-}7]],$$
$$COLOUR(RGB,1.0,0,0)),AXESLABELS(\text{“”},\text{“”}),VIEW(0..6.283185308,DEFAULT))$$

On *voit* ainsi que l'objet *retourné* par la procédure `plot` est un objet **function** dont l'opérateur est le symbole `PLOT`, le premier argument est à son tour un objet **function** dont l'opérateur est cette fois le symbole `CURVES` (on a renoncé ici à le montrer complètement), etc. On voit ainsi la structure *interne* de l'objet retourné par la procédure `plot` où, strictement parlant, aucun nouveau type de donnée est impliqué. C'est seulement la composante *print* (voir Section 2.2.2) du cycle `read-eval-print` qui reconnaît un traitement particulier : si jamais l'objet **function** à afficher en sortie a l'opérateur `PLOT`, c'est qu'il s'agit d'un objet représentant selon un codage pas si compliqué un graphique, de sorte que l'affichage graphique approprié de cet objet est donné à l'écran. Pour mettre ce dernier point en évidence, demandons l'affichage de la *valeur* du symbole `result` :

```
> result ;
```



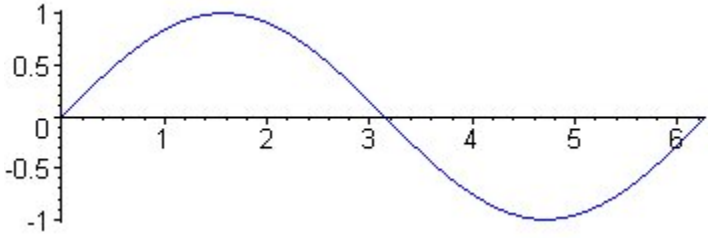
Par comparaison avec le cas de l'instruction précédente, on voit que cet affichage *graphique* n'est obtenu que si l'*unique* objet à afficher est de ce type, sinon l'affichage banal est préféré. Voir par exemple l'affichage de la *suite* à deux éléments où le second est encore la valeur du symbole `result` :

```
> 1, result ;
```

$$1, PLOT(CURVES([[0,0],[.1369555585245651,.1365278179668514],$$
$$\ldots\ \ldots$$

De la sorte, les objets graphiques peuvent être manipulés comme tout objet Maple. Comme exemple simple, transformons à titre d'exercice notre courbe rouge en courbe bleue ; il suffit pour ce faire de modifier l'opérande `COLOUR` de notre objet graphique ; ceci est obtenu par substitution, et ici on affiche immédiatement l'objet transformé :

```
> subs(COLOUR(RGB.1.0.0.0) = COLOUR(RGB.0.0.1.). result) :
```



Le mode `RGB` pour la couleur de l'objet graphique est un triplet de réels compris entre 0 et 1 donnant un indice pour chacune des trois couleurs fondamentales, rouge, vert (green) et bleu. Par défaut, un seul objet graphique est le plus souvent affiché en rouge «pur», donc indices `RGB` $= (1, 0, 0)$; on comprend ainsi comment le résultat après substitution est au contraire affiché en bleu.

Comme autre exercice, on va manipuler la liste des points de notre courbe sinusoïdale comme suit : on va échanger la première et la deuxième coordonnée, réalisant donc une symétrie par rapport à la première bissectrice des deux axes, puis on augmente la première coordonnée obtenue d'une unité. Ceci peut être obtenu par la suite de manipulations élémentaires suivantes dont on cache les résultats. D'abord on extrait la composante `CURVES`, dont on extrait ensuite la liste des points. Un coup de `map` permet de faire facilement la transformation souhaitée, optenant la liste appelée `points2`. On construit par substitution un nouvel objet `CURVES`, puis un nouvel objet `PLOT` qu'enfin on affiche :

```
> curve := op(1, result) :
> points := op(1, curve) :
> points2 := map(item -> [item[2]+1,item[1]], points) :
> curve2 := subsop(1=points2, curve) :
> result2 := subsop(1=curve2, result) :
> result2 ;
```

Bibliographie

[1] Alfred V. Aho, John E. Hopcroft, Jeffrey D. Ullman. *The design and analysis of computer algorithms*. Addison-Wesley, 1974.

[2] Patrick Dehornoy, *Complexité et décidabilité*. Springer, 1993.

[3] Andrew Hodges. *Alan Turing : the enigma of intelligence*. Burnett Books Limited, 1983.

[4] Serge Lang. *Algebra*. Addison-Wesley, 1965.

[5] Patrice Naudin et Claude Quitté. *Algorithmique Algébrique*. Masson, 1992.

[6] Lionel Schwartz. *Mathématiques pour la Liccence, Algèbre*. Dunod, 1998.

[7] Niklaus Wirth. *Algorothms + Data Structures = Programs*. Prentice Hall, 1976.