MAPLE

Introduction à l'utilisation et à la programmation
DEA Informatique Fondamentale et Applications

Sébastien Veigneau
Université de Marne-la-Vallée, Institut Gaspard Monge,
2, rue de la Butte Verte, 93166 Noisy-le-Grand Cedex.

Novembre 1996

Le programme *Maple* est développé depuis 1980 à l'Université de Waterloo au Canada. On en est aujourd'hui à la version 4 de *Maple V* mais des versions 1, 2 et 3 du système tournent encore sur de nombreuses plate-formes. Quoi qu'il en soit le développement continue, les nouvelles versions sortent régulièrement, tous les deux ans à peu près, apportant à la fois des changements dans le langage de programmation, l'interface utilisateur et les librairies disponibles. Le système est largement diffusé sur stations UNIX, Macintosh ou PC.

C'est un système de calcul formel—*computer algebra system*—principalement utilisé pour l'enseignement, la recherche scientifique et les mathématiques. Son intitulé de système de calcul formel tient au fait qu'il a la capacité d'effectuer à la fois des calculs numériques et symboliques. C'est aussi un langage de programmation qui vient compléter le moteur de calcul : le projet initial de 1980 prévoyait de produire un langage de programmation pour implémenter des algorithmes mathématiques, affichant une grande puissance et efficacité de telle sorte qu'il puisse être utilisé pour le codage de la plupart des algorithmes mathématiques sans recours à un langage de type *C* par exemple. Résultat de l'opération, presque la totalité des possibilités tant mathématiques que graphiques de *Maple* a été développée avec le langage de programmation de *Maple*. Le langage n'est pas vraiment typé. Il est interprété et la vérification des types se passe à l'exécution. Ces vérifications correspondent à une restriction d'action de tel ou tel opérateur ou fonction. Autrement dit, la vérification des types intervient pour prévenir l'utilisateur d'un calcul n'ayant pas de sens. Le langage ne nécessite pas spécialement de déclaration des objets mais la notion de type existe.

L'interface utilisateur et le moteur de calcul sont programmés en *C*. Ils représentent de l'ordre de 500 Ko de données alors que la librairie *Maple*—directement programmée en *Maple*—fournit environ 2500 fonctions qui représentent près de 150000 lignes de code. Par ailleurs, les contributions extérieures viennent grossir chaque année la librairie *Maple*—de l'anglais *Maple share library*. Ce sont par exemple des fonctions spécifiques voire des librairies complètes ou plus simplement de la documentation sur tel ou tel sujet, qui proviennent des utilisateurs du monde entier.

En résumé, *Maple* est un programme interactif de calcul symbolique, qui dispose d'un langage de programmation interprété. Il n'est pas le mieux adapté pour faire des calculs numériques intensifs à cause de la surcharge de travail amenée par son interpréteur, mais il est un bon outil pour effectuer des calculs numériques en grande précision, en comparaison avec les langages de programmation plus classiques comme *Pascal* ou *C*.

La partie suivante met en évidence, sous forme de plusieurs remarques, les points clés de l'utilisation et de la programmation de *Maple*.

1 Utiliser et programmer

- La commande `maple` permet de lancer une session *Maple*.
- Chaque commande *Maple* doit se terminer par un '`;`'.
- Pour supprimer l'affichage, on remplace le '`;`' par '`:`'.
- Le caractère `"` fait référence au dernier résultat. L'expression `""` fait référence à l'avant dernière valeur calculée et l'expression `"""` dénote l'avant avant dernier résultat.
- L'opérateur `:=` est l'opérateur *Maple* d'affectation d'une variable.
- *Maple* réalise une arithmétique exacte sur entiers.
- La fonction `evalf` fournit une approximation décimale.
- La fonction `evalhf` fournit une approximation décimale, en utilisant les circuits électroniques spécialement dédiés au calcul flottant.
- Pour revenir à la normale suite à une erreur de syntaxe, on tape un point-virgule validé par un <Retour Chariot>.
- Les commandes `sum` et `product` permettent de manipuler respectivement des sommes et des produits.
- Les anti-quotes ` délimitent une chaîne de caractères.
- La commande `?` doit être un premier réflexe en cas de problème.
- La commande `with` permet de charger un package de fonctions.
- La commande `quit` termine une session *Maple*.
- Il est possible qu'une variable `x` redevienne un symbole par la commande `x:='x';`.
- Programmer *Maple* pour manipuler des formules, c'est basiquement programmer des manipulations d'arbres syntaxiques.
- La fonction `type` : `type(f, t)` retourne `true` si l'expression `f` est du type `t`.
- La fonction `whattype` retourne le type d'une expression.
- Le type `rational` se rapporte aux nombres rationnels, *i.e.* aux fractions et aux entiers. Le type `float` fait référence aux réels c'est-à-dire aux nombres à virgule, alors que le type `numeric` rassemble l'ensemble de ces deux types de nombres.
- La fonction `nops` retourne le nombre d'opérandes d'une expression.
- La fonction `op` permet d'extraire une des opérandes d'une expression.
- Le type `name` rassemble les types `string` et `indexed`.
- Si `f` est de type `indexed`, la fonction `nops` appliquée sur `f` retourne le nombre d'indices et la fonction `op` permet d'extraire un certain indice. Par ailleurs, `op(0, f)` retourne le nom qui est indicé.
- La syntaxe d'un appel de fonction est $f(x_1, x_2, \ldots)$ où f est le nom de la fonction et $x_1, x_2, \ldots$ sont les arguments.
- Si `f` est une fonction, la fonction `nops` appliquée sur `f` retourne le nombre d'arguments et la fonction `op` permet d'extraire un certain argument. Par ailleurs, `op(0, f)` retourne le nom de la fonction.
- Une instruction conditionnelle s'écrit :

  `if` *condition* `then`
      suite_d'instructions
  [`elif` *condition* `then` *suite_d'instructions*]*
  [`else` *suite_d'instructions*]
  `fi`

- Une instruction de boucle s'écrit : [`for` *nom*] [`from` *expression*] [`by` *expression*] [`to` *expression*] [`while` *condition*]
  `do`
      suite_d'instructions
  `od`

  où alors :

```
[for nom] [in expression] [while condition]
do
        suite_d'instructions
od
```

- Un `break` permet de sortir de la boucle la plus intérieure dans laquelle nous nous trouvons.
- Un `next` permet de passer à l'itération suivante dans la boucle la plus intérieure où nous nous trouvons.
- Une *séquence* est une suite d'expressions séparées par des virgules.
- Le symbole spécial `NULL` dénote la séquence vide.
- L'instruction `seq(` *f(*`i`*)*, `i=`*m*`..`*n* `)` permet de créer une séquence. Une autre syntaxe est donnée par `seq(` *f(*`i`*)*, `i=`*a* `)`.
- Pour accéder à la `ieme` entrée d'une séquence `s`, on utilise l'expression `s[i]`.
- Une *liste* est une structure de donnée permettant de regrouper des objets ensemble. Les crochets servent à créer une telle structure.
- La liste vide est notée `[]`.
- Il est impossible de modifier les éléments d'une liste.
- Un *ensemble* est une structure de donnée permettant de regrouper des objets ensembles, sans duplication. Les accolades servent à créer une telle structure.
- L'ensemble vide est noté {}.
- Il est impossible de modifier les éléments d'un ensemble.
- Une *table* est faite de deux parties : une fonction d'indexation et une collection d'objets.
- Attention, seuls les objets de type `table` peuvent être altérés après avoir été créés.
- La fonction `copy` peut être utilisée pour créer une copie d'une table de telle sorte que des opérations sur cette copie n'affectent pas l'originale.
- Les *tableaux* sont des tables ayant des indices entiers pris dans un intervalle déterminé à la création du tableau et de dimension fixée à l'avance aussi.
- La fonction `array` permet de créer un tableau.
- Les tableaux sont encore plus efficaces d'accès que les tables.
- Une fonction d'indexation permet de spécifier quelles doivent être les valeurs par défaut des éléments d'une table ou d'un tableau.
- L'instruction `read` lit un fichier texte ordinaire de commandes *Maple*.
- L'instruction `read`, appelée avec un nom de fichier ayant l'extension `.m`, lit un fichier, au format interne *Maple*, de commandes *Maple*.
- L'instruction `save` permet de sauver tous les résultats déjà calculés dans un fichier texte ordinaire.
- L'instruction `save`, appelée avec un nom de fichier ayant l'extension `.m` permet de sauver une session *Maple* dans un fichier au format interne *Maple*.
- La commande `writeto` écrit les sorties *Maple* dans un fichier, dont le nom est passé en argument.
- La commande `appendto` écrit les sorties *Maple* dans un fichier, dont le nom est passé en argument, en les rajoutant en fin de fichier si celui-ci existait déjà.
- Les commandes `writeto(terminal);` et `appendto(terminal);` ont toutes deux comme effet de rediriger la sortie *Maple* sur l'écran.
- Une procédure se définit par :

```
proc( séquence_paramètres )
        [ global séquence_noms; ]
        [ local séquence_noms; ]
        [ options séquence_options; ]
        instructions
end
```

- La commande `RETURN` réalise une sortie directe de la fonction. La valeur retournée par la fonction est alors, la séquence d'expressions—éventuellement vide—passée en argument du `RETURN`.
- La commande `ERROR` peut être utilisée pour générer un message d'erreur au sein d'une procédure.
- Un appel de fonction avec des paramètres se fait en passant à la fonction la séquence de ces paramètres.
- Si l'évaluation d'une expression passe par une définition de fonction, le résultat est le *dernier nom évalué*.
- La procédure `map` prend en argument une procédure `f`, une liste `[a1, a2, ..., ak]`. Elle retourne alors la liste `[f(a1), f(a2), ..., f(ak)]`.
- La commande `unapply` permet de créer une procédure, à partir d'une expression.
- La valeur de la variable `nargs` est le nombre de paramètres correspondant à l'appel courant.
- La valeur de la variable `procname` est le nom de la procédure courante.
- D'une manière générale, il nous est impossible d'assigner une valeur à un argument à l'intérieur du corps d'une procédure.
- Si l'argument d'appel est entre quotes, ou n'a pas d'évaluation (*i.e.* est un symbole), nous pouvons lui affecter une valeur à l'intérieur du corps d'une procédure.
- Au cours d'un appel de fonction, les variables locales ne sont évaluées qu'à un seul niveau. La commande `eval` permet une évaluation complète d'une variable locale.
- Depuis la version 3 de *Maple V*, toute variable à laquelle on assigne une valeur dans le corps d'une procédure, et qui n'est pas déclarée, est considérée comme locale.
- Cette même remarque est aussi valable pour les variables de boucles et de séquences.
- Pour utiliser une variable globale dans le corps d'une procédure, il faut la déclarer dans la rubrique (optionnelle) `global` au début de la procédure.
- *Maple* supporte pas les imbrications de procédures.
- Les paramètres et variables locales imbriqués n'utilisent pas de règles de portée imbriquées.
- La rubrique `option` dans la définition d'une nouvelle procédure, permet de positionner des propriétés de cette procédure.
- Les options disponibles (pour une version de *Maple V* antérieure à la version 3) sont : `builtin`, `remember`, `system`, `operator`, et `trace`.
- Les options supplémentaires disponibles depuis la version 3 de *Maple V* sont : `arrow`, `angle`, `package` et `Copyright`.
- Lorsque l'option `remember` est positionnée, à la fin de chaque appel de la procédure, une entrée est ajoutée dans une table associée à la procédure. Cette entrée a pour objet de se souvenir du résultat calculé, en fonction des arguments spécifiés.
- Si `f` est une procédure avec l'option `remember` positionnée, alors l'instruction `f := subsop(4=NULL, op(f));` efface cette table.
- Le niveau 0 correspond au niveau de la ligne de commande.
- Les instructions à l'intérieur de structures conditionnelles ou de répétitions correspondent au niveau 1, voire 2 s'il y a imbrication double, etc.
- Le premier niveau de procédure correspond au niveau 5.
- Le positionnement de la variable `printlevel` à la valeur `n`, implique l'affichage de toutes les instructions exécutées à un niveau inférieur ou égal à `n`.
- La valeur par défaut de la variable `printlevel` est 1, alors que le niveau de la ligne de commande est 0.

2 Exercices de découverte

Voici quelques exercices qui devraient vous aider à comprendre les différents points de la section précédente. Ces exercices doivent être compris avant d'envisager de passer à la suite.

▸ **Exercice 1.** *Comment commencer une session Maple ? Quel est le prompt proposé par le logiciel ?*

▸ **Exercice 2.** *Entrez l'instruction* `4/6;` *suivie d'un* <RETOUR CHARIOT>. *Quel affichage obtenez-vous ? Essayez la commande* `";`, *puis une instruction plus compliquée comme* `2*" + 1;`. *Qu'affichent les lignes suivantes ?*

```
> " + "":
> ";
                              ...
> r := " * "";
                             ......
```

▸ **Exercice 3.** *Il existe en Maple un certain nombre de constantes mathématiques prédéfinies. Quelles sont-elles ? Vérifiez votre réponse à l'aide de la fonction* `whattype` *qui retourne le type de son argument.*

▸ **Exercice 4.** *Maple fait la différence entre les minuscules et les majuscules donc nous devons être bien sûr d'utiliser les bonnes lettres par exemple pour référencer ces constantes. Vérifiez que les noms* `pi` *et* `Pi` *ne sont pas équivalents à l'aide par exemple de la commande* `evalf` *qui fournir une approximation décimale de son argument.*

▸ **Exercice 5.** *Maple réalise une arithmétique exacte sur entiers. En fait, Maple réalise des calculs sur des entiers plutôt que sur des nombres décimaux. L'utilisation des décimaux est tout à fait possible mais correspond à un ordre explicite de l'utilisateur. Regardez le résultat des commandes suivantes :*

```
> 2*3+2/7;
                             .....
> 2.0*3 + 2/7;
                          ...........
> sin(Pi/3);
                             .....
> evalf(");
                          ...........
```

▸ **Exercice 6.** *Il n'y a pas de limite sur la longueur des entiers en Maple. Il est ainsi naturel de calculer avec des entiers de plusieurs centaines de chiffres. Calculez* 2^{100}. *D'un autre côté, il est possible de faire de l'arithmétique décimale avec plus que les 10 chiffres, par défaut. C'est la variable* `Digits` *qui stocke le nombre de chiffres utilisés pour les calculs décimaux. Fixez la variable* `Digits` *à 50 et lancez* `evalf(sin(Pi/3));`.

▸ **Exercice 7.** *En fait, la commande* `evalf` *utilise l'arithmétique entière de la machine pour effectuer ses calculs, ce qui apporte beaucoup de souplesse pour par exemple changer de précision via la variable* `Digits`. *Pourtant, c'est au détriment de l'efficacité par rapport à une utilisation des circuits électroniques spécialisés de la machine. Oserons-nous marquer noir sur blanc que la documentation Maple précise sans complexe aucun, que la fonction* `evalf` *peut être de 50 à 500 fois plus lente que son équivalente en Fortran, à précision égale bien entendu. Maple propose donc la fonction* `evalhf`, *similaire à* `evalf`, *mais utilisant les possibilités machine en matière de calculs décimaux. La fonction* `evalhf` *traite la variable* `Digits` *d'une manière spéciale. Elle n'utilise pas sa valeur pour ses calculs mais son évaluation correspond au nombre de chiffres utilisés. Cette valeur est fixe— car elle dépend de la machine. Quelle est cette valeur pour votre machine ?*

▸ **Exercice 8.** *Si nous faisons une erreur de syntaxe lors de la saisie de notre commande, Maple répondra par un message indiquant une "*`syntax error`*". Créez une erreur de syntaxe. Comment se sortir d'une telle situation ?*

▸ **Exercice 9.** *Les sorties de Maple ne sont pas toujours dans leur forme la plus simplifiée. Il existe un certain nombre de commandes telles que* `collect`, `combine`, `expand`, `factor`, *ou* `simplify` *qui sont à disposition pour simplifier ou arranger un résultat suivant une forme voulue. Développez avec Maple le polynôme* $(x+y)(x-y)-x^2$. *Factorizez le polynôme* $x^4+x^2y+2x^2+2x^3+2xy+2x+y+1$. *Simplifiez l'expression Maple* $exp(a+ln(b*exp(c)))$ *et l'expression* $(x^3-y^3)/(x^2+xy+y^2)$.

▸ **Exercice 10.** *Dérivez par rapport à* x *les expressions* $\ln(x)$, $\arcsin(ax)$, $x\ln(x)$. *Calculez les intégrales* $\int\sqrt{(1-x^2)}\,dx$, $\int_1^2 \ln x\,dx$.

▸ **Exercice 11.** *Calculez formellement les sommes des* `n` *premiers carrés, cubes, etc. Quelle est la différence entre les fonctions* `sum` *et* `Sum` *?*

▸ **Exercice 12.** *Analyser la réponse de Maple aux instructions suivantes :*

```
> f:=(x^3+2*x^2-x)/(x^3-x^2+x-1):
> int(f, x);
                    ....................
> diff(", x);
                    ....................
> "-f;
                    ....................
```

Comment obtenir le résultat espéré ?

▸ **Exercice 13.** *Une chaîne de caractères consiste en un certain nombre de caractères entourés par des anti-quotes* ‘. *Que produisent les commandes suivantes ?*

```
> ‘Voila une chaine de caracteres‘;
        ...........................
> ‘123abc‘;
                    ........
> ‘prog.txt‘;
                    ........
```

Certains caractères spéciaux tels que le `+`, *le* `.` *ou encore le* `/` *peuvent être présent dans une chaîne de caractères dès lors que celle-ci est entourée d'anti-quotes. Si ces anti-quotes sont absentes, ces caractères spéciaux seront tout naturellement traités comme des opérateurs Maple normaux. Essayez les instructions :*

```
> 3+abc+4;
                    .......
> ‘3+abc+4‘;
                    .......
> repertoire/fichier;
                    .........
> prog.txt;
                    .........
```

Expliquez les deux derniers résultats.

▸ **Exercice 14.** *Si nous avons besoin d'information sur une fonction Maple, nous pouvons utiliser la commande Maple* `?`. *Cette commande nous donne accès à une foultitude d'informations sur le système Maple en général et sur les fonctions Maple en particulier. En fait, l'ensemble du livre[4] est disponible comme aide en ligne Maple. Entrez la commande* `?` *et lisez cette première page de documentation.*

▸ **Exercice 15.** *Les librairies de fonctions viennent augmenter le noyau de Maple. On parle habituellement de package pour référencer un tel ensemble de fonctions spécifiques. Il y en a beaucoup et maintenant que vous savez demander de l'aide à Maple, commençons par la commande* `?packages`.

▸ **Exercice 16.** *Un des plus classiques mais aussi des plus imposants de part le nombre de ses fonctions, est le package d'algèbre linéaire,* `linalg`. *Chargez ce package et calculez le déterminant de la matrice*

$$\begin{bmatrix} x-1 & 2 & 3 \\ 0 & x-2 & 2 \\ 2 & 1 & x-3 \end{bmatrix}$$

▸ **Exercice 17.** *Il y a plusieurs façons de quitter le logiciel. Quelles sont-elles ?*

3 Exercices de révision

Cette section présente quelques exercices de révision pour voir si les notions présentées ont été assimilées.

▸ **Exercice 18.** *Nous avons vu que pour commencer une session Maple, nous utilisions la commande* UNIX `maple` *par exemple. Cette commande accepte quelques options que nous nous proposons de regarder. Quelle est donc la commande Maple qui devrait nous permettre d'obtenir de telles informations et quelles sont ainsi les options acceptées par la commande* UNIX `maple` *par exemple ?*

▸ **Exercice 19.** *Une des premières remarques que nous avons faite au sujet du langage Maple, concernait l'utilisation du point-virgule (voire du deux-points). Nous avons vu que toute commande Maple doit se terminer par un de ces deux caractères. Il existe pourtant une exception à cette règle que nous ne nous sommes déjà pas privés d'utiliser.*

1. *Quelle est donc cette exception à la règle ?*
2. *Quelle commande Maple pouvons-nous lancer pour confirmer notre idée ?*

▸ **Exercice 20.** *Dans l'exercice 19, nous avons vu une exception à la règle de l'utilisation du point-virgule. En fait, il existe une autre exception qui concerne une commande Maple que l'on ne peut utiliser, par nature, qu'une seule fois par session. Quelle est donc cette commande ?*

▸ **Exercice 21.** *Le logiciel Maple met à disposition une librairie de fonctions concernant l'algèbre linéaire : la libraire (ou package)* `linalg`.

1. *Examiner la documentation de la librairie* `linalg`.
2. *Comment charge-t-on cette librairie dans Maple ?*
3. *Comment obtenir de l'aide sur une fonction d'un package ?*
4. *Essayer les exemples donnés dans l'aide concernant la fonction* `det` *(calcul du déterminant d'une matrice) du package* `linalg`.

▸ **Exercice 22.** *Nous avons vu qu'en Maple, le signe d'affectation est* `:=`. *A la différence des langages de programmation classiques, il est possible d'affecter à une variable une expression contenant d'autres variables ne possédant aucune valeur (variables libres ou formelles). Les polynômes sont un cas particulier de telles expressions. Un polynôme en une indéterminée x est considéré comme une expression avec une variable libre x. Il existe de nombreuses fonctions très utiles pour manipuler par exemple les polynômes en une indéterminée. La fonction* `factor` *permet de factoriser un polynôme, alors que les commandes* `simplify` *et* `expand` *permettent respectivement de simplifier et développer une expression.*

Note *: Ces commandes sont très utilisées lors d'une session Maple et nous conseillons de tout de suite consulter leurs aides en ligne respectives.*

- *Entrez le polynôme $p(x) = x^2 - x - 1$. Affecter le à une variable* `p` *sans l'afficher.*
- *Calculez $p(x)^5$, développer puis refactoriser le résultat.*
- *Substituez y à x dans* `p` *en utilisant la commande* `subs`.
- *Trouvez les solutions de l'équation $p(y) = 0$ à l'aide de la commande* `solve`.
- *Vérifiez que les deux solutions calculées sont effectivement les racines du polynôme.*

▸ **Exercice 23.** *Refaites les questions de l'exercice précédent pour le polynôme $p(x) = x^2 + ax + b$.*

▸ **Exercice 24.** *En Maple, il est possible d'utiliser des variables muettes telles les indices de sommation ou les variables d'intégration par exemple. Calculez la somme*

$$s_1 := \sum_{i=1}^{n} i^3 .$$

▸ **Exercice 25.** *Calculer la somme*

$$s_2 := \sum_{j=0}^{n} j^2 \binom{n}{j} ,$$

où $\binom{n}{j}$ *est le coefficient binomial. Ce coefficient est accessible en Maple par la commande* `binomial(n,j)`.

▸ **Exercice 26.** *Nous avons déjà évoqué l'utilisation de la fonction* `int`. *En fait, il existe une deuxième fonction qui est la sœur jumelle de la fonction* `int` *: la fonction* `Int`. *Essayer sur un exemple, sachant que les deux commandes obéissent à la même syntaxe, de comprendre la différence entre ces deux fonctions.*

▸ **Exercice 27.** *Nous nous proposons de faire quelques calculs avec la fonction* `int`.

1. *Calculez une primitive de la fonction* $\sin^2(x)$.
2. *Vérifiez que le résultat est correct en dérivant puis en simplifiant.*

▸ **Exercice 28.** *Faites de même avec les fonctions* $\sin(1/x)$, $1/p^{(1)}$ *et* $1/(x^2+1)^k$.

▸ **Exercice 29.** *Calculez la séquence des coefficients binomiaux* $\binom{10}{i}$ *pour* $i = 0, \ldots, 10$. *On utilisera la commande* `seq`.

▸ **Exercice 30.** *Les polynômes de Bernoulli* $B_n(x)$ *sont définis par leur série génératrice :*

$$\frac{te^{xt}}{e^t - 1} = \sum_{n=0}^{\infty} B_n(x) \frac{t^n}{n!} ,$$

où $x \in \mathbb{R}, t \in \mathbb{C}, |t| < 2\pi$. *Les nombres de Bernoulli sont par définition les nombres* $B_n = B_n(0)$ *pour* $n \geq 0$. *Ces nombres sont rationnels. Nous nous proposons de calculer les premiers nombres de Bernoulli en admettant que* $B_0 = 1$ *et* $B_1 = -1/2$. *Ces nombres peuvent être calculés par récurrence en utilisant la formule*

$$B_n = \sum_{k=0}^{n} \binom{n}{k} B_k, \quad (n \geq 2) ,$$

sous la forme

$$B_{n-1} = -\frac{1}{n} \sum_{k=0}^{n-2} \binom{n}{k} B_k, \quad (n \geq 3) .$$

Nous donnons le début du calcul pour B_0 *et* B_1 *:*

```
> B.0 := 1;

                    B0 := 1

> B1 := -1/2;

                    B1 := -1/2
```

Quelle propriété évidente sur les nombres de Bernoulli pouvons-nous déduire de ces premiers calculs ?

▸ **Exercice 31.** *Calculez une approximation de l'irrationnel* π *avec 2000 décimales. Pour la petite histoire,* π *est calculée depuis la version V de Maple par la formule de Chudnovsky suivante :*

$$\frac{1}{\pi} = \frac{6541681608}{640320^{3/2}} \sum_{k=0}^{\infty} \Big(\frac{13591409}{545140134} + k \Big) \frac{(6k)!}{(3k)!(k!)^3} \frac{(-1)^k}{640320^{3k}} .$$

(1) Où p est le polynôme introduit dans l'exercice 22.

4 Structures de données et structures de contrôle

Avant de commencer, nous devons mettre en évidence une différence essentielle entre *Maple* et les langages traditionnels de programmation. Si un identificateur n'a pas été assigné d'une valeur, sa valeur est formellement lui-même. C'est ce que nous appelons un *symbole*. Les symboles sont utilisés pour représenter des inconnues dans des équations, des variables (indéterminées) dans des polynômes ou encore des indices de sommation par exemple.

▸ **Exercice 32.** *Entrez les instructions suivantes et analysez les résultats :*

```
> p := x^2 + 4*x + 4;
                    ....................
> p;
                    ....................
> x;
                                ...
> x := 3:
> p;
                                ...
```

Suite aux intructions précédentes, calculez l'intégrale $\int p\,dx$ et conclure. Comment faire en sorte que `x` *retrouve sa valeur formelle ?*

▸ **Exercice 33.** *Les formules en Maple sont représentées par des arbres syntaxiques ou, pour être tout à fait précis, par des DAGs—Directed Acyclic Graphs. Comment sont codées les opérations de soustraction et de division ? Quels sont les différents types arithmétiques en Maple ? Notez la différence cruciale de résultat entre les deux commandes* `2/3;` *et* `2/3.0;`.

▸ **Exercice 34.** *On considère le polynôme $p := x^2y + 3x^3z + 2$. Comme nous le constatons,* `p` *est une somme de termes. Combien y en a-t-il ? Essayer d'extraire chacun des termes à l'aide de la fonction* `op`*. Analysez complètement l'expression* `p`.

▸ **Exercice 35.** *Les noms en Maple ont plusieurs formes. Pour les découvrir, entrez les instructions* `type(a, string);`, `type(a, name);`, `whattype(A[1]);`, `type(A[1], indexed);`, `type(A[1], name);`.

▸ **Exercice 36.** *Les fonctions se comportent quelque peu comme les variables indicées. Commentez les résultats des commandes suivantes :*

```
> nops( f(x,y,z) );
                         .....
> op(1..3, f(x,y,z));
                         .....
> op(0, f(x,y,z));
                          ...
```

▸ **Exercice 37.** *Nous sommes désormais capables d'analyser la structure de n'importe quelle formule Maple car nous connaissons l'action de chacune des fonctions* `nops`, `op`, *etc. sur les différents objets que nous pouvons rencontrer dans une formule. Quel est le type du premier terme de la formule :* $\sin(x[1])^2(1-cos(\pi x[2]))$.

▸ **Exercice 38.** *Nous nous attardons sur un problème d'évaluation qui peut arriver si nous ne faisons pas assez attention aux commandes entrées. Nous considérons le polynôme* `p` *que nous avions défini précédemment. Ici, nous avons assigné au nom* `p` *une formule qui contenait explicitement le symbole* `x`*. Que se passe-t-il si nous utilisons désormais le même nom* `p` *des deux côtés de l'assignation, par exemple pour l'instruction* `p := p + x;` *? Qu'en est-t-il si vous entrez la commande* `q:=q+x;` *et demandez ensuite la valeur de* `q`.

▸ **Exercice 39.** *L'utilisation de la structure conditionnelle* `if` *est parfois la cause de problème, lorsque l'évaluation de la condition n'est pas réalisable. Entrez les commandes suivantes et concluez sur le problème.*

```
> if a>b then print(a) fi;
```

. . . .

```
> if evalb(a>b)=true then print(a) fi;
> if evalb(a>b)=true then print(a)
  else print(evalb(a>b)); 1; fi;
```

. . . .
. . .

▸ **Exercice 40.** *Affichez à l'aide d'une boucle, les 4 premiers carrés.*

▸ **Exercice 41.** *Recherchez avec une boucle le plus petit entier premier supérieur à* $10^{10}+1$*. Proposez une version avec une boucle* `while` *simple et une version plus évoluée avec une boucle* `for-while`*.*

▸ **Exercice 42.** *La notion de séquence est tout à fait essentielle. Elle apparaît réellement partout et c'est tout à notre avantage. Par exemple, nous avons déjà vu que les arguments d'une fonction étaient séparés par des virgules, et bien c'est parce qu'en fait ils sont simplement une séquence—au sens Maple—d'expressions. Entrez l'instruction* `s := 1, 4, 9, 16, 25;` *et demandez le type de* `s`*. Que conclure du résultat des instructions* `u := 1, (4, 9, 16), 25;` *et* `u, u;`*. Calculez le maximum de la séquence* `u`*.*

▸ **Exercice 43.** *Pourquoi ne peut-t-on pas appliquer les fonctions* `op` *et* `nops` *à une séquence.*

▸ **Exercice 44.** *Créez à l'aide d'une boucle la séquence des 10 premiers carrés. Faites de même avec la fonction* `seq` *puis avec l'opérateur* `$`*. Que pensez-vous des temps de création de la séquence lorsque l'on considère des très grandes séquences ? Que conclure ?*

▸ **Exercice 45.** *On considère l'expression* $a := 3x^3+yx-11$*. Créez la séquence des coefficients des puissances de* x *de cette expression.*

▸ **Exercice 46.** *Calculez la dérivée partielle* 22^{eme} *par rapport à* x*, de la dérivée partielle* 23^{eme} *par rapport à* y *de* $sin(xy)$*.*

▸ **Exercice 47.** *Pour bien faire mettre en évidence une différence entre l'opérateur* `$` *et la fonction* `seq`*, regardez les résultats des instructions suivantes :*

```
> a.i $ i=1..10;
```

. . . .

```
> 'a.i' $ i=1..10;
```

. . . .

```
> seq(a.i, i=1..10);
```

. . . .

▸ **Exercice 48.** *Complétez l'éxécution suivante :*

```
> ll := [x, 1, 1-z, x];
```

. . . .

```
> whattype(ll);
```

. . . . . .

```
> ll[3];
```

. . . . . .

```
> op(3, ll);
```

. . . . . .

```
> ll[];
```

. . . .

```
> ll[1..nops(ll)];
```

. . . .

```
> whattype(");
```

. . . .

```
> [""];
```

. . . .

▸ **Exercice 49.** *Que produisent les instructions suivantes ?*

```
> i := 4:
> ll := [ ll[1..i-1],
         ll[i+1..nops(ll)] ];
```

....

```
> ll := subsop(i=NULL, ll);
```

....

▸ **Exercice 50.** *Complétez l'exécution suivante :*

```
> s := {x, 1, 1-z, x};
```

....

```
> whattype(s);
```

...

Trouvez les opérations possibles sur les ensembles.

▸ **Exercice 51.** *Partant d'une séquence quelconque, créez une liste puis transformez-la en un ensemble (pour par exemple éliminer les répétitions d'éléments) et enfin reconstruisez une liste.*

▸ **Exercice 52.** *L'enjeu de cet exercice est la découverte des objets de type* `table`*. Complétez l'exécution suivante et commentez la nature des résultats :*

```
> couleur[rouge] := red, rot;
```

....

```
> couleur[bleu] := blue, blau;
```

....

```
> couleur[jaune] := yellow, gelb;
```

....

```
> table();
```

....

```
> table([22,42]);
```

....

```
> F := table([sin=cos,cos=-sin]):
> op(op(F));
```

....

```
> F[cos](Pi/2);
```

....

```
> indices(couleur);
```

....

```
> entries(couleur);
```

....

```
> "[2][1];
```

....

```
> couleur;
```

....

```
> eval(couleur);
```

....

```
> whattype(couleur);
```

....

```
> whattype(eval(couleur));
```

....

```
> T[x] := a;
```

....

```
> eval(T);
```

....

▸ **Exercice 53.** *Un tableau est un type particulier de table. Quel est donc la définition exacte d'un tableau en Maple et à quel type Maple ces objets correspondent-ils ? Ordonnez en utilisant un tri à bulle un tableau quelconque.*

5 Exercices de révision

▸ **Exercice 54.** *Les expressions sous Maple sont complètement évaluées. Pour bien s'en rendre compte, complétez l'exécution suivante :*

```
> c := b: b := a: a := 1: b;
                              ...
> c;
                              ...
> b := 'b': c;
                              ...
> b;
                              ...
> a;
                              ...
```

▸ **Exercice 55.** *Compléter l'exécution suivante :*

```
> c := b: b := a: a := 1: b := `b`: c;
                              ...
> b;
                              ...
> a;
                              ...
```

▸ **Exercice 56.** *Il est possible de savoir, non pas la valeur d'une variable, mais la valeur qui lui a été assignée. Par exemple, après avoir entré les instructions* `b:=1: a:=b;` *nous aimerions savoir que* `a` *a reçu la valeur* `b` *et non* `1` *comme indiqué par le résultat de ces instructions. Comment devons-nous faire ?*

▸ **Exercice 57.** *Il existe une fonction en Maple qui retourne la liste de toutes les variables assignées (d'une valeur). Quelle est cette fonction ?*

▸ **Exercice 58.** *Voici quelques questions pour voir si nous avons les bons réflexes en ce qui concerne les expressions Maple.*

1. *Soit le polynôme $P = (x - y + 5)^5$. Quel est le troisième terme de l'expression développée de P ?*
2. *Combien P a-t-il de termes ?*
3. *Quels sont les types de P,* `factor(`P`)`, *$(x+1)(x+2)$, x^2+3x+2 et $(x+1)(x+1)$?*

▸ **Exercice 59.**

1. *Construisez la séquence des carrés des entiers de 10 à 15 ;*
2. *Construisez la séquence des plus grands communs diviseurs de 12 et des entiers de 1 à 30 ;*
3. *Construisez la liste et l'ensemble de ces plus grands communs diviseurs.*

▸ **Exercice 60.** *Complétez et expliquez l'exemple suivant :*

```
> x.i$i=1..5;
                 .................
> i:=3: x.i$i=1..5;
                 .................
> seq(x.i, i=1..5);
                 .................
```

▸ **Exercice 61.**

1. *Construisez la liste $l = [x_1, x_2, \ldots, x_{30}]$;*
2. *A partir de la liste l, construisez la liste $[x_4, x_5, \ldots, x_{30}]$; A partir de cette dernière liste, construisez la liste $[x_{30}, x_{29}, \ldots, x_4]$;*
3. *Construisez de toute pièce cette même liste.*

▸ **Exercice 62.** *En utilisant la fonction* `time`*, étudiez la différence de temps entre la création de la séquence $1, \ldots, 2^{10}$ par une boucle et par un appel à la fonction* `seq`(2)*.*

▸ **Exercice 63.** *Refaire l'exercice 62 pour cette fois comparer les performances d'une boucle et de l'opérateur* `$`*, dans le cadre de la création d'une séquence.*

(2) On relancera *Maple* à chaque fois, afin de se retrouver dans les mêmes conditions d'exécution.

6 Programmation Maple

▸ **Exercice 64.** *Trouvez et commentez les résultats des instructions suivantes :*

```
> proc(x, y) x^2 + y^2 end;
      ..............................
> sommecarres := ";
      ..............................
> sommecarres(3, 4);
                  ........
> sommecarres(3, b);
                  ........
```

▸ **Exercice 65.** *Donnez une procédure* `MEMBER` *prenant en argument un élément* `x` *et une liste* `L`*, testant l'appartenance de* `x` *à* `L`*.*

▸ **Exercice 66.** *Refaites l'exercice précédent en améliorant votre fonction* `MEMBER` *afin qu'elle détecte d'éventuelles erreurs de types d'arguments.*

▸ **Exercice 67.** *Définissez un alias pour votre fonction* `MEMBER`*.*

▸ **Exercice 68.** *Donnez une fonction prenant en argument une liste, et retournant la liste de tous les carrés des éléments de la liste d'entrée. La fonction utilisera une boucle* `for` *dans cette première version. Que pensez-vous de la complexité en temps de votre procédure ?*

▸ **Exercice 69.** *Refaites l'exercice précédent afin de donner une version s'exécutant en un temps linéaire en la taille de la liste d'entrée.*

▸ **Exercice 70.** *Comment utiliser la fonction* `map` *pour* `maper` *une fonction à plusieurs arguments ?*

▸ **Exercice 71.** *Pour comprendre l'usage de l'opérateur flèche (*`->`*), complétez la session suivante :*

```
> (x -> x+1)(3);
                     ....
> f := (x,y) -> x^2 + y^2;
            ...............
> f(2, 3);
                 ....
```

▸ **Exercice 72.** *Donnez une fonction* `maximum` *calculant le maximum de ces arguments. La fonctions devra accepter un nombre variable d'arguments.*

▸ **Exercice 73.** *Modifiez la fonction précédente afin que l'on ait accès à l'extérieur de la fonction à la position (dans la séquence d'arguments) du maximum.*

▸ **Exercice 74.** *Que conclure sur le résultat de l'exécution des instructions suivantes :*

```
> f := proc(e, n) local i;
>   n:=nops(e); sum(op(i,e), i=1..n);
> end:
> f(x+y, 's');
```

▸ **Exercice 75.** *Comment modifier la fonction* `f` *afin qu'elle réalise le travail escompté ?*

▸ **Exercice 76.** *Donnez une version récursive efficace d'une fonction calculant le n^{eme} nombre de Fibonacci. Dans ce cas précis, le mot efficace signifie s'exécutant en temps linéaire par rapport à n.*

▸ **Exercice 77.** *Il est possible de donner un ordre explicite de sauvegarde du résultat d'un appel de fonction. Les lignes suivantes illustrent cette situation. Complétez l'exécution suivante :*

```
> F := proc(n) F(n) := F(n-1) + F(n-2)
> end:
> F(0) := 0: F(1) := 1: op(4, op(F));
          ..........................
> F(4): op(4, op(F));
          ..........................
```

▸ **Exercice 78.** *Donnez une fonction* `PGCD` *calculant le plus grand commun diviseur de deux entiers* `a` *et* `b` *passés en argument. Positionnez habilement la variable* `printlevel` *pour suivre le déroulement du calcul* `PGCD(21, 15)`*.*

7 Exercices de programmation

▸ **Exercice 79.** *Ecrivez une procédure* `NextPrime` *prenant en argument un entier* `N` *et retournant le plus petit nombre premier supérieur strict à* `N`*. Prévoyez un contrôle du type de l'argument.*

▸ **Exercice 80.** *Comparez les sources de la fonction* `NextPrime` *de l'exercice 79 avec ceux de la fonction Maple* `nextprime`*. Les sources de cette dernière sont accessibles par les instructions* `interface(verboseproc=2): print(nextprime);`.

▸ **Exercice 81.** *Les nombres de Fermat sont les nombres de la forme* $2^{a(n)}+1$ *avec* $a(n) = 2^n$*. Pierre de Fermat les croyait à tord, premiers pour tout* n*. Mais quel est donc le plus petit* $n \geq 0$ *, tel que* $2^{a(n)} + 1$ *ne soit pas premier ? On fera tout d'abord une procédure* `Fermat` *prenant en argument un entier* `n` *et retournant le nombre de Fermat correspondant. Puis, en utilisant cette procédure, on écrira la procédure de calcul de ce plus petit entier recherché.*

▸ **Exercice 82.** *Les nombres de Mersenne sont les nombres entiers de la forme* $M_p = 2^p - 1$*. Si* M_p *est premier,* p *est alors nécessairement premier. On ne connaît à ce jour que 33 nombres de Mersenne premiers dont le plus grand (pour l'instant) a été découvert en 1994 et correspond à* $p = 859433$*.*

1. *Mais en fait, jusqu'à quel* p*, pouvez-vous calculer en un temps "raisonnable" le nombre* M_p *sur votre machine ? Etes-vous loin de* $p = 859433$ *?*

2. *On ne sait pas s'il existe ou non une infinité de nombres de Mersenne premiers. Dans ce cadre, regardez pour le* p *évalué à la question précédente le temps d'exécution de la commande* `isprime(p)`*, en utilisant la commande* `showtime`.

▸ **Exercice 83.** *Un entier est dit parfait s'il est égal à la somme de ses diviseurs (à l'exception de lui-même mais en comptant 1).*

1. *Ecrivez une procédure* `Parfait` *prenant en argument un entier* `n`*, testant si cet entier est parfait. On retournera utilement la valeur* `true` *ou* `false`.

2. *Utilisez cette fonction pour écrire une fonction* `kParfait` *prenant en argument un entier* `k` *et retournant la liste des* `k` *premiers nombres parfaits.*

3. *Les nombres parfaits auraient-ils tendance à n'être que pairs ?*

▸ **Exercice 84.** *L'objet de cet exercice est de comprendre ce que nous apporte la phrase suivante : "Maple réalise une arithmétique exacte sur entier". Proposer une procédure calculant le* i^{eme} *terme d'une suite définie par récurrence. La suite devra converger vers une certaine valeur en calcul entier, et vers une toute autre valeur en calcul flottant.*

8 Librairie de fonctions

Maintenant que nous avons un certain nombre d'éléments en notre possession, nous pouvons envisager le développement notre propre librairie de fonctions. Nous avons vu, dans l'introduction, qu'en définitive *Maple* se réduisait à un petit noyau—moteur de calcul—autour duquel venait graviter un ensemble impressionnant de fonctions directement programmées en langage *Maple*. Pour être précis à ce sujet, nous pouvons classer l'ensemble des fonctions en quatre catégories :

1. **fonctions builtin** : ce sont les fonctions internes (du noyau) au système. Leur appartenance au noyau est tout naturellement justifiée par un besoin d'efficacité. Les sources de ces fonctions ne sont d'ailleurs pas accessibles ;

2. **fonctions directement accessibles** : ces fonctions sont immédiatement utilisables par l'utilisateur. Pourtant, elles ne sont pas à proprement parler chargées au lancement de *Maple*. En fait, elles correspondent à un appel à la fonction `readlib`. Supposons par exemple que la fonction `f` fasse partie de cette catégorie de fonctions; le premier appel à la fonction `f` se déroulera comme suit :

   (a) lecture du fichier `f.m` dans la librairie *Maple* (dont le chemin sur le disque est spécifié par la variable `libname`). Suite à cette lecture, `f` est assignée d'une valeur—en fait une procédure—et cette valeur est retournée par la fonction `readlib`;

   (b) appel de la dite fonction `f` avec ses arguments.

   La fonction `gcd` entre dans cette catégorie et nous le vérifions aisément par :

```
> eval(gcd, 1);

                    readlib('gcd')
```

   Ces fonctions qui sont directement disponibles sont ainsi déclarées par une instruction du type : `f := 'readlib('f')';`. Il doit être clair que ces fonctions sont écrites en langage *Maple* ;

3. **fonctions à charger** : ces fonctions sont identiques aux précédentes à ceci près que leur utilisation correspond au préalable à un ordre explicite (de la part de l'utilisateur) de chargement. Cette action peut être réalisée à l'aide de la fonction `readlib`;

4. **fonctions de packages** : ces fonctions sont définies au sein des librairies. Leur utilisation nécessite le chargement préalable de la libraire en question (*package*), voire seulement d'une partie de cette libraire.

Nous nous attachons à regarder plus précisément la création d'une telle librairie de fonctions. Ces librairies permettent en fait de regrouper des fonctions traitant d'un sujet particulier. Elles apportent énormément de confort d'utilisation en ce sens qu'une librairie est un ensemble de fonctions, cohérent et réfléchi. Outre ces considérations quelque peu subjectives, nous nous attachons maintenant à observer la structure d'un *package*. Supposons que `PAC` dénote le nom d'un package contenant deux fonctions `f1` et `f2`. La notion fondamentale qu'il faut retenir est la suivante : **un package est une table contenant les définitions de ses propres fonctions**. Pour être très précis à ce sujet, regardons la définition de notre librairie `PAC` :

```
> PAC[f1]:='readlib(`PAC/f1`,`f1.m`)';
   PAC[f1] := readlib(PAC/f1, f1.m)
> PAC[f2]:='readlib(`PAC/f2`,`f2.m`)';
   PAC[f2] := readlib(PAC/f2, f2.m)
```

Ces lignes définissent `PAC` comme un *package*.

```
> eval(PAC);
     table([
         f1 = readlib(PAC/f1, f1.m)
         f2 = readlib(PAC/f2, f2.m)
     ])
```

Pourtant, pour que cela ait un sens, il faudrait sauver ces instructions dans un fichier `PAC.m`. Lorsque l'utilisateur donnera l'ordre explicite de chargement de la libraire `PAC`, via l'instruction `with(PAC);`, cela aura pour effet de réaliser les assignations :

```
> f1 := PAC[f1]:
> f2 := PAC[f2]:
```

Les fonctions `f1` et `f2` seront alors directement utilisables par leur noms, respectivement `f1` et `f2`. Cependant, un appel à `PAC[f1]` sera tout à fait(3) équivalent à un appel à `f1`.

Nous serions donc tenter de conclure qu'il est aisé de développer sa propre librairie de fonctions. Ce serait se masquer les problèmes

(3) Et d'ailleurs, c'est très utile dans le cas d'un conflit de nom de fonction entre deux librairies.

suivants : il faut organiser très soigneusement son travail; il faut réfléchir aux fonctions qui seront mises à disposition, à leurs noms par exemple (lequels devront être intuitifs pour la majorité pour ne pas dire la totalité des utilisateurs potentiels) ; il faut aussi incorporer les aides en lignes sur la librairie et toutes ses fonctions ; il faut finalement rester cohérent avec tous les *packages* déjà existants afin que votre produit s'intègre parfaitement dans l'ensemble des *packages Maple* déjà distribués. Par contre, lorsque c'est bien fait, c'est très utile et très satisfaisant.

Nous nous proposons de développer un petit *package* permettant de manipuler des automates. Un automate sera représenté[1] en *Maple* par une liste de trois éléments. Le premier est un entier qui correspond à l'état initial, le second est un ensemble d'entiers représentant les états finaux et le troisième est la table (au sens *Maple*) qui correspond à la table de transition. Nous utiliserons une table à double indices de sorte que `t[1][a]={2,3}` représentera les transitions $(1, a, 2)$ et $(1, a, 3)$. Plus précisément, nous représenterons les transitions possibles à partir d'un état donné, par une table qui à une lettre associe un ensemble d'états d'arrivés. Voici un exemple d'automate codé selon ce format :

```
> [1, {5}, table([1=table([a={2,4}]),
                  2=table([b={3}]),
                  3=table([a={2,4}]),
                  4=table([b={5}])])];

       [1, {5}, table([           ]
                 1 = table([
                       a = {2, 4}
                     ])
                 2 = table([
                       b = {3}
                     ])
                 3 = table([
                       a = {2, 4}
                     ])
                 4 = table([
                       b = {5}
                     ])
              ])
```

▸ **Exercice 85.** *Définissez une librairie* `AUTO`*—AUTomat—dans un fichier de nom* `AUTO`*. N'oubliez pas de modifier votre fichier* `.mapleinit` *afin que Maple trouve votre librairie : on définira utilement une variable* `AUTOLib` *stockant cette information.*

▸ **Exercice 86.** *Intégrez à votre librairie la fonction* `Recognize` *permettant de tester si une chaîne de caractères est reconnue par un automate donné. On pourra prévoir que les automates sont non-déterministes avec des transisitons possibles par ϵ représentée par* `eps`.

▸ **Exercice 87.** *Intégrez à votre librairie la fonction* `Letter2Auto` *prenant une lettre en paramètre et retournant l'automate reconnaissant cette lettre.*

▸ **Exercice 88.** *Intégrez la fonction* `ConcatAuto` *prenant deux automates en paramètres et retournant la concaténation de ces deux automates.*

▸ **Exercice 89.** *Ajoutez aussi la fonction* `UnionAuto` *retournant l'automate union de deux automates passés en paramètres.*

▸ **Exercice 90.** *Intégrez la fonction* `KleenAuto` *retournant l'automate correspondant à la fermeture de Kleene de l'automate passé en paramètre.*

▸ **Exercice 91.** *Définissez les opérateurs infixes correspondant aux fonctions suivantes :*

1. `&C` *: pour la concaténation ;*
2. `&U` *: pour l'union ;*
3. `&K` *: pour la fermeture de Kleene.*

▸ **Exercice 92.** *Intégrez à votre librairie la fonction* `Reg2Auto` *prenant en paramètre une expression régulière et retournant l'automate correspondant. Par exemple, la fonction devra pouvoir construire l'automate correspondant à l'expression $a^*.b.(a+b)^*$ codé par* `(&K A) &C B &C (&K (A &U B))` *où* `A:=Letter2Auto(a)` *et* `B:=Letter2Auto(b)`.

▸ **Exercice 93.** *Intégrez la fonction* `ReduceAuto` *permettant d'émonder un automate passé en paramètre.*

▸ **Exercice 94.** *Ajoutez la fonction* `IsDeterAuto` *testant si un automate passé en paramètre est déterministe.*

▸ **Exercice 95.** *Intégrez la fonction* `DeterAuto` *déterminisant un automate passé en paramètre.*

▸ **Exercice 96.** *Intégrez la fonction* `MiniAuto` *minimisant un automate passé en paramètre.*

▸ **Exercice 97.** *Prévoir une fonction* `Auto2Reg` *affichant l'expression régulière correspondant à un automate passé en paramètre.*

▸ **Exercice 98.** *Ajouter la fonction* `InterAuto` *d'intersection de deux automates passés en paramètre.*

▸ **Exercice 99.** *Ajoutez l'aide en ligne de votre librairie et les aides en ligne pour toutes les fonctions de votre librairie.*

▸ **Exercice 100.** *Maintenant que vous avez quelques techniques de développement, attaquez-vous au délicat travail d'amélioration des fonctions de la librairie Maple. Elles sont programmées en Maple, donc pas de problème, à vos claviers...*

Bibliographie

[1] C.Caron, *AG Manuel de l'utilisateur*, LIR, **96.09**, Université de Rouen, 1996.

[2] B.W.Char, K.O.Geddes, G.H.Gonnet, B.L.Leong, M.B.Monagan et S.M.Watt, *Maple V : First Leaves*, Springer-Verlag.

[3] B.W.Char, K.O.Geddes, G.H.Gonnet, B.L.Leong, M.B.Monagan et S.M.Watt, *Maple V : Language Reference Manual*, Springer-Verlag.

[4] B.W.Char, K.O.Geddes, G.H.Gonnet, B.L.Leong, M.B.Monagan et S.M.Watt, *Maple V : Library Reference Manual*, Springer-Verlag.

[5] J.Davenport, Y.Siret et E.Tournier, *Calcul formel, Systèmes et algorithmes de manipulations algébriques*, Études et recherches en informatique, MASSON, 1993.

[6] K.Forsman, *A short Maple Primer*, Dept. of Electrical Engineering, Linköping University, S-581 83 Linköping, Sweden, February 28, 1992.

[7] C.Gomez, *Notes de cours Maple*, INRIA, Domaine de Voluceau, Rocquencourt - BP105, 78153 Le Chesnay Cedex, France, Juin 1995.

[8] D.Gruntz, K.Meier and M.Monagan, *Maple: an introduction*, Institute for Scientific Computation, ETH Zürich, March 12, 1992.

[9] B.F.Madore, *Some Maple Questions and Answers*, July 6, 1994.

[10] M.Monagan, *Programming in Maple: The Basics*.

[11] M.Monagan, *The Maple Computer Algebra System*.

[12] D.Redfern, *The Maple Handbook (maple V release 3)*, Springer-Verlag.

[13] Symbolic Computation Group, *New features in Maple V Release 2*, University of Waterloo, Waterloo, Ontario, N2L 3G1, Canada.

[14] Symbolic Computation Group, *New features that have been added to Maple for version V*, University of Waterloo, Waterloo, Ontario, N2L 3G1, Canada.

[15] S.Veigneau, *ACE, an Algebraic Combinatorics Environment for the computer algebra system Maple : User's reference manual, Version 2.0*, Université de Marne-la-Vallée 1996.