

Notes de Cours du module ULIN101

Algorithmique et Maple – L1

P. Janssen J.F. Vilarem

Université Montpellier 2

Septembre 2007

La Discipline Informatique

Comme nous l'avons indiqué lors des journées d'accueil, l'*Informatique* est l'association d'une science et d'une technologie.

Une **science**, ensemble de théories, outils formels et méthodes s'intéressant à :

- la modélisation de l'information
- la résolution de problèmes à l'aide d'ordinateurs,

et une **technologie**, consistant en la conception, la réalisation et le maintien d'infrastructures matérielles et logicielles.

Sommaire

- 1 Préambule
- 2 Introduction
- 3 Langage algorithmique
 - Les Types, les Variables. Notion d'Environnement
 - Affectation et mécanisme d'évaluation
 - Structures de Contrôle et Algorithmes
 - Séquence et conditionnelle
 - Itérations
- 4 Tableaux
- 5 Algorithmique de base
 - Manipulation des types de base
 - Tableaux
- 6 Un langage de programmation : Maple
 - Introduction
 - Types de base
 - Variables, expressions et affectation
 - Algorithmes et procédures/fonctions Maple
 - Structures de contrôle en Maple
 - Tableaux en MAPLE

Positionnement du cours

- La partie scientifique de ce cours :
 - Utilise des « modèles » qui vous sont familiers : les entiers \mathbb{N} , les booléens, les vecteurs d'entiers, les rationnels \mathbb{Q} , la géométrie du plan, des calculs formels (polynômes,...), et un peu de calcul numérique. Tous ces modèles font partie du *bagage mathématique* du scientifique qui rentre à l'Université.
 - Un modèle étant choisi, on pose un problème. On cherche alors un algorithme qui décrit la suite des étapes qui permet de résoudre le problème. La *machine* utilisée reste abstraite.
- La partie technologique de ce cours utilise les machines réelles (nos ordinateurs) et un environnement logiciel (Maple) pour traduire les algorithmes en des fonctions (des programmes) Maple, qu'on exécutera.

Motivation

- L'algorithmique (discipline de conception des algorithmes) est-elle récente ?
- Les algorithmes ne traiteraient que des nombres ?

Motivation (suite et fin)

- Les algorithmes se décrivent avec un langage de programmation ?
- Les algorithmes nécessitent un ordinateur ?
- Le mot « algorithme » est un mot d'origine grecque ?

Introduction

Objectif : résolution de problèmes sur ordinateur. Cette résolution s'effectue en 2 étapes :

- Écriture de l'algorithme : méthode permettant de calculer le résultat à partir de la donnée du problème (sur une machine abstraite).
- Écriture du programme : traduction de l'algorithme pour une machine physique et un langage de programmation donnés

Définition (Algorithme)

Un algorithme est une description finie d'un calcul qui associe un résultat à des données. Il est composé de 3 parties :

- son **nom**
- sa **spécification** qui décrit quels sont les **paramètres** en entrée et quel est le **résultat** en sortie. Elle décrit le problème résolu par l'algorithme (la fonction calculée par l'algorithme).
- son **corps** décrit le calcul dans un pseudo-langage ou langage algorithme. Ce langage fournit des opérations et objets primitifs et des moyens de les composer.

Exemple

Algorithme : estPair ?

Données : $a \in \mathbb{N}$

Résultat : *true* si a est pair,
false sinon.

début

```
si (a mod 2) = 0 alors
| renvoyer true;
sinon
| renvoyer false;
fin si
```

fin algorithme

Une fois l'algorithme écrit, on l'utilise par **application à des arguments**.

Définition (Appliquer un algorithme)

- La **syntaxe** d'une application d'algorithme est celle de l'application d'une fonction en math.
- La **valeur** d'une telle application est obtenue :
 - en **substituant** dans le corps de l'algorithme les **arguments** (ici 15) aux paramètres de l'algorithme (ici a).
 - en appliquant le corps substitué de l'algorithme obtenu dans l'étape précédente ; la valeur résultat est celle donnée par l'instruction `renvoyer`

Exemple

Exécution de `estPair?(15)` :

Langage algorithmique

Dans cette section, nous étudierons :

- Définition d'un **langage algorithmique** :
 - Quels sont les éléments de base : valeurs et opérations primitives ?
 - Quelles sont les règles de composition ?
- Exactitude de l'algorithme : Comment s'assurer de l'arrêt de l'algorithme ? Comment s'assurer qu'il résout le problème spécifié ?
- Le langage de programmation qui servira à traduire et tester l'exécution de nos algorithmes sera le langage MAPLE.

Traduction dans des langages de programmation

Un algorithme peut être traduit dans plusieurs langages de programmation :

Exemple (Traduction de `estPair?` en SCHEME)

```
(define estPair? (lambda (a)
  (if (= (modulo a 2) 0) #t #f)))
```

Exemple (Traduction de `estPair?` en MAPLE)

```
estPair? := proc(a::integer)::boolean;
  description "renvoie true si a est pair, faux sinon";
  if (a mod 2)=0 then return true
  else return false
  end if;
end proc;
```

Les objets manipulés par un algorithme ont un type.

Définition (Type)

Un **type** est défini par :

- un **domaine** : l'ensemble des valeurs que peuvent prendre les objets du type
- un ensemble d'opérations qu'on peut appliquer aux objets du type.

Type symbole

- domaine : des noms (séquence de caractères)
- pas d'opération
- exemple : `estPair?`, `a`, `sin`,...

Type entiers (relatifs)

- domaine : \mathbb{Z}
- opérations binaires classiques : $\mathbb{Z} \times \mathbb{Z} \rightarrow \mathbb{Z}$, comme $+$, $*$, $-$, quo , mod , ..., utilisées en notation infixée
autres opérations binaires classiques, utilisées en notation préfixée comme max , ...
- opérations unaires classiques : $\mathbb{Z} \rightarrow \mathbb{Z}$, comme abs , ...
Attention $13 \text{ quo } 5$ renvoie 2
 $13 \text{ mod } 5$ renvoie 3

Type réels

- domaine : \mathbb{R}
- opérations binaires classiques : $\mathbb{R} \times \mathbb{R} \rightarrow \mathbb{R}$, comme $+$, $*$, $-$, $/$, ..., utilisées en notation infixée
autres opérations binaires classiques, utilisées en notation infixée comme $^$, ...
- opérations unaires classiques : $\mathbb{R} \rightarrow \mathbb{R}$, comme \log , \sin , ...

Environnement

Définition (Variable)

Une variable a un nom (un symbole), un type, et éventuellement une valeur, (qui peut varier au cours de l'exécution de l'algorithme). Une variable **doit être déclarée** par une instruction de la forme `nom : type`;

Exemples : `a : Entier;` `test : Booléen;` `b : Réel;`

Une variable déclarée n'a pas de valeur. La valeur d'une variable est définie/modifiée par une instruction **d'affectation**.

Définition (Environnement)

- On appelle **environnement** un ensemble d'associations `nom-valeur` (`symbole-valeur`).
- L'**environnement par défaut** est formé des symboles prédéfinis du langage d'algorithme qui sont conventionnellement : Les noms des opérateurs, fonctions et constantes prédéfinis (ceux des types de base)
- Un **environnement peut être enrichi ou modifié** en affectant une valeur à une variable.

Type booléens

- domaine : `Bool={ true, false }`
 - opérations
 - **et, ou** : $\text{Bool} \times \text{Bool} \rightarrow \text{Bool}$
 - **non** : $\text{Bool} \rightarrow \text{Bool}$
- dont la sémantique (la valeur) est définie dans la table :

a	b	a et b	a ou b	non(a)
true	true	true	true	false
true	false	false	true	false
false	true	false	true	true
false	false	false	false	true

On utilise également des opérateurs de comparaison dont la signature est :

$=, \neq, <, \leq, >, \geq : \mathbb{R} \times \mathbb{R} \rightarrow \text{Bool}$

D'autres sont définis sur les entiers et ont pour signature :

$=, \neq, <, \leq, >, \geq : \mathbb{Z} \times \mathbb{Z} \rightarrow \text{Bool}$

Exemple : $(2 > 3)$ a pour valeur

Expression

Avec les constantes de base, les opérations et fonctions de base, les algorithmes que nous avons définis et les variables, on construit des **expressions**.

Définition (Expression)

Une expression est **récurivement** définie par :

- un symbole de constante ou constante (**2, 17, true**)
- le nom d'une opération ou fonction de base (**+, -, log**) : `<symbole>`
- le nom d'une variable (**x, a, somme**) : `<symbole>`
- le nom d'un algorithme (**estPair ?**) : `<symbole>`
- l'application d'une opération de base (**2 + 3**) : `(<expression> <symbole> <expression>)`
- l'application d'une fonction ou d'un algorithme (**log(5), estPair ?(7)**) : `<symbole>(<expression>, ..., <expression>)`

Exemple

a, x, y et som étant des noms de variable :

- x
- (6 + (5 * 3))
- max((15-a), som)
- (true et false)
- ((3 < 8) et ((1 + a) = 7))
- estPair?(max((15-a), som))
- estPair?(5, 6)
- (5 + estPair?(7))

sont

La valeur Val(<exp>) d'une expression <exp> dans un environnement Env est définie récursivement par :

Au cas où <exp> est une constante	Val(<exp>) dans Env renvoie la constante
un symbole nom de variable	Renvoyer la valeur de la variable dans Env
l'application d'une opération : <exp1> op <exp2>	Évaluer v1 = Val(<exp1>) et v2 = Val(<exp2>). Renvoyer l'application de op aux opérandes v1 et v2
l'application d'une fonction : f (<exp1>, ... , <expn>)	Évaluer dans l'environnement Env v1 = Val(<exp1>), ..., vn = Val(<expn>). Renvoyer l'application de f aux arguments v1, ..., vn

Au cas où <exp> est	Val(<exp>) dans Env renvoie
l'application d'un algorithme : algo (<exp1>, ... , <expn>)	Évaluer dans l'environnement Env v1 = Val(<exp1>), ..., vn = Val(<expn>). algo a pour paramètres x1, ..., xn et un corps C. Remplacer dans C chaque paramètre par la valeur calculée précédemment. La valeur renvoyée est celle du corps ainsi substitué.
Autre cas	Renvoyer <i>Erreur</i> . Par exemple quand le nombre de paramètres de la fonction ou de l'algorithme ne correspond pas au nombre d'arguments. Ou bien quand le type d'un argument n'est pas celui du paramètre. Ou bien si le symbole nom de variable est celui d'une variable qui n'a pas de valeur dans Env

Exemple

En supposant que dans l'environnement courant :

- les variables ont la valeur

a	som	y
5	9	true

,
- la variable x n'est pas déclarée,
- l'algorithme estPair? est défini comme dans l'exemple initial,

alors :

- Val((6 + (5 * 3))) renvoie
- Val(max((15-a), som)) renvoie
- Val((true et false)) renvoie
- Val(((3 < 8) et ((1 + a) = 7))) renvoie
- Val((y et estPair?((1 + a)))) renvoie
- Val((1+x)) renvoie
- Val((1+y)) renvoie
- Val(estPair?(2, 3, a)) renvoie

Affectation

Définition (Affectation)

La syntaxe d'une affectation est
`nomVariable := expression`

Définition (Effets d'une affectation)

Les actions réalisées lors de son exécution sont :

- 1 On évalue `expression` dans l'environnement courant.
- 2 Si cette évaluation renvoie `Erreur`, l'affectation n'est pas exécutée, l'exécution générale se termine (il faut éviter ce genre de situation !). Sinon, soit `E` la valeur `Val(expression)`
- 3 Si la variable n'a pas été déclarée, l'affectation n'est pas exécutée, l'exécution générale se termine (il faut éviter ce genre de situation !).
- 4 Si la variable a été déclarée d'un type différent du type de `E`, l'exécution générale se termine (il faut éviter ce genre de situation !).
- 5 Sinon, l'environnement est modifié : la valeur de la variable devient `E`.

Remarque

On considèrera que l'ordre d'évaluation des sous-expressions d'une expression n'a pas d'importance (`Val(a + b) = Val(b + a)`).

Exception pour les opérateurs booléens `et` et `ou` :

- Lors de l'évaluation de l'expression `(a et b)`,
 - on calcule `Val(a)`
 - Si `Val(a) = false` alors `Val((a et b)) = false` (**on n'évalue pas b**)
 - Si `Val(a) = true` on calcule `Val(b)` ; `Val((a et b)) = Val(b)`
- Lors de l'évaluation de l'expression `(a ou b)`,
 - on calcule `Val(a)`
 - Si `Val(a) = true` alors `Val((a ou b)) = true` (**on n'évalue pas b**)
 - Si `Val(a) = false` on calcule `Val(b)` ; `Val((a ou b)) = Val(b)`

Conséquence : `(a et b)` et `(b et a)` n'ont pas nécessairement la même valeur

Exemple

Val(a)	Val((a>0) et (log(a)<20))	Val((log(a)<20) et (a>0))
2		
0		

Exemple

`a, b` sont deux variables déclarées de type `Entier`.

Env avant		Affectation	Env après	
Val(a)	Val(b)		Val(a)	Val(b)
×	10	<code>a := 4</code>		
2	10	<code>a := (a + 1)</code>		
5	10	<code>a := (b+2)</code>		
×	10	<code>a := (a+1)</code>		
×	10	<code>a := (b+2)</code>		
2	10	<code>a := ((a+2)*b)</code>		
2	10	<code>a := ((a et 2)*b)</code>		
2	10	<code>a := (a < b)</code>		

Algorithmes

Définition (Algorithme)

Un algorithme est composé :

- d'un nom (un symbole)
 - d'une **spécification** composée
 - d'une partie *Données* : noms, types des paramètres, conditions qu'ils vérifient
 - d'une partie *Résultat* : valeur calculée (renvoyée) par l'algorithme, définie en fonction des paramètres
 - d'un **corps** composé
 - d'une partie *déclaration de variable* : noms et types de toutes les variables utilisées par l'algorithme (éventuellement vide)
 - d'une partie *instructions* : instructions réalisées par l'algorithme pour calculer le résultat. Parmi les instructions doit figurer l'instruction `renvoyer exp`. Le résultat calculé par l'algorithme est la valeur de l'expression `exp` dans l'environnement courant.
- Attention : on s'interdit de modifier les paramètres par une affectation (penser à la substitution)**

D'où le schéma :

Schéma d'algorithme

Algorithme : nom de l'algorithme
Données : description des paramètres
Résultat : description du résultat
Déclaration des variables;
début
| Partie instructions
fin algorithme

Le corps d'un algorithme est une instruction. L'instruction de base est l'affectation. On peut composer ces instructions pour définir de nouvelles instructions. Il existe plusieurs types de composition, appelés *structures de contrôle*.

Définition (La séquence)

La **séquence d'instructions** $Inst1, Inst2, \dots, Instn$ s'écrit
 $Inst1; Inst2; \dots; Instn$.
L'exécution de cette instruction a pour effet d'exécuter $Inst1$, puis $Inst2$, ..., enfin $Instn$.

Définition (Les instructions conditionnelles)

Si simple

si $Cond$ **alors**
| $Inst$
fin si
où $Cond$ est une expression à valeur booléenne
lors de son exécution, l'expression $Cond$ est évaluée. Si $Val(Cond) = true$,
l'instruction $Inst$ est exécutée, sinon rien n'est exécuté

Si alors sinon

si $Cond$ **alors**
| $Inst1$
sinon
| $Inst2$
fin si
où $Cond$ est une expression à valeur booléenne
lors de son exécution, l'expression $Cond$ est évaluée. Si $Val(Cond) = true$,
l'instruction $Inst1$ est exécutée, sinon l'instruction $Inst2$ est exécutée

Définition (Conditionnelle suite)

Si alors sinonsi

si $Cond1$ **alors**
| $Inst1$
sinon si $Cond2$ **alors**
| $Inst2$
sinon
| $Inst3$
fin si
où $Cond1$ et $Cond2$ sont des expressions à valeur booléenne
lors de son exécution, l'expression $Cond1$ est évaluée. Si
 $Val(Cond1) = true$, l'instruction $Inst1$ est exécutée, sinon, l'expression
 $Cond2$ est évaluée. Si $Val(Cond2) = true$, l'instruction $Inst2$ est exécutée,
sinon l'instruction $Inst3$ est exécutée.

Exemples de Si Alors Sinon

Exemple (Si Alors simple)

Algorithme : estPair ?
Données : $a \in \mathbb{N}$
Résultat : true si a est pair, false sinon.
début
 si $(a \bmod 2) = 0$ **alors**
 renvoyer true;
 sinon
 renvoyer false;
 fin si
fin algorithme

Exemple (Un booléen équivalent)

Algorithme : estPair ?
Données : $a \in \mathbb{N}$
Résultat : true si a est pair, false sinon.
début
 renvoyer $((a \bmod 2) = 0)$;
fin algorithme

Exemple (Si Alors Sinon Si)

Algorithme : Médian
Données : $a, b, c \in \mathbb{Z}$
Résultat : L'élément médian de a, b, c
début
 si $a > b$ **alors**
 si $a > c$ **alors**
 renvoyer $\max(b, c)$;
 sinon
 renvoyer ? ? ? ? ?;
 fin si
 sinon si $b > c$ **alors**
 renvoyer ? ? ? ? ?;
 sinon
 renvoyer ? ? ? ? ?;
 fin si
fin algorithme

Exemples initiaux

Algorithme : MultiplierPar4
Données : $n \in \mathbb{N}$
Résultat : $4 \cdot n$
 S : Entier ;
1 **début**
2 $S := 0$;
3 **Iterer** 4 fois
4 $S := S + n$
5 **fin itérer**
6 renvoyer S ;
7 **fin algorithme**

Exécution de
MultiplierPar4(3)
En fin de ligne Itération Val(S)

Itération Pour – version simple

Définition (Itération Pour – version simple)

K étant une variable de type entier déclarée, $E2$ une expression à valeur entière supérieure ou égale à 1, l'instruction répétitive **Pour** s'écrit :

pour K **de** 1 **à** $E2$ **faire**
 Inst
finpour

Exécution d'une Itération Pour version simple

- Soit $V2$ la valeur de $E2$.
- Exécuter l'affectation $K := 1$
- Ensuite :
 - 1 Soit V_K la valeur de K Si $V_K > V2$ l'itération s'arrête
 - 2 Sinon ($V_K \leq V2$) exécuter l'instruction Inst puis exécuter l'affectation $K := K + 1$ puis recommencer en 1

Algorithme : MultiplierPar4**Données :** $n \in \mathbb{N}$ **Résultat :** $4 \cdot n$ S, K : Entier ;

```

1 début
2   S := 0;
3   pour K de 1 à 4 faire
4     | S := S + n
   finpour
5   renvoyer S;
fin algorithme

```

Exécution de**MultiplierPar4(3)**

En fin de ligne Val(K) Val(S)

Algorithme : Multiplier**Données :** $a, b \in \mathbb{N}$ **Résultat :** $a \cdot b$ S, K : Entier ;

```

1 début
2   S := 0;
3   pour K de 1 à b faire
4     | S := S + a
   finpour
5   renvoyer S;
fin algorithme

```

Exécution de Multiplier(2, 3)Il y a d'abord substitution : a est remplacé par 2, et b par 3 dans le corps, ce qui donne :

```

1 début
2   S := 0;
3   pour K de 1 à 3 faire
4     | S := S + 2
   finpour
5   renvoyer S;
fin algorithme

```

```

1 début
2   S := 0;
3   pour K de 1 à 3 faire
4     | S := S + 2
   finpour
5   renvoyer S;
fin algorithme

```

Exécution de Multiplier(2, 3)

En fin de ligne Val(K) Val(S)

Itération Pour

Définition (Itération pour)

 K étant une variable de type entier déclarée, $E1$, $E2$, $E3$ 3 expressions à valeur entière, l'instruction répétitive **Pour** s'écrit :

```

pour K de  $E1$  à  $E2$  par pas de  $E3$  faire
  | Inst
finpour

```

Exécution d'une itération Pour

- Évaluer les expressions $E1, E2, E3$.
Soient $v1, v2, v3$ leurs valeurs respectives.
- Exécuter l'affectation $K := v1$
- $v3$ est supposée positive.
 - 1 Évaluer K . Soit v_K sa valeur. Si $v_K > v2$ l'itération s'arrête
 - 2 Sinon ($v_K \leq v2$) exécuter l'instruction **Inst**
puis exécuter l'affectation $K := K + v3$
puis recommencer en 1

Remarque

- La partie `par pas` de `E3` est optionnelle. Si elle est omise `E3` vaut 1.
- `E1`, `E2`, `E3` sont évaluées une fois pour toute avant l'itération : le corps de l'itération (l'instruction `Inst`) **ne peut pas modifier** leur valeur.
- le corps de l'itération **ne peut pas modifier** la valeur de la variable `K`
- en sortie de l'itération `Pour` la variable de contrôle `K` n'a pas de valeur.

Algorithme : SommeCarrés

Données : $n \in \mathbb{N}$

Résultat : $\sum_{i=1}^n i^2$

`K` : Entier ;

`S` : Entier ;

```
1 début
2   S := 0;
3   pour K de 1 à n faire
4     S := S + K*K
5   finpour
6   renvoyer S;
7 fin algorithme
```

Exécution de `SommeCarrés(3)`

En fin de ligne `Val(K)` `Val(S)`

Définition (Itération tant que)

L'instruction répétitive `Tant que` s'écrit :

tant que `Cond` **faire**

`Inst`

fin tq

où `Cond` est une expression à valeur booléenne

L'exécution d'une itération `Tant que` revient à :

- 1 évaluer `Cond`. Soit `B` cette valeur.
- 2 si `B` est `false` l'itération s'arrête
- 3 sinon exécuter l'instruction `Inst` et recommencer en 1

Remarque

- à la différence de la répétitive `Pour`, l'instruction `Inst` doit modifier la valeur de l'expression `Cond`
- Contrairement à l'itération `Pour`, le nombre d'itérations de l'itération `Tant que` dépend de l'instruction itérée

On se pose la question : le nombre entier positif `p` est-il une puissance de 2 ?

```
2 p := xxxx ;
3 tant que estPair?(p) faire
4   p := p / 2 ;
5 fin tq
6 renvoyer ??? ;
```

Exécution, `xxxxx` remplacé par 24

En fin de `Val(p)` `Val(estPair?(p))`
ligne

Que faut-il renvoyer, et pourquoi ?

On se pose la question : le nombre entier positif p est-il une puissance de 2 ?

```

2 p := xxxx ;
3 tant que estPair?(p) faire
4   | p := p / 2 ;
5   fin tq
5 renvoyer ????? ;

```

Exécution, xxxx remplacé par 16

En fin de Val(p) Val(estPair?(p))
ligne

Que faut-il renvoyer, et pourquoi ?

Algorithme : PuissanceDe2 ?

Données : $n \in \mathbb{N}^*$

Résultat : True si n est une puissance de 2,
False sinon

```

p : entier;
1 début
2   p := n;
3   tant que (p mod 2) = 0
4     faire
5       | p := p / 2 ;
6     fin tq
7   renvoyer ( p = 1 );
8 fin algorithme

```

Exécution de PuissanceDe2 ? (6)

En fin de Val(p) Val((p mod 2)=0)
ligne

Pourquoi ne pas opérer ces divisions par 2 directement sur n ?
Pourquoi l'instruction de la ligne 2 est nécessaire ?

Algorithme : PuissanceDe2 ?

Données : $n \in \mathbb{N}^*$

Résultat : True si n est une puissance de 2,
False sinon

```

p : entier;
1 début
2   p := n;
3   tant que (p mod 2) = 0
4     faire
5       | p := p / 2 ;
6     fin tq
7   renvoyer ( p = 1 );
8 fin algorithme

```


Exécution de PuissanceDe2 ? (8)

En fin de Val(p) Val((p mod 2)=0)
ligne

Tableaux

Un tableau est une structuration de données analogue aux vecteurs des mathématiques, par exemple 8 réels regroupés en un tableau de nom T .

T	5.4	7.2	0.8	-3.5	2.5	8.6	2.5	7.0
-----	-----	-----	-----	------	-----	-----	-----	-----

$T[4]$ 

Définition (Tableau)

Une variable tableau a un nom, une taille entière non nulle et un type commun à tous ses éléments.

On déclare une variable tableau de nom T , de taille 8, et dont les éléments sont de type `Réel` par l'instruction : `T : array 1..8 of Réel ;`
 T structure 8 variables de type `Réel`. On accède à ces variables par leur indice i qui est un entier entre 1 et 8. La notation est $T[i]$. On appelle ces variables les **éléments** de T .

La fonction **taille** : `Tableau` $\rightarrow \mathbb{N}^*$ renvoie la taille entière non nulle de son argument.

Définition (Initialisation de tableau)

La déclaration d'un tableau ne fournit pas de valeur pour chacun de ses éléments. Ils doivent être **initialisés** par une instruction d'affectation.

Exemple

```
T : array 1..8 of Réel;
T[1] := 5.4 ;
T[2] := 7.2 ;
T[3] := 0.8 ;
a T[8] := 7.0 ;
```

À la fin de l'instruction a, le tableau T est partiellement initialisé et a pour valeur :

T	5.4	7.2	0.8					7.0
---	-----	-----	-----	--	--	--	--	-----

T[2] a pour valeur 7.2 alors que T[4] renvoie **Erreur** comme une variable simple qui a été déclarée et non initialisée.

Algorithme : f

Données : T un tableau d'entiers

Résultat : la somme des éléments de T

somme, i : Entier;

début

 somme := 0;

pour i de 1 à taille(T)

faire

 somme := somme + T[i]

finpour

renvoyer somme

fin algorithme

S : array 1..4 of Entier;

x : Entier ;

S[1] := 1 ; S[2] := 3 ;

a S[3] := 7 ; S[4] := 7 ;

...

b x := f(S) ;

...

À la fin de l'instruction a le tableau S est entièrement initialisé et a pour valeur :

À la fin de l'instruction b la variable x a pour valeur : .

Tableau comme résultat d'un algorithme

Algorithme : f

Données : N un entier positif

Résultat : Le tableau de taille N dont les éléments sont les N premiers entiers non nuls

i : Entier ;

T : array 1..N of Entier ;

début

pour i de 1 à N **faire**

 T[i] := i

finpour

renvoyer T

fin algorithme

S : array 1..4 of Entier;

a S := f(4) ;

...

À la fin de l'instruction a le tableau S est entièrement initialisé et a pour valeur :

Attention, c'est un cas où on s'autorise l'affectation d'un tableau à une variable. Mais on ne peut exécuter d'instruction comme S := T, où S, T sont des tableaux pour initialiser S

Un algorithme peut avoir en donnée un tableau, et renvoyer un –autre– tableau comme résultat. C'est l'autre cas où on s'autorisera à affecter globalement un tableau à une variable.

Algorithme : g

Données : T un tableau d'entiers

Résultat : Le tableau de même taille que T et dont les éléments sont les doubles des éléments de T

i : Entier ;

S : array 1..taille(T) of Entier ;

début

pour i de 1 à taille(T)

faire

 S[i] := 2 * T[i]

finpour

renvoyer S

fin algorithme

T1, T2 : array 1..4 of Entier;

a T1 := f(4) ;

...

b T2 := g(T1) ;

À la fin de l'instruction a le tableau T1 est entièrement initialisé et a pour valeur :

À la fin de l'instruction b le tableau T2 est entièrement initialisé et a pour valeur :

Algorithmes de base

Problème : Multiplier l'entier naturel a par l'entier naturel b . Seule l'addition est autorisée.

Solution naïve : Calculer $a+a+\dots+a$, le tout b fois.

Algorithme : MulAdd

Données : $a, b \in \mathbb{N}$.

Résultat : Le produit ab

i, r : Entier ;

début

```
1  /* Seule l'addition est autorisée. */
2  r := 0 ;
3  pour i de 1 à b faire
4  | r := r + a ;
5  finpour
6  renvoyer r
7  fin algorithme
```

Exécution de MulAdd(5, 3)

En fin de ligne Val(r) Val(i)

Algorithme : MulAdd2

Données : $a, b \in \mathbb{N}$.

Résultat : Le produit ab

i, r : Entier ;

début

```
1  /* Seule l'addition est autorisée. */
2  r := 0 ; i := 0 ;
3  tant que i <> b faire
4  | /* r = a * i */
5  | i := i + 1 ; r := r + a ;
6  fin tq
7  renvoyer r
8  fin algorithme
```

Exécution de MulAdd2(5, 3)

En fin de ligne Val(r) Val(i)

Remarque

$r = a * i$ est la **propriété invariante** de l'algorithme, à partir de la ligne 1.

PGCD de 2 entiers

Définition (Rappels)

Soient $a, b \in \mathbb{N}^*$. Il existe de manière unique q, r tels que $a = bq + r$ avec $0 \leq r < b$.

q, r sont appelés le quotient et le reste de la division euclidienne de a par b .

On a deux opérations qui les renvoient :

- $a \bmod b$ renvoie le reste r
- $a \text{ quo } b$ renvoie le quotient q

a **divise** b si $a \bmod b = 0$

tout nombre a au moins pour diviseurs : 1 et lui même

dans l'ensemble des diviseurs **communs** à a et b , il y a au moins 1

le plus grand de ces diviseurs communs est appelé leur **PGCD** noté

$\text{pgcd}(a, b)$

Exemple

Les diviseurs de 12 sont 1,2,3,4,6,12

Les diviseurs de 30 sont 1,2,3,5,6,10,15,30

$\text{pgcd}(12, 30)$ est 6

Même problème de multiplication par additions successives. Version de l'algorithme précédent qui utilise l'itération `tant que`.

Propriété (PGCD)

Le **plus grand commun diviseur** ou **pgcd**, de a, b est aussi le pgcd de b, r .

En fait x divise a et b si et seulement si x divise b et r .

L'idée de l'algorithme consiste à remplacer a, b par b, r pour rechercher le pgcd, et à itérer le procédé jusqu'à ce que

Algorithme : pgcd

Données : $a, b \in \mathbb{N}$. Peu importe que $a \geq b$ ou non.

Résultat : Le pgcd de a et b
 x, y, r : Entier ;

début

```

1  x := a ; y := b ;
2  r := a mod b ;
3  tant que r ≠ 0 faire
4  |   x := y ;
   |   y := r ;
   |   r := x mod y ;
   fin tq
renvoyer y ;
fin algorithme

```

Exécution de pgcd(25, 9)

ligne	Val(x)	Val(y)	Val(r)	Val($r \neq 0$)
1	25	9	25 mod 9 = 7	1
2	9	7	9 mod 7 = 2	1
3	7	2	7 mod 2 = 1	1
4	2	1	2 mod 1 = 0	0

Algorithme : pgcd

Données : $a, b \in \mathbb{N}$. Peu importe que $a \geq b$ ou non.

Résultat : Le pgcd de a et b
 x, y, q, r : Entier ;

début

```

1  x := a ; y := b ;
2  r := a mod b ;
3  tant que r ≠ 0 faire
4  |   x := y ;
   |   y := r ;
   |   r := x mod y ;
   fin tq
renvoyer y ;
fin algorithme

```

Exécution de pgcd(9, 25)

ligne	Val(x)	Val(y)	Val(r)	Val($r \neq 0$)
1	9	25	9 mod 25 = 9	1
2	25	9	25 mod 9 = 7	1
3	9	7	9 mod 7 = 2	1
4	7	2	7 mod 2 = 1	1
5	2	1	2 mod 1 = 0	0

Erreur à ne pas commettre

Algorithme avec erreur

Algorithme : pgcd-erreur

Données : $a, b \in \mathbb{N}$.

Résultat : Le pgcd de a et b
 q, r : Entier ;

début

```

1  r := a mod b ;
2  tant que r ≠ 0 faire
3  |   a := b ;
   |   b := r ;
   |   r := a mod b ;
   fin tq
renvoyer b ;
fin algorithme

```

Attention, on n'a pas le droit de modifier les paramètres donnés de l'algorithme. Voyez-vous pourquoi ?

Minimum d'un tableau d'entiers

Algorithme : MinTableau

Données : Un tableau d'entiers
initialisé T .

Résultat : L'entier minimum des éléments de T

m, i : Entier ;

début

```

1  m := T[1] ;
2  pour i de 2 à taille(T) faire
3  |   si T[i] < m alors
4  |   |   m := T[i] ;
   |   fin si
   finpour
renvoyer m ;
fin algorithme

```

Soit le tableau :

S	7	9	6	6
---	---	---	---	---

Exécution de MinTableau(S)

ligne	Val(i)	Val(m)	Val($T[i] < m$)
1	1	7	0
2	2	7	0
3	3	6	1
4	4	6	0

Un algorithme faux

Algorithme avec **erreur**

Algorithme : MinTableauErreur

Données : Un tableau d'entiers **initialisé** T.

Résultat : L'entier minimum des éléments de T

m, i : Entier ;

début

```

1  m := 0 ;
2  pour i de 1 à taille(T)
3    faire
4      si T[i] < m alors
5        m := T[i] ;
      fin si
    finpour
  renvoyer m ;
fin algorithme

```

Soit le tableau :

S	7	9	6	6
---	---	---	---	---

Exécution de

MinTableauErreur(S)

ligne	Val(i)	Val(m)	Val(T[i] < m)
1	7	0	faux
2	9	0	faux
3	6	6	vérité
4	6	6	faux

Un algorithme faux

Algorithme avec **erreur**

Algorithme : MinTableauErreur2

Données : Un tableau d'entiers **initialisé** T.

Résultat : L'entier minimum des éléments de T

m, i : Entier ;

début

```

1  m := T[1] ;
2  pour i de 2 à taille(T)
3    faire
4      si T[i] < m alors
5        renvoyer T[i] ;
      fin si
    finpour
  renvoyer m ;
fin algorithme

```

Soit le tableau :

S3	7	6	4	5
----	---	---	---	---

Exécution de

MinTableauErreur2(S3)

ligne	Val(i)	Val(m)	Val(T[i] < m)
1	7	7	faux
2	6	6	vérité
3	4	4	vérité
4	5	4	faux

Recherche d'un élément dans un tableau

Algorithme : ChercherTableau

Données : Un entier x, un tableau d'entiers **initialisé** T.

Résultat : Le booléen qui indique si x est élément de T

i : Entier ;

début

```

1  pour i de 1 à taille(T)
2    faire
3      si T[i] = x alors
4        renvoyer ????? ;
      fin si
    finpour
  renvoyer ????? ;
fin algorithme

```

Soit le tableau :

S	7	9	6	6
---	---	---	---	---

Exécution de

ChercherTableau(6, S)

ligne	Val(i)	Val(T[i] = x)
1	7	faux
2	9	faux
3	6	vérité
4	6	vérité

Recherche d'un élément dans un tableau

Algorithme : ChercherTableau

Données : Un entier x, un tableau d'entiers **initialisé** T.

Résultat : Le booléen qui indique si x est élément de T

i : Entier ;

début

```

1  pour i de 1 à taille(T)
2    faire
3      si T[i] = x alors
4        renvoyer ????? ;
      fin si
    finpour
  renvoyer ????? ;
fin algorithme

```

Soit le tableau :

S	7	9	6	6
---	---	---	---	---

Exécution de

ChercherTableau(5, S)

ligne	Val(i)	Val(T[i] = x)
1	7	faux
2	9	faux
3	6	faux
4	6	faux

Algorithme : ChercherTableauFaux

Données : Un entier x , un tableau d'entiers **initialisé** T .

Résultat : Le booléen qui indique si x est élément de T

```

i : Entier ;
0 début
1   pour i de 1 à taille(T)
2     faire
3       si T[i] = x alors
4         renvoyer true ;
        sinon
          renvoyer false ;
        fin si
      finpour
    fin algorithme
  
```

Soit le tableau :

S	7	9	6	6
---	---	---	---	---

Exécution de

ChercherTableauFaux(3, S)
 ligne Val(i) Val(T[i] = x)

Jusque là, ça va ?

Algorithme : ChercherTableauFaux

Données : Un entier x , un tableau d'entiers **initialisé** T .

Résultat : Le booléen qui indique si x est élément de T

```

i : Entier ;
0 début
1   pour i de 1 à taille(T)
2     faire
3       si T[i] = x alors
4         renvoyer true ;
        sinon
          renvoyer false ;
        fin si
      finpour
    fin algorithme
  
```

Soit le tableau :

S	7	9	6	6
---	---	---	---	---

Exécution de

ChercherTableauFaux(6, S)
 ligne Val(i) Val(T[i] = x)

Et maintenant, ça va toujours ?

Définition (Occurrence dans un tableau)

Une **occurrence** d'une valeur v dans un tableau T est associée à un indice i tel que $T[i] = v$.

Exemple

S	7	9	6	6
---	---	---	---	---

- Dans le tableau S l'entier 6 a 2 occurrences. Elles sont associées aux indices 3 et 4 du tableau S .
- Dans S l'entier 9 a une seule occurrence.
- Il n'y a pas d'occurrences de 5 dans le tableau S

Algorithme : NbOccTableau

Données : Un entier v , Un tableau d'entiers **initialisé** T .

Résultat : Le nombre d'occurrences de v dans T

```

nbocc, i : Entier ;
0 début
1   nbocc := 0 ;
2   pour i de 1 à taille(T)
3     faire
4       si T[i] = v alors
5         nbocc := nbocc + 1 ;
        fin si
      finpour
    renvoyer nbocc ;
  fin algorithme
  
```

Soit le tableau :

S	7	9	6	6
---	---	---	---	---

NbOccTableau(6, S)

ligne Val(i) Val(nbocc) Val(T[i] = v)

Motivations

- **Pourquoi un langage de programmation ?** On écrit nos algorithmes dans un langage abstrait et indépendamment de toute « machine ». On veut ensuite tester/exécuter. Il nous faut donc un environnement d'exécution, composé d'une machine réelle, un langage de programmation et les **règles de traduction** qui permettent de passer du langage algorithmique au langage de programmation.
- **Pourquoi ce langage de programmation (Maple) ?** Honnêtement, c'est un très mauvais langage de programmation. Mais Maple n'est pas fait pour celà. Maple est un remarquable environnement de prototypage pour faire du calcul symbolique. Nous allons donc faire des compromis et diverses contorsions pour traduire les structures de notre langage.
- **Pourquoi ne pas écrire directement les algorithmes en Maple ?** Le langage d'algorithme s'affranchit des contraintes des langages (absence de type en Scheme, typage fort en Caml, modèle particulier pour l'affectation en Maple ...). Dans un langage d'algorithme on définit précisément la méthode, et les données employées pour résoudre un problème.

Les types de base en Maple

Pas question d'étudier Maple de façon exhaustive. Par exemple Maple possède **plus d'une centaine de types** de données.

Les entiers relatifs \mathbb{Z}

- Nom du type : `integer`
- Représentation du type : avec une taille bornée, mais grande (vraiment grande) et système dépendante, $268\,435\,448 = 2^{28} - 8$ chiffres sur ma machine.
- Opérations/fonctions : Classiques et nombreuses. `+`, `*`, `-`, `mod` en notation infixée, d'autres comme `abs`, `iquo`, `irem` ... en notation préfixée, enfin ! en notation suffixée (la factorielle). On ne fera pas le tour de la vaste bibliothèque des fonctions du langage.

Les fractions rationnelles : \mathbb{Q}

- Nom du type : `fraction`
- Pas un type de base ! MAPLE fait du calcul formel en opérant sur des classes d'équivalences définies sur $\mathbb{Z} \times \mathbb{N}^*$.
Exactement comme en Mathématiques, $15/9$ et $10/6$ sont deux éléments de la même classe, dont le représentant canonique est $5/3$.
Attention, $5/3$ n'est pas le flottant `evalf(5/3)` qui est $1.666666\dots$.
Attention encore, $6/3$ est un entier !
Attention enfin, les opérations dans \mathbb{Q} se font en « précision infinie », au sens, avec tous les chiffres permettant de représenter des entiers.
`2^200/100 ! ; renvoie`
`10141204801825835211973625643008/588971222367687651371`
`627846346807888288472382883312574253249804256440585603`
`406374176100610302040933304083276457607746124267578125 ;`
- Les opérations du « type » `fraction` sont classiques `+`, `-`, `*`, `/`, `^`

Les réels \mathbb{R}

- Nom du type : `float`
- La représentation est classique : une *mantisse* et un *exposant* qui sont des entiers qui ont chacun une taille bornée, système dépendante. (268 435 448 chiffres pour la mantisse et 2 147 483 646 pour l'exposant). La base de la représentation est 10.
Exemple : `41.87` est un flottant dont la mantisse est `4187` et l'exposant `-2`. On le note aussi `0.4187e2`.
Attention le domaine est une partie finie des décimaux ! Ce n'est évidemment pas un intervalle de \mathbb{R} .
- Les opérations sont classiques : `+`, `-`, `*`, `/`, `^`, ..., `log`, `sin`, `exp`, ...
- Des fonctions opèrent d'un type numérique à l'autre :
 - `evalf` transforme son argument de type `integer` ou `fraction` en `float`.
Exemple : `evalf(5)` ; renvoie `5.`
 - `floor`, `ceil`, `trunc`, `frac`, `round` transforment leur argument flottant en un entier : partie entière classique, par valeur supérieure, troncature, partie fractionnaire, arrondi.
Exemple : `floor(-6.789)` ; renvoie `-7`

Les chaînes de caractères

- Nom du type : `string`
- Ce sont des suites d'au maximum 268 435 439 caractères, entourées de guillemets `"`. Le caractère guillemet est obtenu par `\`.
Exemple : `cat("Mon nom est ", "Personne".)` renvoie la chaîne de caractères `"Mon nom est "Personne".` qui a pour longueur 23
- Les fonctions du type `string` sont `cat` la concaténation, `substring` l'extraction d'une sous-chaîne, `length` qui renvoie l'entier longueur de son argument.

La traduction est assez directe :

Traduction

Langage d'algorithme	Traduction en Maple
<code>variable : type;</code>	<code>variable ::type;</code>
<code>variable := expression;</code>	<code>variable := expression;</code>
Exemples	
<code>a, b : entier;</code>	<code>a::integer; b::integer;</code>
<code>superieur : boolean;</code>	<code>superieur :: boolean;</code>
<code>a := 3; b := 4;</code>	<code>a := 3; b := 4;</code>
<code>superieur := (b > 9);</code>	<code>superieur := evalb(b > 9);</code>

Remarque (Particularités de Maple)

- Les symboles en Maple différencient majuscule et minuscule. `Trouve` n'est pas `trouve`. Certains noms sont réservés par MAPLE car alloués à des primitives du langage : `Pi`, `I`, `D`, ...
- Par défaut d'affectation initiale, toute variable est initialisée automatiquement par MAPLE à la constante symbolique qu'est son nom. Ceci permet de faire du calcul symbolique, mais sort du cadre de ce cours.

Les booléens

- Nom du type : `boolean`
- Les valeurs du type sont `true` et `false`
- Les opérations sont `and`, `or`, `not`. Les règles d'évaluation sont celles vues dans le cours d'algorithmique.
- Les opérations qui renvoient un booléen sont, comme dans le cours d'algorithmique `=`, `<`, `>`, `<>`, `<=`, `>=`.
Attention : Maple utilise les opérations de comparaison pour faire du calcul formel. Il évalue donc une expression comme `(2 < 3)` :
 - Un booléen normal, dans le contexte des conditions des structures de contrôle (`si (2 < 3) .. ?`)
 - Une égalité, inégalité en dehors de ce contexte. Dans ce cas, on forcera l'évaluation comme `boolean` en utilisant la fonction `evalb`.
`trouve := evalb(2 < 3);`

Exemple

Après `restart`; `a:=1` ; `b:=2` ; `a := a + b`;

- Maple est un interpréteur. L'environnement peut être remis à sa valeur initiale par la commande `restart` ;
- Par défaut de déclaration, une affectation `var := expression` donne le type de `Val(expression)` à la variable `var`
- Par défaut d'affectation initiale, toute variable est initialisée automatiquement par MAPLE à la constante symbolique qu'est son nom. Ceci permet de faire du calcul symbolique, mais sort du cadre de ce cours.
- l'expression `x + a` a pour valeur `x + 3`, dans laquelle `x` est la valeur de la constante symbolique `'x'`.
- l'expression `a*x^2 + c*x + d` a pour valeur `3x^2 + cx + d`, dans laquelle `x`, `c`, `d` sont les valeurs des constantes symboliques `'x'`, `'c'`, `'d'`.
- l'expression `f(a, x)` a pour valeur `f(3, x)` dans laquelle `x`, `f` sont les valeurs des constantes symboliques `'x'`, `'f'`.

Tout ce qui précède dans cet exemple n'a aucun sens pour nos algorithmes. On explique juste ce qui peut se produire en TP.

Définition (Procédures Maple)

On traduit la notion d'algorithme par une procédure MAPLE. La syntaxe est :

```
proc( param1::type1, param2::type2,...)::typeRés ;
description "Chaîne de caractère1", "Chaîne 2",
    ..., "Chaîne p" ;
local var1::type1, var2::type2, ... ;
instr1;
instr2;
... ;
instrn;
end proc;
```

Syntaxe d'une procédure Maple

- Les terminateurs ; sont obligatoires.
- `description ... ;` est facultative. Elle permet d'afficher un texte d'explication (de spécification) lorsqu'on demande l'affichage de la procédure.
- La liste des paramètres, typés, est éventuellement vide.
- `typeRés ;` est le type du résultat de la procédure. On peut omettre cette déclaration, mais dans ce cas, il faut aussi omettre le ; qui suit le type du résultat (ouch !).
- La dernière instruction exécutée doit être une instruction de type `return expression ;` Ce n'est pas forcément `instrn` (voir exemples) Toute exécution de l'instruction `return expression ;` termine l'exécution de l'algorithme en renvoyant la valeur de `expression`.

Fonction Maple

Un cas particulier d'algorithme est celui d'une fonction « classique » : il n'y a pas de variables déclarées dans le corps, il n'y a pas de structuration des instructions. Le corps est réduit à `return expression ;`. On traduira ainsi la fonction mathématique classique

$$f: x1 : type1, \dots, xn : typen \longrightarrow expression$$

`(var1::type1, var2::type2, ..., varn::typen) -> expression`

C'est réellement une abréviation de la procédure Maple :

```
proc(var1::type1, var2::type2, ..., varn::typen) return
expression; end proc;
```

On notera que dans la syntaxe Maple des opérateurs de type `->`, on ne peut donner un type au résultat (contrairement aux procédures)... **soupir**.

Nom des procédures et fonctions

Enfin un algorithme a un nom `NomA` qu'on déclare en MAPLE en affectant la procédure (la fonction) à la variable qui a pour nom `NomAlgo`

```
NomAlgo := proc( x1::type1, ..., xp::typep ) ... end proc;
```

ou

```
NomAlgo := ( x1::type1, ..., xp::typep ) -> expression ;
```

Une telle procédure ou fonction s'utilise alors comme un appel de fonction :

```
NomAlgo (arg1, ..., argp)
```

La sémantique est celle de la substitution des valeurs des arguments aux paramètres dans le corps de la procédure, resp. dans l'expression de la fonction. Le corps ainsi substitué, resp. l'expression substituée sont évalués à leur tour.

Algorithme : ExAlgo

Données : a,b : Entier

Résultat : description du résultat qui est, par exemple, entier

/* Déclaration des variables locales */

u :Entier, m :booléen;

début

instruction 1 ;

... ;

renvoyer u ;

fin algorithme

```
ExAlgo := proc(a::integer, b::integer)::integer ;
description " Spécification de la procédure" ;
# ici la déclaration de variables locales
local u :: integer , m :: boolean ;
instruction 1 ;
... ;
return u ;
end proc;
```

Algorithme : ExFonction

Données : a : Entier

Résultat : L'entier $2 * a + 1$

début

| **renvoyer** $2*a + 1$;

fin algorithme

```
ExFonction := (a::integer) -> 2 * a + 1 ;
```

Exemple (Utilisation des procédures/fonctions)

Syntaxe et sémantique sont exactement les mêmes en langage d'algorithme et en Maple (enfin presque **soupir**).

- $x::integer$; ; $x := \text{ExAlgo}(3,8)$; On substitue 3 à a et 8 à b dans le corps de Exalgo puis on évalue ce corps. Le résultat est celui de l'évaluation de exp dans la première instruction return exp ; rencontrée
- $y::integer$; ; $y := \text{ExFonction}(3)$;
On substitue 3 à a dans l'expression $2 * a + 1$ et on renvoie l'évaluation du résultat qui est ...

L'affectation et la séquence :

```
a := expression ;
inst1 ; ... ; instn;
```

```
a := expression ;
inst1 ; ... ; instn;
```

Les instructions conditionnelles :

```
si Cond alors
| Inst1 ;
| ... ;
| Instn;
fin si
```

```
if Cond then
Inst1;
...;
Instn;
end if ;
```

```
si Cond alors
| Inst1 ; ... ; Instk ;
sinon
| Instp ; ... ; Instn
fin si
```

```
if Cond then
Inst1 ; ... ; Instk ;
else
Instp ; ... ; Instn ;
end if ;
```

```
si Cond1 alors
| Inst1 ; ... ;
sinon si Cond2 alors
| Inst2 ; ... ;
sinon
| Inst3 ; ... ;
fin si
```

```
if Cond1 then
Inst1 ; ... ;
elif Cond2 then
Inst2 ; ... ;
else
Inst3 ; ... ;
end if ;
```

Les itérations :

```
pour v de e1 à e2 par
pas de e3 faire
| Inst;...;
finpour
```

```
for v from e1 to e2 by e3 do
  Inst ; ... ;
end do;
```

```
pour v de e1 à e2 faire
| Inst;...;
finpour
```

```
for v from e1 to e2 do
  Inst ; ... ;
end do;
```

```
tant que Cond faire
| Inst;...;
fin tq
```

```
while Cond do ;
  Inst ; ... ;
end do;
```

Nous utiliserons une petite partie des nombreuses variantes de `array` Maple.

- Déclaration du tableau `T` par l'instruction `T :: array(1..Taille);` où `T` est le nom du tableau, `Taille` sa taille entière ≥ 1 . Attention on ne peut typer les éléments d'un `array` Maple dans la déclaration (**soupir**). Attention encore, la déclaration n'est pas la même pour une variable tableau locale à une procédure (**gros soupir**). Attention enfin, la déclaration du type est possible pour un tableau paramètre d'une procédure (**très gros soupir**).
- On calcule la taille d'un tableau en utilisant la fonction `taille`. Attention, cette fonction n'est pas une fonction standard Maple.
- On accède à l'élément de rang (d'indice) `i` par `T[i]`. Attention, un élément non initialisé a quand même une valeur : le nom symbolique qui permet de le manipuler. Ainsi, après :
`T := array(1..3); T[1] := 4 ; T[2] := 8 ;`
`T[3]` renvoie `T3`. Bien sûr, ceci n'a aucun sens algorithmiquement parlant.
- On peut déclarer et initialiser un tableau en une seule instruction :
`T := array(1..4, [2,5,7,1]);`

- Attention, un tableau s'évalue à son nom ! Après

```
T := array(1..3); T[1] := 4 ; T[2] := 8 ;
```

```
T ; renvoie T alors que eval(T) ; renvoie [4,8,?3]
```

Donc on s'interdira `S := T;`, car après cette instruction, d'une certaine façon, `S`, `T` sont deux noms pour un même tableau.

Ainsi `S := T; S[1] := 0; eval(T);` renvoie `[0,8,?3]`

Tableau comme donnée d'une procédure

On typera le paramètre `NomParam::array(TypeElements)`, où `TypeElements` est le type commun des éléments du tableau paramètre.

Exemple

```
restart;
T := array(1..3, [1,2,3]); U:=array(1..2, [true, true]);
f := proc( S::array(integer))::integer;
  local x::integer, i::integer;
  x := 0;
  for i from 1 to taille(S) do
    x := x + S[i];
  end do;
  return x;
end proc;
```

Après ce début `f(T)` ; renvoie ... et `f(U)` ; ...

Tableau comme résultat d'une procédure

Un tableau peut être le résultat d'une procédure. Dans ce cadre, le type du résultat est `array (TypeElements)`. Attention à renvoyer **la valeur** du tableau déclaré dans la procédure, et non son nom :

```
local t::array;...; return eval(t);...
```

Exemple

```
h := proc(x::integer)::array(integer);
  local t::array, i::integer;
  t:=array(1..x);
  for i from 1 to x do t[i] := i end do;
  return eval(t);
end proc;
```

L'exécution de `S := array(1..3); S := h(3);` modifie l'environnement dans lequel maintenant S est ...

Déclaration :

```
T : array 1..5 of Entier;
T[1] := 3; T[3] := 5;
```

```
T :: array;
T:=array(1..5) ;
T[1] := 3; T[3] := 5;
```

Déclaration, initialisation :

```
T : array 1..3 of Entier;
T[1] := 8; T[2] := 7 ;
T[3] := 6;
```

```
T :: array;
T := array(1..3, [8,7,6]) ;
```

Accès aux éléments d'un tableau.

```
T : array 1..3 of Entier;
T[1] := 8; T[2] := 7 ;
T[3] := 6;
T[2] := T[1] mod T[3];
```

```
T :: array;
T := array(1..3, [8,7,6]) ;
T[2] := T[1] mod T[3];
```

Tableau paramètre d'un algorithme.

Algorithme : f

Données : T un tableau d'entiers
Résultat : somme des éléments de T

S, i : Entier;

début

S := 0;

pour i de 1 à

taille(T) **faire**

| S := S + T[i]

finpour

renvoyer S

fin algorithme

```
x : Entier;
T : array 1..3 of Entier;
T[1] := 8; T[2] := 7 ;
T[3] := 6; x := f(T); ...;
```

```
f :=
proc(
  T::array(integer)::integer;
  description "renvoie la ",
    "somme des éléments de T";
  local i::integer, S::integer;
  S := 0;
  for i from 1 to taille(T) do
    S := S + T[i];
  end do;
  return S;
end proc;
```

```
x :: integer; T :: integer;
T := array(1..3, [8,7,6]) ;
x := f(T) ; ...;
```

Tableau résultat d'un algorithme.

Algorithme : f

Données : N un entier positif

Résultat : Le tableau de taille N dont les éléments sont les N premiers entiers impairs non nuls

i : Entier; T : array 1..N of Entier ;

début

pour i de 1 à N **faire**

| T[i] := 2*i - 1

finpour

renvoyer T

fin algorithme

```
R:array(1..3) of integer;
R := f(3); ...;
```

```
f := proc(
  N::integer)::array(integer);
  description "renvoie le",
    "tableau des N premiers",
    "impairs";
  local i::integer, T::array ;
  T :=array(1..N);
  for i from 1 to N do
    T[i] := 2 * i - 1;
  end do;
  return eval(T);
end proc;
R:=array(1..3);
R := f(3); ...;
```