



L'indexation: liste, table, array, rtable-Array

© Pierre Lantagne

Enseignant retraité du Collège de Maisonneuve

Ce document est une révision de celui produit en 2004. L'objectif principal de ce document est d'introduire le lecteur aux objets indexés et à leurs manipulations afin que le lecteur puisse acquérir une meilleure compréhension du logiciel.

Bonne lecture à tous !

* Ce document Maple est exécutable avec la version 2020.2

Initialisation

```
> restart;
> arbre:=proc(expr)
  local k,Options_base,Opts,plot1,plot2,Size;
  global affx,affy,affT,affS,marque;
  Opts:=[args[2..nargs]];
  Options_base:=NULL;
  for k from 1 to nops(Opts) do
    if lhs(map(op,k,Opts))=size then Options_base:=size=rhs(Opts[k])
  fi;
  od;
  marque:=proc(expr)
    local xa,xb,pos,lpos;
    global affx,affy,affT,affS;
    if type(expr,symbol) or type(expr,integer) then
      affT:=affT,[affx,affy,expr];
      xa:=affx;
      affx:=affx+1;
      xa
    else
      affy:=affy-1;
      lpos:=map(marque,[op(expr)]);
      xa:=op(1,lpos);
      xb:=op(nops(lpos),lpos);
      affy:=affy+1;
      pos:=(xa+xb)/2;
      affT:=affT,[pos,affy,op(0,expr)];
      affS:=affS,op(map(proc(a,b) CURVES([[a,affy-0.85],[b,affy-.15]],
\
      COLOR(RGB,0.6156862700, 0.0156862750, 0.6313725500))end,
lpos,pos));
```

```

        pos;
    fi
end proc;
affx:=0; affy:=0; affT:=NULL; affS:=NULL;
marque(expr);
plot1:=plots[textplot]({affT}, font=[TIMES,ROMAN,10], color="MapleV
21");
plot2:=PLOT(affS, THICKNESS(1), LINESSTYLE(2));
unassign('affx,affy,affT,affS,marque');
plots[display]({plot1,plot2}, axes=NONE, Options_base);
end proc;

```

Il y a plusieurs fonctionnalités de la bibliothèque de base qui permettent la création et la manipulation d'expressions indexées.

- les objets listes (list)
- les objets tables (table)
- les objets tableaux (array)
- les objets tableaux (rtable - Array)

Liste (list)

Une [liste](#) est créée en énumérant une [séquence](#) d'expressions que l'on place entre crochets. La présence des crochets implique que l'ordre dans lequel s'effectue l'énumération des éléments de la séquence est pris en compte par Maple.

Soit la liste $L = [\ln(3), \textit{Francine}, 7, \int x^2 dx]$.

```

> L:=[ln(3),Francine,7,Int(x^2,x)];
          L := [ln(3),Francine,7,∫x2 dx]

```

(2.1)

Comme pour la plupart des objets Maple, on peut faire afficher le détail de l'objet lors de l'appel par leur nom dans une zone de requêtes.

```

> L;
          [ln(3),Francine,7,∫x2 dx]

```

(2.2)

Il est possible d'obtenir le nombre d'éléments composant une liste.

```

> numelems(L);
          4

```

(2.3)

Voici la manière dont Maple a mémorisé l'objet L.

```

> whattype(L);
          list

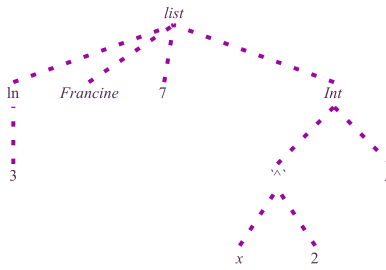
```

(2.4)

```

> arbre(L,size=[300,200]);

```



L'indexation des éléments d'une liste correspond à l'ordre d'énumération des éléments composant la liste. Ainsi, pour pointer vers le deuxième élément de la liste L , il suffit d'énoncer le nom de la liste suivi du nombre entier spécifiant le rang de l'élément dans la liste. Ce nombre entier doit être écrit entre crochets:

- entier positif: de la gauche vers la droite dans la liste
- entier négatif: de la droite vers la gauche dans la liste

```
> L[2];
    Francine
L[-3];
    Francine
```

(2.5)

Il est possible de pointer vers tous les éléments d'une liste comme suit

```
> L[];
    ln(3), Francine, 7, ∫x² dx
```

(2.6)

Nous pouvons modifier les éléments d'une liste existante de plusieurs manières

- On peut ajouter un ou de plusieurs éléments à une liste existante comme suit $[op(L), x, y, z]$.

```
> L:=[op(L), a, b, c];
    L := [ln(3), Francine, 7, ∫x² dx, a, b, c]
```

(2.7)

- On peut remplacer un élément de rang k d'une liste existante par un autre élément comme suit $subsop(i=x, L)$.

```
> L:=subsop(2=Pierre, L);
    L := [ln(3), Pierre, 7, ∫x² dx, a, b, c]
```

(2.8)

Si la liste ne comporte pas un trop grand nombre d'éléments, on peut assigner directement un élément pour remplacer celui de rang k .

```
> L[6]:=t;
    L := [ln(3), Pierre, 7, ∫x² dx, a, t, c]
```

(2.9)

```
> L;
    [ln(3), Pierre, 7, ∫x² dx, a, t, c]
```

(2.10)

- On peut supprimer un élément de rang k d'une liste existante comme suit $subsop(i=NULL, L)$.

```
> L:=subsop(4=NULL, L);
    L := [ln(3), Pierre, 7, a, t, c]
```

(2.11)

Pour terminer ce survol, spécifions que des éléments d'une liste peuvent être aussi des listes et/ou bien des

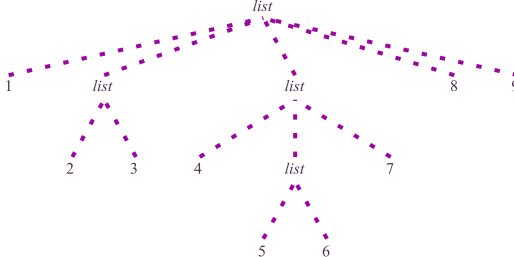
listes imbriquées dans une liste.

Par exemple, soit la liste $T := [1, [2, 3], [4, [5, 6], 7], 8, 9]$.

```
[ > T := [1, [2, 3], [4, [5, 6], 7], 8, 9];  
      T := [1, [2, 3], [4, [5, 6], 7], 8, 9] (2.12)
```

Voici la manière dont Maple a mémorisé l'objet T.

```
[ > arbre(T, size=[400, 200]);
```



Pointons vers le chiffre 6 de deux manières:

```
[ > T[3][2][2];  
      6 (2.13)
```

```
[ > T[3, 2, 2];  
      6 (2.14)
```

Table (table)

```
[ > T := table();  
      T := table([ ]) (3.1)
```

Il est possible de réaliser une indexation des éléments d'une expression en ne passant pas obligatoirement par les nombres entiers. Un tel objet indexé est appelée une `table`. L'indexation des éléments d'une table peut être faite au moment la création de la table ou après. Contrairement aux listes, l'indexation peut être faite de manière dynamique.

La création d'un objet de type `table` peut être réalisée explicitement avec la macro-commande `table`. Nous allons créer une table de manière explicite en spécifiant une liste d'égalités comme entrées initiales de la table. Le membre de gauche de chaque égalité sera l'index du membre de droite dans cette table.

```
[ > L := [ Catherine = `450-123-4567`,  
      Anne = `514-777-5533`,  
      Jacinthe = `418-765-4321`,  
      Pierre = `514-987-6541`  
      ];  
      L := [ Catherine = 450-123-4567, Anne = 514-777-5533, Jacinthe = 418-765-4321, Pierre  
      = 514-987-6541 ] (3.2)
```

```

> No_téléphone:=table(L);
No_téléphone :=table([Catherine = 450-123-4567, Jacinthe = 418-765-4321, Anne
= 514-777-5533, Pierre = 514-987-6541])

```

(3.3)

Contrairement à la plupart des objets Maple (comme les listes) , on ne peut faire afficher le détail de l'objet lors de l'appel par son nom dans une zone de requêtes. Les objets de type `table` possède une règle de simplification spéciale tout comme pour les procédures.

```

> No_téléphone;
No_téléphone

```

(3.4)

```

> whattype(No_téléphone);
table

```

(3.5)

Pour pointer vers la table elle-même, nous devons saisir `op(No_téléphone)` :

```

> op(No_téléphone);
table([Catherine = 450-123-4567, Jacinthe = 418-765-4321, Anne = 514-777-5533, Pierre
= 514-987-6541])

```

(3.6)

Pour pointer vers la liste des éléments de la table, nous devons saisir `op(op(No_téléphone))` :

```

> op(op(No_téléphone));
[Catherine = 450-123-4567, Jacinthe = 418-765-4321, Anne = 514-777-5533, Pierre
= 514-987-6541]

```

(3.7)

Nous pouvons obtenir la séquence des indices

```

> indices(No_téléphone);
indices(No_téléphone, 'nolist');
[Catherine], [Jacinthe], [Anne], [Pierre]
Catherine, Jacinthe, Anne, Pierre

```

(3.8)

et la liste des entrées.

```

> entries(No_téléphone);
entries(No_téléphone, 'nolist');
[450-123-4567], [418-765-4321], [514-777-5533], [514-987-6541]
450-123-4567, 418-765-4321, 514-777-5533, 514-987-6541

```

(3.9)

```

> indices(No_téléphone, 'pairs');
Catherine = 450-123-4567, Jacinthe = 418-765-4321, Anne = 514-777-5533, Pierre = 514-987-6541

```

(3.10)

Comme pour une liste, nous pouvons pointer vers un élément particulier d'une table en spécifiant son index.

```

> No_téléphone[Catherine];
450-123-4567

```

(3.11)

Il est possible d'obtenir le nombre d'éléments composant une table.

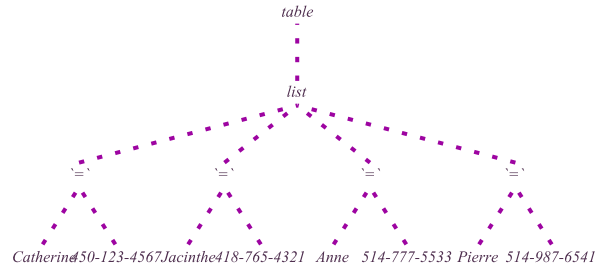
```
> numelems(No_téléphone);
4 (3.12)
```

Voici la manière dont Maple a mémorisé l'objet L.

```
> whattype(No_téléphone);
table (3.13)
```

```
> whattype(op(1, No_téléphone)); # C'est l'objet table lui-même
table (3.14)
```

```
> arbre(op(1, No_téléphone), size=[450, 200]);
```



Remarque: La mémorisation de chaque entrée a été faite de la gauche vers la droite car nous avons créé la table *No_téléphone* avec une liste initiale d'entrées.

Il est possible d'ajouter et de supprimer des entrées d'une table existante comme suit:

- Pour ajouter une entrée dans une table

```
> No_téléphone[Maxime] := `514-888-8888`;
No_téléphone_Maxime := 514-888-8888 (3.15)
```

```
> op(1, No_téléphone);
table([Catherine = 450-123-4567, Jacinthe = 418-765-4321, Anne = 514-777-5533, Pierre = 514-987-6541, Maxime = 514-888-8888]) (3.16)
```

- Pour supprimer une entrée dans une table

```
> No_téléphone[Pierre] := evaln(No_téléphone[Pierre]);
No_téléphone_Pierre := No_téléphone_Pierre (3.17)
```

```
> op(1, No_téléphone);
table([Catherine = 450-123-4567, Jacinthe = 418-765-4321, Anne = 514-777-5533, Maxime = 514-888-8888]) (3.18)
```

Nous n'allons pas davantage approfondir l'étude des objets de type *table*.

Tableau (array)

Attention: cette section est strictement pour votre information. La macro-commande `array` est obsolète. Les tableaux créés avec cette macro-commande sont des objets dont la structure est incompatible avec les macro-

commandes de plusieurs bibliothèques telles que `LinearAlgebra`, `Student[LinearAlgebra]`, `VectorCalculus`, etc.

Dans une liste, l'indexation des éléments est réalisée avec des nombres entiers commençant avec le nombre 1. La macro-commande `array` de la bibliothèque de base généralise le concept de `table`. Un tableau (`array`) possède des règles d'évaluation semblables pour les objets de type `table`. La dimension du tableau, c'est-à-dire le nombre d'emplacements à être indexés, peut être spécifiée soit implicitement (dynamiquement), soit explicitement (déclaration). Les indices de l'indexation ne sont plus limités aux seuls nombres entiers commençant avec le nombre 1: les nombres entiers négatifs et même le nombre 0 sont aussi permis dans ce cas.

La création d'une *structure de données* d'éléments indexés de type `array` est réalisée avec la macro-commande `array`.

Déclarons un *tableau à une dimension* (un seul ensemble d'indices) de quatre emplacements en commençant l'indexation avec l'entier -1.

```
> T:=array(-1..2);  
      T := array( -1 ..2, [ (-1) = ?-1, (0) = ?0, (1) = ?1, (2) = ?2 ] )
```

 (4.1)

Le résultat précédent montre que le tableau créé est initialement vide (paire de crochets " vide "). En créant le tableau T de cette manière, nous venons de créer une *structure de données* nommée T ayant quatre emplacements indicés T_{-1} , T_0 , T_1 et T_2 dont le contenu de chacun d'entre-eux est la séquence `NULL`.

Pour pointer vers le contenu d'un emplacement particulier d'un tableau à une dimension, on fait exactement comme pour un objet de type `table`: on précise d'abord le nom du tableau suivi de l'indice de l'emplacement visé. On doit écrire l'indice entre crochets.

```
> T[-1];  
      T[0];  
      T[1];  
      T[2];  
  
      T-1  
      T0  
      T1  
      T2
```

 (4.2)

Pour pointer vers le contenu d'un emplacement particulier d'un tableau à une dimension, on fait exactement comme pour un objet de type `table`: on précise d'abord le nom du tableau suivi de l'indice de l'emplacement visé. On doit écrire l'indice entre crochets. Assignons une expression particulière à chaque emplacement T_{-1} , T_0 , T_1 et T_2 du tableau T .

```
> T[-1]:=ln(3);  
      T[0]:=Francine;  
      T[1]:=7;  
      T[2]:=Int(x^2,x);  
  
      T-1 := ln(3)  
      T0 := Francine  
      T1 := 7
```

$$T_2 := \int x^2 dx \quad (4.3)$$

Comme pour les objets de type `table`, on ne peut faire afficher le détail du tableau T lors de l'appel par son nom dans une zone de requêtes.

```
> T;
```

T

(4.4)

```
> whattype(T);
```

array

(4.5)

Pour pointer vers le tableau lui-même, nous devons saisir `op(T)` :

```
> op(T);
whattype(op(T));
```

$\text{array}(-1..2, [(-1) = \ln(3), (0) = \textit{Francine}, (1) = 7, (2) = \int x^2 dx])$

array

(4.6)

Le résultat précédent montre que cet objet possède deux opérandes: la liste des indices d'abord puis la liste des égalités de l'indexation.

Pour pointer vers chaque opérandes de ce tableau, nous devons saisir `op(2,op(T))` et `op(3,op(T))`.

```
> op(2,op(T));
op(3,op(T));
```

$-1..2$

$[-1 = \ln(3), 0 = \textit{Francine}, 1 = 7, 2 = \int x^2 dx]$

(4.7)

Pour pointer vers les entrées du tableau, il faut saisir `op(op(op(No_téléphone)))`.

```
> indices(T);
indices(T, 'nolist');
```

$[-1], [0], [1], [2]$

$-1, 0, 1, 2$

(4.8)

```
> entries(T, 'nolist');
entries(T, 'pairs')
```

$\ln(3), \textit{Francine}, 7, \int x^2 dx$

$-1 = \ln(3), 0 = \textit{Francine}, 1 = 7, 2 = \int x^2 dx$

(4.9)

Tout comme dans le cas d'un objet de type `table`, nous pouvons pointer vers un élément particulier d'un tableau en spécifiant son indice.

```
> T[0];
```

Francine

(4.10)

Il est possible d'obtenir le nombre d'éléments composant un tableau.

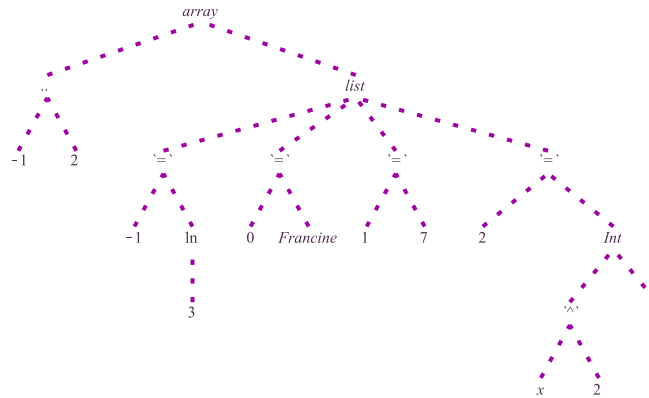

```
> numelems(T);
```

4

(4.11)

Voici comment a été mémorisé ce tableau.

```
> arbre(op(T),size=[500,300]);
```



On aurait pu également effectuer ces assignations au moment même de la création du tableau. Dans ce cas, chaque emplacement T_{-1} , T_0 , T_1 et T_2 du tableau T est aussitôt assigné et la structure elle-même s'affiche automatiquement (évidemment dans le cas où la requête est terminée par le point-virgule).

Créons de cette manière le tableau à une dimension $T = [\ln(3) \text{ Francine } 7 \int x^2 dx]$.

Il y a deux syntaxes pour la saisie directe d'un tableau à une dimension. Pour chaque cas, afin que les assignations puissent se faire dans l'ordre, on doit énumérer, *dans une liste*, les expressions qui doivent être assignées respectivement aux variables (emplacements) T_{-1} , T_0 , T_1 et T_2 .

– une saisie *explicite* où il faut d'abord déclarer, à l'aide de l'opérateur «.», le nombre d'emplacements à réserver pour le tableau suivi de la liste des expressions à être assignées. *Dans le cas où les indices commencent avec l'entier 1, l'affichage du tableau est semblable à celui d'une liste.*

```
> T:=array(-1..2,[ln(3), Francine, 7, Int(x^2,x)]);
```

$$T := \text{array} \left(-1..2, \left[(-1) = \ln(3), (0) = \text{Francine}, (1) = 7, (2) = \int x^2 dx \right] \right)$$

(4.12)

– une saisie *implicite* où il suffit de donner la liste des expressions à être assignées. Dans ce cas, le nombre d'emplacements à réserver pour le tableau correspond au nombre d'expressions composant la liste. *Dans le cas d'une saisie implicite, les indices de l'indexation commencent, par défaut, avec l'entier 1 et il y a l'affichage du tableau lui-même.*

```
> T:=array([ln(3), Francine, 7, Int(x^2,x)]);
```

$$T := \left[\ln(3) \text{ Francine } 7 \int x^2 dx \right]$$

(4.13)

Remarque: Notons que lorsque le format du tableau n'est pas explicitement déclaré, il y a affichage du tableau lui-même et que l'affichage est semblable à celui d'une liste sauf que les virgules sont omises.

Que ce soit de manière explicite ou implicite, chacune de ces manières crée un objet de même nature. Et dans

un cas comme dans l'autre, les règles d'évaluation des objets de type `array` sont différentes.

Affichons autrement le tableau `T` en l'appelant par son nom.

```
> T;
                                     T
(4.14)
```

Contrairement à ce qui était attendu, le tableau lui-même ne s'est pas affiché. Dans le cas de ce type de *structure de données* (`array`), il en sera toujours ainsi: un objet de type `array` a ses propres règles de simplification automatique. Le nom `T` ayant été assigné à un objet de type `array`, la requête `T` a été simplifiée par le nom `T` et non pas par ce que vers quoi ce nom pointe, c'est-à-dire son contenu. C'est pour une raison d'efficacité: imaginons l'affichage non désirée des éléments dans le contexte des manipulations formelles de tableaux de très grands formats.

Par contre, les tableaux eux-mêmes peuvent être affichés si on utilise l'une ou l'autre des macro-commandes suivantes de la bibliothèque de base:

- `eval`
- `op`
- `print`

```
> op(T);
                                     [ ln(3) Francine 7 ∫x² dx ]
(4.15)
```

En résumé, la création avec la macro-commande `array` d'un tableau `T` à une dimension définit `T` comme une structure de données où

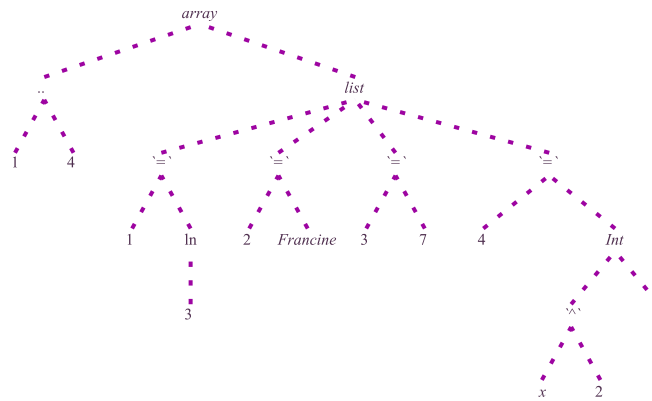
- `T` est le nom de la structure de données, donc un symbole

```
> V;
whattype(V);
                                     V
                                     symbol
(4.16)
```

- `eval(V)` est le tableau lui-même.

```
> eval(T);
whattype(eval(T));
                                     [ ln(3) Francine 7 ∫x² dx ]
                                     array
(4.17)
```

```
> arbre(op(T),size=[500,300]);
```



Attention, même si un tableau dont l'ensemble des indices ne commence pas avec l'entier 1 est quand même un objet de type `array`. Par exemple, créons un tableau `S` composé de ces quatre mêmes expressions mais où l'ensemble des indices commencera avec l'entier -2.

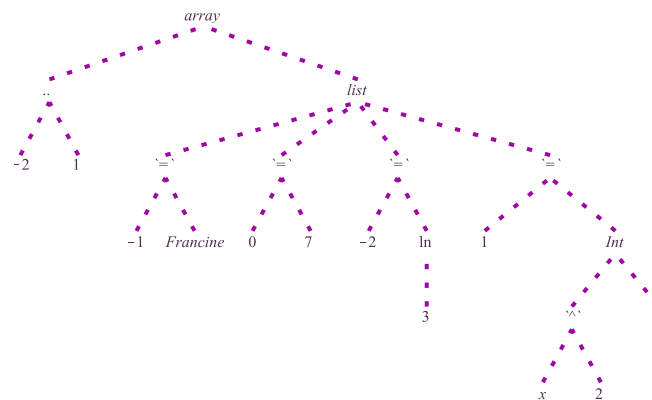
```
> S:=array(-2..1,[ln(3), Francine, 7, Int(x^2,x)]);
      S := array( -2..1, [ (-2) = ln(3), (-1) = Francine, (0) = 7, (1) = ∫x2 dx ] )
```

(4.18)

```
> whattype(eval(S));
      array
```

(4.19)

```
> arbre(op(S),size=[500,300]);
```



Il est primordial de toujours avoir à l'esprit la nature des objets qui sont créés. On comprendra mieux pourquoi, parfois, certaines opérations sur des objets sont permises et d'autres pas. En effet, il est facile de comprendre que la définition et le résultat même d'une opération dépend de la nature des opérandes d'un objets.

Lorsque, dans un tableau à une dimension, les indices commencent avec l'entier 1, le tableau est également reconnu comme un objet de type `vector` et, dans ce cas, on dit aussi que `T` est un vecteur.

```
> type(T,array);
      type(T,vector);
      true
      true
```

(4.20)

```
> type(S,array);
      type(S,vector);
      true
      false
```

(4.21)

La macro-commande `vector` de la bibliothèque de base facilite la création d'un vecteur.

Comme pour les tableau (`array`), il y a deux syntaxes par la saisie d'un vecteur avec la macro-commande `vector`.

- Une saisie *explicite* où il faut d'abord indiquer le nombre d'emplacements à réserver pour le vecteur suivi de la liste des expressions à être assignées.

$$\left[\begin{array}{l} > \mathbf{V:=vector(4,[ln(3), Francine, 7, Int(x^2,x)]); \\ \mathbf{V} := \left[\begin{array}{cccc} \ln(3) & \text{Francine} & 7 & \int x^2 dx \end{array} \right] \end{array} \right. \quad (4.22)$$

- Une saisie *implicite* (dynamique) où il suffit de donner la liste des expressions à être assignées. Dans ce cas, le nombre d'emplacements à réserver pour le vecteur correspond au nombre d'expressions composant la liste.

$$\left[\begin{array}{l} > \mathbf{V:=vector([ln(3), Francine, 7, Int(x^2,x)]); \\ \mathbf{V} := \left[\begin{array}{cccc} \ln(3) & \text{Francine} & 7 & \int x^2 dx \end{array} \right] \end{array} \right. \quad (4.23)$$

Remarque: un vecteur est *une structure de données* de type `array` à une dimension dont les indices commencent avec l'entier 1. En conséquence, un objet de type `list` n'est pas un vecteur.

La pertinence d'un objet Maple de type `array` est davantage révélée avec la possibilité de manipuler des tableaux indexés ayant plus d'une dimension: deux, trois, ...etc.

La création d'un tableau à deux dimensions doit prendre en compte deux ensembles d'indices:

- le premier ensemble d'indices concerne les lignes du tableau
- le second concerne les colonnes.

$$\left[\begin{array}{l} > \mathbf{M:=array(1..2,1..3); \\ \mathbf{M} := \left[\begin{array}{ccc} ?_{1,1} & ?_{1,2} & ?_{1,3} \\ ?_{2,1} & ?_{2,2} & ?_{2,3} \end{array} \right] \end{array} \right. \quad (4.24)$$

Comme pour tout objet de type `array`, les assignations aux emplacements indicés peuvent être effectuées après la création du tableau ou au moment de sa création. Pour que les assignations puissent s'effectuer dans l'ordre au moment de la création du tableau, il faut énumérer, *dans une liste*, les lignes qui composeront le tableau. *Chaque ligne de cette liste est elle-même une liste* d'expressions qui seront assignées, dans l'ordre, aux emplacements de chaque colonne de cette ligne. Il faut, bien sûr, que toutes les listes de la liste des lignes possèdent le même nombre d'éléments sinon un message d'erreur sera affiché.

Créons le tableau à deux dimensions $M = \begin{bmatrix} 0 & 2 & 4 \\ 1 & 3 & 5 \end{bmatrix}$.

La création du tableau `M` dans la forme précédente n'est possible seulement si chaque ensemble des indices commencent avec l'entier 1.

Il y a deux syntaxes par la saisie d'un tableau à deux dimensions avec la macro-commande

array.

- Une saisie *explicite* où il faut d'abord indiquer, à l'aide de l'opérateur «.», les indices à utiliser pour l'indexation des lignes et les indices à utiliser pour l'indexation des colonnes, suivi de l'énumération, dans une liste, des lignes qui composeront le tableau. Chaque ligne de cette liste est elle-même une liste d'expressions qui seront assignées, dans l'ordre, aux emplacements de chaque colonne de cette ligne.

```
> M:=array(1..2,1..3,[ [0,2,4],[1,3,5] ]);
```

$$M := \begin{bmatrix} 0 & 2 & 4 \\ 1 & 3 & 5 \end{bmatrix}$$

(4.25)

- Une saisie *implicite* (dynamique) où il suffit de donner la liste de listes d'expressions à être assignées. Dans ce cas, les indices qui seront utilisés pour l'indexation des lignes commencera avec l'entier 1 et se terminera avec le nombre d'éléments de la liste des lignes et l'indexation des colonnes commencera également avec l'entier 1 pour se terminer avec la valeur correspondant au nombre d'expressions de chaque ligne.

```
> M:=array( [ [0,2,4],[1,3,5] ]);
```

$$M := \begin{bmatrix} 0 & 2 & 4 \\ 1 & 3 & 5 \end{bmatrix}$$

(4.26)

En créant ainsi le tableau M comme une *structure de données* de type array à deux dimensions, il y a eu six (2x3) assignations, dans l'ordre, aux emplacements (variables): M_{11} , M_{12} , M_{13} , M_{21} , M_{22} et M_{23} .

Remarquons que les assignations se font par ligne.

Pour pointer vers un emplacement particulier du tableau M, il suffit de taper le nom du tableau suivi des indices de la ligne et de la colonne de l'emplacement visé. L'énumération des indices de la ligne et de la colonne doit être placée entre crochets. Par exemple, pour pointer vers la valeur 5 de la matrice M, l'emplacement de ce nombre est indiqué par la valeur 2 du premier ensemble d'indice (2ème ligne) et par la valeur 3 du second ensemble d'indice (3ème colonne).

```
> M[2,3];
```

5

(4.27)

Ce qui est fort différent de

```
> M[3,2];
```

```
Error, 1st index, 3, larger than upper array bound 2
```

Lorsque, dans un tableau à deux dimensions, les indices des lignes et des colonnes commencent avec l'entier 1, le tableau est également reconnu comme un objet de type [matrix](#).

```
> type(M,array);
```

```
type(M,matrix);
```

true

true

(4.28)

La macro-commande `matrix` de la bibliothèque de base facilite quelque peu la création d'un tableau à deux dimensions (matrice) où chaque ensemble des indices commence avec l'entier 1.

Il y a trois syntaxes pour la saisie d'une matrice avec la macro-commande `matrix`.

- Une saisie *explicite abrégée* où il faut d'abord déclarer le format de la matrice, c'est-à-dire, donner

directement le nombre de lignes et le nombre de colonnes composant la matrice suivi de la liste des expressions à être assignées aux emplacements de la matrice. L'assignation des expressions est faite selon les lignes.

```
> M:=matrix(2,3,[0,2,4,1,3,5]);
```

$$M := \begin{bmatrix} 0 & 2 & 4 \\ 1 & 3 & 5 \end{bmatrix} \quad (4.29)$$

– Une saisie *explicite* où il faut d'abord déclarer le format de la matrice suivi de la liste des listes d'expressions composant chaque ligne de la matrice.

```
> M:=matrix(2,3,[ [0,2,4],[1,3,5] ]);
```

$$M := \begin{bmatrix} 0 & 2 & 4 \\ 1 & 3 & 5 \end{bmatrix} \quad (4.30)$$

– Une saisie *implicite* où il suffit de donner la liste des listes d'expressions à être assignées. Dans ce cas, le nombre de lignes de la matrice correspond au nombre d'éléments de cette la liste et le nombre de colonnes correspondra au même nombre d'éléments de chaque ligne de la liste des lignes.

```
> M:=matrix([ [0,2,4],[1,3,5] ]);
```

$$M := \begin{bmatrix} 0 & 2 & 4 \\ 1 & 3 & 5 \end{bmatrix} \quad (4.31)$$

Terminons cette section-ci avec la remarque suivante: même si cela est d'une telle évidence et que leur affichage se ressemble, une liste, un vecteur et une matrice ligne sont trois objets de nature différentes:

- une liste est une donnée
- un vecteur est une structure de données array à une dimension
- une matrice ligne est une structure de données array à deux dimensions

```
> L:=[ln(3),Francine,7,Int(x^2,x)];
T:=array(1..4,[ln(3),Francine,7,Int(x^2,x)]);
M:=matrix(1,4,[ln(3),Francine,7,Int(x^2,x)]);
```

$$L := \left[\ln(3), \text{Francine}, 7, \int x^2 dx \right]$$

$$T := \left[\ln(3) \quad \text{Francine} \quad 7 \quad \int x^2 dx \right]$$

$$M := \left[\ln(3) \quad \text{Francine} \quad 7 \quad \int x^2 dx \right] \quad (4.32)$$

Notons l'absence de virgules comme séparateurs dans le tableau à une et à deux dimensions M (une ligne par quatre colonnes).

▼ Tableau (rtable - Array)

La création d'une *structure de données* d'éléments indexés de type `rtable` est réalisée avec la macro-

commande `rtable`. Cette macro-commande (*built-in*) de la bibliothèque de base est utilisée pour la création des objets de type `Array`, `Matrix` et `Vector` (vecteur ligne et vecteur colonne). Ce sont seulement ces objets de type `rtable` qui sont compatibles avec les macro-commandes des bibliothèques de calculs vectoriels et matriciels.

Les objets `Array`, `Matrix` et `Vector` sont distincts des objets `array`, `matrix` et `vector`. La création d'objets de type `Array`, `Matrix` et `Vector` se fait de la même manière que s'ils étaient des objets de type `array`, `matrix` et `vector` sauf que leur structure de données et leur mode de stockage sont très différents. De plus, pour un usage avancé de ce mode de création, `rtable` possède une flopée de paramètres optionnels. En fait, ce qui distingue principalement les objets de type `table` d'avec les objets de type `rtable` est le mode de stockage de ces structures de données: dans le cas des objets de type `table`, le mode de stockage est le *hash table* tandis que celui des objets de type `rtable` consiste en un stockage direct utilisant des blocks d'octets de taille fixe. La distinction à faire est donc purement informatique et non pas mathématique.

En conséquence, la manière d'accéder au contenu de tels objets diffère et c'est la raison qui explique qu'il faudra créer les matrices et les vecteurs avec les macro-commandes `Matrix` et `Vector` pour les opérer ultérieurement avec des macro-commandes de bibliothèque de calculs vectoriels et matriciels.

Contrairement aux tables (`table`) et aux tableaux (`array`), `rtable` inclus par défaut (*built-in*) quelques paramètres d'initialisation système, Ces paramètres pouvant, bien sûr, être personnalisables. Ce document ne présentera pas exhaustivement l'ensemble des paramètres d'initialisation et options de la macro-commande `rtable`. Ce document se limitera seulement aux paramètres d'initialisation par défaut afin de bien montrer que ce sont des structures de données différentes de toutes les autres présentées précédemment.

Voici, la configuration par défaut de la macro-commande `rtable`.

```
> T:=rtable();
      T := Array(fill = 0, datatype = anything, storage = rectangular, order = Fortran_order) (5.1)
```

Notons que `rtable` dispose, par défaut, d'un mode de remplissage (*fill* = 0) du tableau qui sera créé. Ce mode est personnalisable.

Lorsque le format est déclaré, il y a affichage automatique du tableau à la condition que

- chaque dimension du format débute avec l'entier 1
- le tableau soit de dimension 1 ou de dimension 2

```
> T:=rtable(1..5);
      T := [ 0 0 0 0 0 ] (5.2)
```

```
> T:=rtable(1..5,1..3);
      T := [ 0 0 0
            0 0 0
            0 0 0
            0 0 0
            0 0 0 ] (5.3)
```

Ainsi, si l'indexation d'une des dimensions ne débute pas avec l'entier 1 ou si le tableau a plus de deux dimensions, il y a plutôt un affichage intrinsèque de l'objet.

```
> T:=rtable(1..5,2..4);
```

$$T := \begin{bmatrix} 0 & 0 & 0 \\ 0 & 0 & 0 \\ 0 & 0 & 0 \\ 0 & 0 & 0 \\ 0 & 0 & 0 \end{bmatrix}$$

1 .. 5 × 2 .. 4 Array

(5.4)

```
> T:=rtable(1..5,1..4,1..5);
```

$$T := \begin{bmatrix} 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 \end{bmatrix}$$

slice of 5 × 4 × 5 Array

(5.5)

Comme pour les tableaux (`array`), la saisie d'un objet créé avec la macro-commande `rtable` peut être faite avec

– une saisie *explicite* où il faut d'abord déclarer, à l'aide de l'opérateur «`_`», le nombre d'emplacements à réserver pour le tableau suivi de la liste des expressions à être assignées. *Dans le cas où les indices commencent avec l'entier 1, l'affichage du tableau est semblable à celui d'un objet de type `array`*;

```
> T:=rtable(1..4,[ln(3),Francine,7,Int(x^2,x)]);
```

$$T := \left[\ln(3) \quad \text{Francine} \quad 7 \quad \int x^2 dx \right]$$

(5.6)

– une saisie *implicite* où il suffit de donner la liste des expressions à être assignées. Dans ce cas, le nombre d'emplacements à réserver pour le tableau correspond au nombre d'expressions composant la liste. *Dans le cas d'une saisie implicite, les indices de l'indexation commencent, par défaut, avec l'entier 1 et il y a l'affichage du tableau lui-même.*

```
> T:=rtable([ln(3),Francine,7,Int(x^2,x)]);
```

$$T := \left[\ln(3) \quad \text{Francine} \quad 7 \quad \int x^2 dx \right]$$

(5.7)

Attention, l'affichage précédent ne correspond pas à un objet de type `array` mais plutôt à un objet de type `Array`.

```
> type(T,array);
```

false

(5.8)

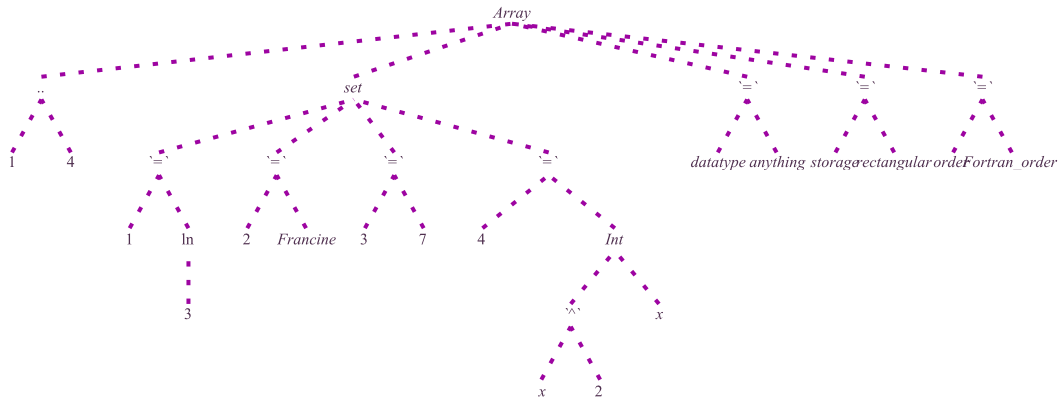
```
> type(T,Array);
```

true

(5.9)

Voyons maintenant la mémorisation par Maple de T. Contrairement à un objet de type `table`, nous n'avons pas à utiliser l'opérateur `op`.

```
> arbre(T,size=[800,300]);
```



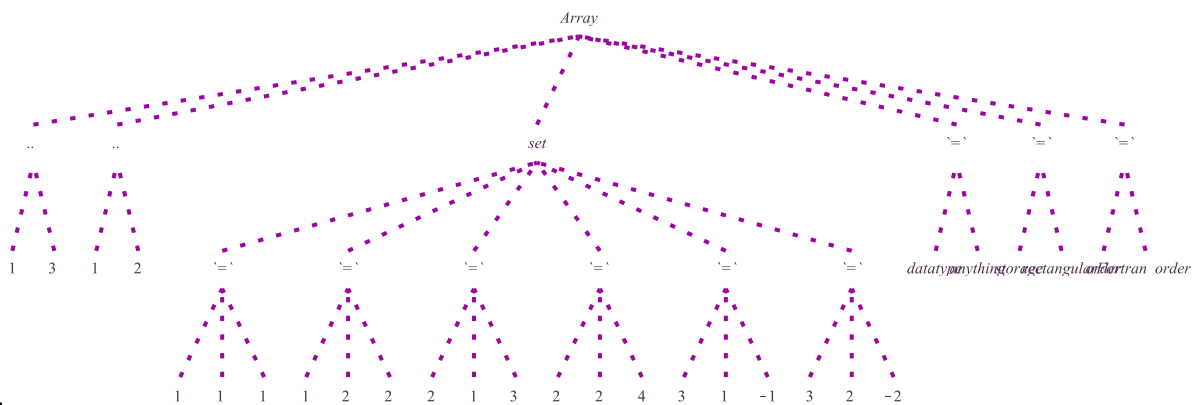
Et voici pour un tableau à deux dimensions.

```
> M:=rtable(1..3,1..2,[[1,2],[3,4],[-1,-2]]);
```

$$M := \begin{bmatrix} 1 & 2 \\ 3 & 4 \\ -1 & -2 \end{bmatrix}$$

(5.10)

```
> arbre(M,size=[1000,300]);
```



Il y a donc une grande différence que montre l'arborescence respective d'un objet de type `array` et un objet de type `Array`.

Il est possible, bien sûr, de pointer vers un élément particulier en spécifiant son indice.

```
> T[4];
M[3,2];
```

$$\int x^2 dx$$

$$-2$$

(5.11)

Il est possible de modifier une entrée ou ajouter une entrée d'un tableau existant:

- pour modifier une entrée dans un tableau (si l'option `readonly=true` n'a pas été spécifiée lors de la création de l'objet)

```

> T[2]:=Pierre;
M[2,2]:=exp(Pi);

T2 := Pierre
M2,2 := eπ

```

(5.12)

```

> T;
M;

[ ln(3)  Pierre  7  ∫x2 dx ]
[ 1      2      ]
[ 3      eπ    ]
[ -1     -2     ]

```

(5.13)

- pour ajouter une entrée dans un tableau

```

> T(6):=Catherine;

T := [ ln(3)  Pierre  7  ∫x2 dx  0  Catherine ]

```

(5.14)

Notons que l'emplacement d'indice 4 a été génériquement généré.

Par contre, pour un tableau de plus d'une dimension, nous ne pouvons étendre dynamiquement son format.

```

> M[1,3]:=sqrt(3);
Error, Array index out of range

```

On peut aussi pointer vers les indices et les entrées d'un objet de type Array.

```

> indices(T, 'nolist');
indices(M);

1, 2, 3, 4, 5, 6
[1, 1], [2, 1], [3, 1], [1, 2], [2, 2], [3, 2]

```

(5.15)

```

> entries(T, 'nolist');
entries(M, 'pairs');
rtable_elems(M)

ln(3), Pierre, 7, ∫x2 dx, 0, Catherine

(1, 1) = 1, (1, 2) = 2, (2, 1) = 3, (2, 2) = eπ, (3, 1) = -1, (3, 2) = -2
{ (1, 1) = 1, (1, 2) = 2, (2, 1) = 3, (2, 2) = eπ, (3, 1) = -1, (3, 2) = -2 }

```

(5.16)

Pour terminer ce survol succinct de la fonctionnalité `rtable`, faisons la remarque suivante quant au type d'objets créés avec `rtable` lorsqu'on spécifie, en option, l'une ou l'autres des options suivantes: `Matrix`, `Vector[row]` et `Vector[column]`. (Notez les majuscules)

Comme nous l'avons vu précédemment, les objets créés avec `rtable` sont de type `Array`. C'est le type par défaut. En spécifiant l'une ou l'autre des options au moment de la création d'un objet `rtable`, nous en modifions le type en conséquence.

```
[ > M:=rtable(1..3,1..2,[[4,3],[2,1]],subtype = Matrix);
```

$$M := \begin{bmatrix} 4 & 3 \\ 2 & 1 \\ 0 & 0 \end{bmatrix} \quad (5.17)$$

```
[ > type(M,Array);
type(M,Matrix)
false
true (5.18)
```

Attention, cette option ne permet plus de reconnaître cet objet comme du type `Array`. L'option `Matrix` (aussi avec `Vector[row]` et `Vector[column]`) sont en fait des procédures.

La macro-commande `arbre` est impuissante pour faire afficher l'arborescence d'un tel objet.

```
[ > arbre(M);
Error, (in marque) improper op or subscript selector
```

Les macro-commandes de la bibliothèque de base `Matrix`, `Vector[row]` et `Vector[column]` sont des procédures qui, en appelant `rtable`, stockent en mémoire physique ces objets ainsi que leurs caractéristiques.

```
[ > M;
```

$$\begin{bmatrix} 4 & 3 \\ 2 & 1 \\ 0 & 0 \end{bmatrix} \quad (5.19)$$

```
[ > seq(op(k,M),k=0..3);
Matrix, 3, 2, {(1, 1) = 4, (1, 2) = 3, (2, 1) = 2, (2, 2) = 1}, datatype = anything, storage
= rectangular, order = Fortran_order, shape = [ ] (5.20)
```

La macro-commande `lprint` permet de seulement d'afficher les caractéristiques dimensions, valeurs initiales et options de l'objet.

```
[ > lprint(M);
Matrix(3,2, {(1, 1) = 4, (1, 2) = 3, (2, 1) = 2, (2, 2) = 1}, datatype = anything
, storage = rectangular, order = Fortran_order, shape = [ ])
```

La requête suivante permet l'affichage de la procédure elle-même qui a permis la création de M.

```
[ > interface(verboseproc=2):  
  op(1,op(0,M)): # très longue procédure
```

En tant que procédure, nous pouvons en dessiner l'arborescence mais ce n'est pas instructif.

```
[ > arbre(op(1,op(0,M)),size=[900,300]):
```

Vous retrouvez la continuation de la présentation des `Matrix`, `Vector[row]` et `Vector[column]` dans le document *Vecteurs algébriques et matrices* sur mon site Internet dans l'onglet NYC.