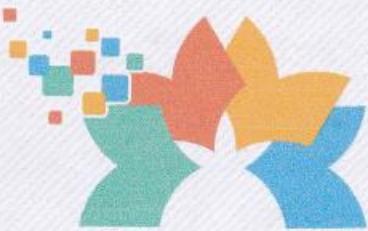




Centre national de télé-enseignement de Madagascar



# CNTEMAD

Apprendre et réussir en toute liberté

[www.cntemad.mg](http://www.cntemad.mg)

cntemad@cntemad.mg  
22 600 57

## LICENCE 1 EN INFORMATIQUE

MODULE N°03  
TOME 1

ALGORITHMIQUE  
ET  
LANGAGES 1

Mod 03 T1



**ALGORITHMIQUE ET LANGAGES I**

**Par RAKOTOZANANY Andriamora Norbert**

## **I. INTRODUCTION**

Les notions d'algorithme et de programme sont très utilisées en informatique. Elles sont souvent mélangées, voir confondues, alors que même si elles sont fortement liées, elles s'appliquent à des domaines différentes.

Un algorithme est appliqué quotidiennement par l'homme pour résoudre un problème donné ou pour arriver à un résultat donné.

Tandis qu'un programme est donné à un ordinateur pour lui permettre d'effectuer une tâche donnée.

L'homme, face à un problème donné, recherche une solution en élaborant un algorithme. Ensuite, les informaticiens traduisent cet algorithme dans un langage compréhensible par l'ordinateur. Cette traduction est appelée programme.

Les quelques définitions libres suivantes sont nécessaires pour mieux comprendre l'algorithmique et la programmation.

### **I.1 Algorithme**

C'est une méthode de calcul qui indique la démarche à suivre pour résoudre une série de problèmes équivalents en appliquant dans un ordre précis une suite finie de règles.

### **I.2 Algorithmique**

C'est l'ensemble des règles et des techniques qui sont impliquées dans la définition et la conception des algorithmes.

### **I.3 Informatique**

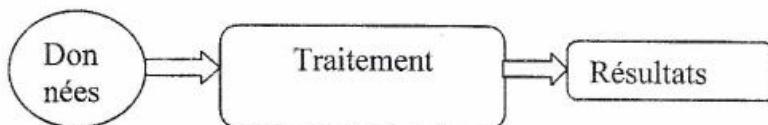
C'est une combinaison des mots « information » et « automatique ». Il signifie « traitement automatique de l'information ».

Automatique signifie « fait par une machine » qui est l'ordinateur.

### **I.4 Programme**

Un programme est une suite d'instructions élémentaires écrit dans un langage de programmation, et exécutée par un ordinateur. Un programme reçoit un ensemble d'informations de départ appelé « données » ou « input », effectue les calculs nécessaires ou fait les traitements, et renvoie un ensemble d'informations appelé « résultat ».

Un programme fonctionne toujours selon le schéma simple suivant :



### **I.5 Programmation**

C'est la technique d'élaboration des programmes.

### **I.6 Langage de programmation**

C'est un ensemble de « mots » ou « instructions » et de règles appelé syntaxe qu'une machine est capable de « comprendre » et d'exécuter.

Il existe plusieurs langages de programmation dont Pascal, C, C++, C#, VB ou Visual Basic, Java, ...

Quelque soit le langage de programmation utilisé, il est toujours articulé autours des deux notions de base suivantes :

- Les informations à traiter et les résultats : «Comment seront-elles représentées ?»
- Les actions à effectuer par la machine : «Quelles sont les actions que la machine sait faire ? Comment les agencer pour aboutir aux résultats ?»

Ces deux notions restent invariantes quelque soit le langage de programmation utilisé car elles ne dépendent pas du langage mais de la machine.

Cette solution invariante d'un problème lorsqu'il est destiné à être traité par une machine est appelée « algorithme informatique ».

Quand un informaticien reçoit aura à résoudre un problème donné, il déterminera d'abord l'algorithme pour la résolution de ce problème, puis traduira cette algorithme dans un langage de son choix.

### **I.7 Représentation d'un algorithme**

Il y a deux techniques de représentation d'un algorithme:

- **L'Organigramme:** représentation graphique avec des symboles (carrés, losanges, etc.). Cette technique offre une vue d'ensemble de l'algorithme très facile à lire pour les algorithmes de petite taille mais elle est quasiment abandonnée aujourd'hui car les algorithmes sont de plus en plus long et la taille du diagramme n'est plus gérable.
- **Le pseudo-code ou langage algorithmique ou langage de définition d'algorithme:** C'est une représentation textuelle avec une série de conventions ressemblant à un langage de programmation. Cette technique est plus pratique pour écrire un algorithme et est très largement utilisée de nos jours.

C'est cette deuxième technique que nous allons utiliser par la suite.

**I.8 Structure générale d'un algorithme**

Un algorithme écrit en pseudo-code prend la forme générale suivante :

**Nom de l'algorithme**

**Déclarations**

- ✓ Déclaration des Constantes
- ✓ Déclaration des Variables
- ✓ Déclaration des Tableaux
- ✓ Déclaration des Procédures et Fonctions

**Corps**

**Début**

*Actions*

**Fin**

Le nom de l'algorithme doit être un identificateur.

Un identificateur un nom utilisé pour désigner (ou identifier) quelque chose. Il doit être en une seul mot (suite de caractères sans espace).

Nous verrons les autres parties plus loin, au fur et à mesure de notre progression.

## II. LES VARIABLES

### 1. Qu'est ce qu'une variable ?

Dans un programme informatique consiste à manipuler des valeurs. On a donc besoin de stocker provisoirement ces valeurs. Il peut s'agir de données issues du disque dur, fournies par l'utilisateur (frappées au clavier) ou provenir des résultats intermédiaires ou définitifs du programme lui même.

Ces données peuvent être de plusieurs types : des nombres, du texte, ou d'autres types.

C'est pour ces stockages d'information au cours d'un programme qu'on utilise les **variables**.

Une variable est un emplacement en mémoire repéré par une adresse. Le nom de la variable sert à référencier la variable.

### 2. Types des variables

Lorsqu'on déclare une variable, il ne suffit pas de réservé un emplacement mémoire. On préciser la taille de cet emplacement et le **type de codage** utilisé.

D'une manière générale, on distingue 3 types de données:

- Le type numérique ;
- Le type alphanumérique ou texte;
- Le type booléen ou logique.

#### Le type numérique

Il contient plusieurs sous-types selon le tableau suivant.

Type Numérique	Plage
Octet	0 à 255
Entier simple	-32 768 à 32 767
Entier long	-2 147 483 648 à 2 147 483 647
Réel simple	$-3,40 \times 10^{38}$ à $-1,40 \times 10^{45}$ pour les valeurs négatives $1,40 \times 10^{-45}$ à $3,40 \times 10^{38}$ pour les valeurs positives
Réel double	$1,79 \times 10^{308}$ à $-4,94 \times 10^{-324}$ pour les valeurs négatives $4,94 \times 10^{-324}$ à $1,79 \times 10^{308}$ pour les valeurs positives

## Le Type alphanumérique

Il y a le sous-type caractère formé par un seul caractère et le sous-type texte ou chaîne formé de plusieurs caractères.

## Le type booléen

Le dernier type de variables est le type **booléen** : on y stocke uniquement les valeurs logiques VRAI et FAUX.

### 3. Déclaration de variable

Avant de pouvoir utiliser une variable, il faut **la déclarer**. Ceci se fait tout au début de l'algorithme, avant même les instructions proprement dites.

Il consiste à déclarer le nom et le type de chaque variable.

Un **nom** de variable est un identificateur, donc composé uniquement des lettres et des chiffres, et du caractère trait d'union (touche 8). Il commence impérativement par une lettre.

La longueur du nom de variable dépend du langage utilisé.

En pseudo-code algorithmique, la longueur du nom de variable n'est pas limitée bien qu'on évite généralement les noms trop longs.

En pseudo-code, une déclaration de variables suit cette forme générale ou syntaxe :

**Variable <nom\_variable> : <type\_varianble> ;**

Où on a :

Variable : mot clé obligatoire :

<nom\_variable> : à remplacer par le nom de la variable ;

<type\_varianble> : à remplacer par le type de la variable ;

Le caractère « : » sépare le nom et le type de la variable. Il est obligatoire.

Le caractère « ; » termine la déclaration d'une variable. Il est aussi obligatoire.

## Exemples

Variable nom : texte;

Variables i , j, k : entier;

Variable ok : booléenne;

Ce sont des déclarations de variables correctes.

Dans la deuxième ligne, on déclare en même temps trois variables i, j et k de même type.

Elles sont séparées par le caractère « , ».

Variable nombre de ligne : entier ;

Ce nom de variable n'est pas correct car il contient des espaces.

### 4. L'instruction d'affectation

Pour donner une valeur à une variable, on utilise l'affectation, c'est-à-dire lui attribuer une valeur.

La valorisation d'une variable est toujours obligatoire avant de pouvoir l'utiliser car après sa déclaration, la valeur d'une variable est **indéterminée**. Autrement dit, la variable peut avoir n'importe quelle valeur.

En pseudo-code, l'instruction d'affectation se note avec le symbole  $\leftarrow$

Ainsi :

$x \leftarrow 24 ;$

Attribue la valeur 24 à la variable x.

#### Ordre des instructions

L'ordre dans lequel les instructions sont écrites va jouer un rôle essentiel dans le résultat final.

Considérons les deux algorithmes suivants :

#### Exemple 1

Variable A : entier ;

Début

$A \leftarrow 34 ;$

$A \leftarrow 12 ;$

Fin

#### Exemple 2

Variable A : entier ;

Début

$A \leftarrow 12 ;$

$A \leftarrow 34 ;$

Fin.

Dans le premier cas la valeur finale de A est 12, dans l'autre elle est 34

### 5. Expressions et opérateurs

Dans une instruction d'affectation, on trouve :

- à gauche de la flèche, un nom de variable, et uniquement cela.
- à droite de la flèche, ce qu'on appelle une expression.

En informatique, le terme expression désigne une seule chose bien précise :

C'est une suite de variables et de valeurs, reliées par des opérateurs, et équivalent à une seule valeur

Un opérateur est un signe qui relie deux valeurs, pour produire un résultat.

Ainsi :

7 ;      5+4 ;    123-45+844 ;    x-12+5-y

sont toutes des expressions valides si x et y sont des variables de type numérique.

Dans une affectation, il faut que l'expression située à droite de la flèche soit du même type que la variable située à gauche.

Les opérateurs possibles dépendent du type des valeurs qui sont en jeu.

a) Les Opérateurs numériques :

Ce sont les quatre opérations arithmétiques :

+ : addition

- : soustraction

\* : multiplication

/ : division

Mentionnons également le  $\wedge$  qui signifie « puissance ».  $45$  au carré s'écrira donc  $45 \wedge 2$ .

Enfin, on a le droit d'utiliser les parenthèses, avec les mêmes règles qu'en mathématiques. La multiplication et la division ont « naturellement » priorité sur l'addition et la soustraction. Les parenthèses ne sont ainsi utiles que pour modifier cette priorité naturelle.

Ainsi,  $12 * 3 + 5$  et  $(12 * 3) + 5$  valent strictement la même chose, à savoir 41.

Tandis que  $12 * (3 + 5)$  vaut  $12 * 8$  soit 96.

b) Opérateur alphanumérique : &

Cet opérateur appelé opérateur de concaténation, permet de concaténer, autrement dit de mettre bout à bout, deux chaînes de caractères.

Exemple :

Variables A, B, C : Chaine ;

Début;

    A  $\leftarrow$  "Université ";

    B  $\leftarrow$  "CNTEMAD";

    C  $\leftarrow$  A & B ;

Fin.

La valeur de C à la fin de l'algorithme est " Université CNTEMAD "

c) Opérateurs logiques (ou booléens) :

Il s'agit du ET, du OU, du NON et du XOR (ou exclusif)

Si a et b sont 2 variables de type booléen, le résultat de ces opérations, selon les valeurs de a et b, sont données par le tableau suivant appelé également table de vérité.

<u>a</u>	<u>b</u>	<u>a ET b</u>	<u>a OU b</u>	<u>NON a</u>	<u>a XOR b</u>
F	F	F	F	V	F
F	V	F	V	V	V
V	F	F	V	F	V
V	V	V	V	F	F

Dans ce tableau, F signifie « Faux » et V « Vrai ».

d) Opérateurs relationnels ou opérateurs de comparaison :

Ce sont les opérateurs de comparaison habituels :

= pour égalité

< pour strictement inférieur ou inférieur

<= pour inférieur ou égal

> pour strictement supérieur ou supérieur

<= pour supérieur ou égal

≠ ou <> ou != pour différent.

Le résultat d'une comparaison est toujours une valeur booléenne soit Vrai ou Faux.

Notez que :

NON = , c'est ≠ et vis versa.

NON <, c'est >= et vis versa

NON >, c'est <= et vis versa

### III. LES INSTRUCTIONS DE BASE

Les instructions de base, qu'on appelle également action, sont les seules actions reconnues par les ordinateurs. Aucune autre action ne devrait figurer dans un algorithme informatique ou dans un programme.

#### 1. L'affectation

Elle permet de valoriser une variable à l'aide d'une expression.

C'est l'affectation qui permet de faire effectuer un calcul par une machine.

L'affectation est l'action principale d'un algorithme.

Sa syntaxe est :

**<nom\_variable> ← <expression>;**

Exemple :

z ← x-y ;

n ← 2\*k +1 ;

prenom ← prenom1 & prenom2 ;

On souligne bien qu'on affecte une valeur dans une variable. Donc à gauche d'une affectation, on doit avoir une variable.

#### 2. La lecture

La lecture est une instruction qui ordonne la machine à lire une donnée à partir du clavier et l'affecter dans une variable.

L'instruction de lecture permet à l'utilisateur (la personne qui va utiliser le programme) de saisir une donnée à partir du clavier.

Sa syntaxe est :

**Lire(<nom\_variable>);**

Ou bien :

**Saisir(<nom\_variable>);**

#### 3. L'écriture

L'écriture permet d'afficher des résultats à l'écran (ou de les écrire dans un fichier).

Sa syntaxe est :

**Ecrire (<Expression1>, <Expression1>, ..., <ExpressionN>);**

Ou bien :

**Afficher (<Expression1>, <Expression1>, ..., <ExpressionN>);**

Exemple :

Afficher ("Voici le resultat : ", res);

Si la variable res vaut 100, cette instruction affiche à l'écran : Voici le resultat : 100

### Remarque :

Pour faire la saisie de données, il est fortement conseillé d'afficher d'abord un message d'invite avant de lancer l'instruction de lecture. Cela permet à l'utilisateur de savoir ce qu'il doit faire.

### Exemple (lecture et écriture)

Voici un algorithme qui demande un nombre entier à l'utilisateur, puis qui calcule et affiche le double de ce nombre

```
Algorithme Calcul_double ;
variables A, B : entier ;
Début
    Ecrire("entrer le nombre : ");
    Lire(A);
    B ← 2*A;
    Ecrire("le double de ", A, " est :", B);
Fin.
```

Cet algorithme aura comme résultat, si on saisie 7 :

```
Entrer le nombre : 7
Le double de 7 est 14
```

### 4. La structure de contrôle alternative ou instruction conditionnelle ou Test

Les instructions conditionnelles servent à exécuter une instruction ou une séquence d'instructions dans le seul cas où une condition est vérifiée.

Os syntaxe est la suivante:

**SI <condition> ALORS**

<instruction ou suite d'instructions 1>

[ SINON

<instruction ou suite d'instructions 2>]

FIN ;

Elle s'exécute de la manière suivante :

- <condition> est évaluée. C'est une expression booléenne dont la valeur est soit Vrai soit Faux.
- Si elle vaut Vrai, c'est <instruction ou suite d'instructions 1> qui est exécutée puis la machine passe à l'instruction après FIN ;
- Si elle vaut Faux, c'est <instruction ou suite d'instructions 2> qui est exécutée puis la machine passe à l'instruction après FIN ;

Donc c'est l'une, et l'une seulement, des instructions après ALORS et SINON sera exécutée en un temps donné.

Les [ ] qui encadre SINON signifie, en lecture de syntaxe, que cette partie est facultative.

Exemple (Si...Alors...Sinon)

```
Algorithme AffichageValeurAbsolue ;
Variable x : réel ;
Début
    Ecrire (" Entrez un réel : ");
    Lire (x);
    Si (x < 0) alors
        Ecrire ("la valeur absolue de ", x, "est:",-x)
    Sinon
        Ecrire ("la valeur absolue de ", x, "est:",x)
    Fin;
Fin.
```

Exemple (Si...Alors)

```
Algorithme AffichageValeurAbsolue ;
Variable x,y : réel ;
Début
    Ecrire (" Entrez un réel : ");
    Lire (x);
    y← x;
    Si (x < 0) alors
        y ← -x
    Fin;
    Ecrire ("la valeur absolue de ", x, "est:",y);
Fin.
```

Ces deux algorithmes donnent exactement le même résultat.

Tests imbriqués

Les tests peuvent avoir un degré quelconque d'imbriques :

```
Si <condition1> alors
    Si <condition2> alors
        <instructionsA>
    Sinon
        <instructionsB>
    Finsi
Sinon
    Si <condition3> alors
        <instructionsC>
    Finsi
Finsi
```

## Tests imbriqués exemple

```
Algorithme TestNombre ;  
Variable n : entier ;  
Début  
    Ecrire ("entrez un nombre : ") ;  
    Lire (n) ;  
    Si (n < 0) alors  
        Ecrire ("Ce nombre est négatif") ;  
    Sinon  
        Si (n = 0) alors  
            Ecrire ("Ce nombre est nul")  
        Sinon  
            Ecrire ("Ce nombre est positif")  
        Fin ;  
    Fin ;  
Fin.
```

## Version 2 du même test (sans imbrication)

```
Algorithme TestNombre ;  
Variable n : entier ;  
Début  
    Ecrire ("entrez un nombre : ") ;  
    Lire (n) ;  
    Si (n < 0) alors  
        Ecrire ("Ce nombre est négatif")  
    Fin ::  
    Si (n = 0) alors  
        Ecrire ("Ce nombre est nul")  
    Fin ;  
    Si (n > 0) alors  
        Ecrire ("Ce nombre est positif")  
    Fin ;  
Fin
```

### Remarque :

Dans la version 2 on fait trois tests systématiquement alors que dans la version 1, si le nombre est négatif on ne fait qu'un seul test

### Conseil :

Utilisez les tests imbriqués pour limiter le nombre de tests et placez d'abord les conditions les plus probables en avant.

## 5. Le choix multiple

On utilise cette structure de contrôle quand on a à choisir entre n actions différentes ( $n > 2$  sinon on pourrait utiliser SI...) selon n valeurs possibles d'une variable de type énumérable (entier ou caractère).

Syntaxe:

```
SELON <nom_variable>
Debut
    cas v1:
        <instruction1>;
    cas v2:
        <instruction2>
    ...
    Autres:
        <instruction N>;
Fin;
```

Les v1, v2, ... sont des valeurs possibles de la variable <nom\_variable>.

Elle fonctionne de la manière suivante :

- On détermine La valeur de <nom\_variable> ;
- Si elle vaut v1, on exécute <instruction1> puis on passe après Fin ;
- Si elle vaut v2, on exécute <instruction2> puis on passe après Fin ; et ainsi de suite pour toutes les autres valeurs possibles de la variable <nom\_variable>.

Exemple :

Soit un algorithme qui détermine et affiche le nom du chef lieu de province selon le numéro de code de la province.

```
Algorithme province ;
Variable codeprov : entier ;
    nomprov : chaîne ;
Debut
    Ecrire ("Entrez le code de la province : ");
    Lire(codeprov) ;
    Selon codeprov
        Cas 1 :
            nomprov ← "Antananarivo";
        Cas 2 :
            nomprov ← "Antsiranana";
        Cas 3 :
            nomprov ← " Fianarantsoa ";
        Cas 4 :
            nomprov ← "Mahajanga";
        Cas 5 :
            nomprov ← "Toamasina";
        Cas 6 :
            nomprov ← "Toliara";
        Autres :
            nomprov ← "Code incorrect";
    Fin ;
    Ecrire ("C'est la province de ", nomprov) ;
Fin.
```

Voici deux exemples d'exécution de cette algorithme :

Entrez le code de la province : 9  
C'est la province de Code incorrect

Entrez le code de la province : 4  
C'est la province de Mahajanga

## 6. La boucle POUR

Une boucle est une structure de contrôle permettant de répéter plusieurs fois les mêmes actions. La structure POUR est utilisée quand le nombre de répétition est connu d'avance par le programmeur. Dans ce cas, la condition de bouclage dépend de ce nombre..

C'est le cas aussi quand la ou les actions sont à exécuter chaque fois qu'une variable de type énumérable prend une valeur en parcourant un intervalle entre une borne inférieure et une borne supérieure.

### Syntaxe :

```
POUR <compteur> DE <borneinf> A <borbesup> FAIRE  
    <instructions>  
FIN ;
```

### Fonctionnement :

- La valeur <borneinf> est affectée à la variable <compteur>;
- <instructions> est exécutée;
- <compteur> est **incrémentée**, c'est-à-dire, qu'on lui affecte son ancienne valeur + 1 (comme si on a fait : `compteur ← compteur + 1`);
- Si <compteur> est encore  $\leq$  <borbesup>, <instructions> est exécutée et on revient au tiret précédent.
- Si <compteur> est devenu  $>$  <borbesup>, on passe à l'instruction après FIN ;

### Exemple :

Calcul de  $x^n$  où  $x$  est un réel non nul et  $n$  un entier positif ou nul

```
Algorithme CalculPuissance ;  
Variables x, puiss : réel  
n, i : entier  
Debut  
    Ecrire (" Entrez la valeur de x : ") ;  
    Lire (x) ;  
    Ecrire (" Entrez la valeur de n : ") ;  
    Lire (n) ;  
    puiss ← 1 ;  
    Pour i de 1 à n Faire  
        puiss← puiss*x ;  
    Fin ;  
    Ecrire (x, " à la puissance ", n, " est égal à ", puiss) ;  
Fin.
```

### REMARQUE

Il est **interdit de modifier la valeur** du compteur et des bornes à l'intérieur d'une boucle POUR. En effet, une telle action perturbe le nombre d'itérations prévu par la boucle POUR et donnera un erreur.

### 7. La boucle TANT QUE

Cette structure permet de répéter une série d'instructions en fonction d'une condition qu'on vérifie à chaque pas.

Syntaxe :

```
TANT QUE <condition> FAIRE  
    <instructions>;  
    FIN;
```

Fonctionnement :

- <condition> (dite condition de contrôle de la boucle) est évaluée avant chaque itération
- si la condition est vraie, on exécute <instructions> (corps de la boucle), puis, on retourne tester la condition. Si elle est encore vraie, on répète l'exécution, ...
- si la condition est fausse, on sort de la boucle et on exécute l'instruction qui est après Fin ;

Remarque :

- Le nombre d'itérations dans une boucle TantQue n'est pas connu au moment d'entrée dans la boucle. Il dépend de l'évolution de la valeur de condition
- Une des instructions du corps de la boucle doit absolument changer la valeur de condition de vrai à faux (après un certain nombre d'itérations), sinon le programme tourne indéfiniment et on aura ce qu'on appelle une boucle infinie.

Premier exemple de boucle infinie :

```
i ← 2;  
TantQue i > 0 Faire  
    i ← i+1;  
Fin;
```

Ici, on part de  $i=2$  puis on l'incrémente à chaque passage dans la boucle. Donc  $i$  restera à jamais  $>0$  et on aura une boucle infinie.

Deuxième exemple de boucle infinie :

```
i ← 1;  
TantQue i < 10 Faire  
    Ecrire(i);  
Fin;
```

Ici, on part de  $i=1$  puis on ne modifie pas cette valeur lors des passages dans la boucle. Donc  $i$  restera à jamais égal à 1 qui est  $< 10$  et on aura une boucle infinie.

## Exemple de boucle TANT QUE correcte :

Un algorithme qui détermine le premier nombre entier N tel que la somme de 1 à N dépasse strictement 100

```
Algorithme PremierN;  
Variables som, i : entier;  
Debut  
    i ← 0 ;  
    som← 0 ;  
    TantQue (som <=100)  
        i ← i+1  
        som ← som+i  
    FinTantQue ;  
    Ecrire (" La valeur cherchée est N= ", i) ;  
Fin
```

## Deuxième version du même algorithme.

La différence se trouve au niveau de l'ordre des instructions et des valeurs initiales.

```
Algorithme PremierN;  
Variables som, i : entier  
Debut  
    som ← 0;  
    i ← 1 ;  
    TantQue (som <=100)  
        som ← som + I;  
        i ← i+1 ;  
    FinTantQue ;  
    Ecrire (" La valeur cherchée est N= ", i-1) ;  
Fin
```

Ici, la valeur du N recherchée est i-1 et non pas i comme dans la première version.

## Relation avec la structure POUR ...

Une boucle POUR peut toujours être transformée en TANT QUE et on aura :

```
compteur ← borneinf ;  
TANT QUE compteur <= bornsup FAIRE  
    Instructions ;  
    compteur ← compteur +1 ;  
FIN ;
```

Par contre, il y a des boucles TANT que qu'on ne peut pas réaliser avec POUR.

Exemple :

```
Ecrire("Entrez B : ") ;  
Lire(B) ;  
Ecrire("Entrez A : ") ;  
Lire(A) ;  
TANT QUE A < (B+4)/2 FAIRE  
    Ecrire("C'est normal") ;  
    Ecrire("Entrez B : ") ;  
    Lire(B) ;  
    Ecrire("Entrez A: ") ;  
    Lire(A) ;  
FIN.  
Ecrire("La situation est devenue anormale") ;
```

Ici, la condition de boucle  $A < (B+4)/2$  n'exprime le parcours d'un intervalle par une variable. Donc, cette boucle ne peut pas être exprimée avec POUR.

Voici un exemple d'exécution de cette boucle.

```
Entrez B : 14  
Entrez A :7  
C'est normal  
Entrez B : 13  
Entrez A :7  
C'est normal  
Entrez B : 12  
Entrez A :7  
C'est normal  
Entrez B : 10  
Entrez A :7  
La situation est devenue anormale
```

## 8. La boucle REPETER ... JUSQU'A

Cette structure est presque identique à la structure TANT QUE sauf que l'évaluation de la condition de boucle se fait après l'exécution des instructions de la boucle alors que c'est avant dans tant que.

La conséquence est que ces instructions de boucle seront exécutées au moins une fois même si la condition de bouclage est fausse dès le départ.

Syntaxe :

```
REPETER
    instructions ;
    JUSQU'A condition ;
```

### Exemple

Un algorithme qui détermine le premier nombre entier N tel que la somme de 1 à N dépasse strictement 100 (version avec répéter jusqu'à)

```
Algorithme PremierNRep ;
Variables som, i : entier
Debut
    som ← 0
    i ← 0
    Répéter
        i ← i+1
        som ← som+i
    Jusqu'à ( som > 100 )
    Ecrire (" La valeur cherchée est N= ", i)
Fin
```

Relation entre boucles TANT QUE ... FAIRE et REPETER ... JUSQU'A ..

On remarquera que la condition de TANT QUE est une condition de poursuite de la boucle tandis que celle REPETER ... JUSQU'A est une condition d'arrêt de la boucle. Dons, on a :

Condition de TANT QUE = NON (condition de REPETER JUSQU'A°)

Exemple : Affichage des entiers de 1 à 9 avec les 2 structures

Avec TANT QUE	Avec REPETER ... JUSQU'A
<u>i &lt; 1 ;</u> <u>TANT QUE i &lt; 10 FAIRE</u> <u>Ecrire-(i) ;</u> <u>Fin ;</u>	<u>i &lt; 1 ;</u> <u>REPETER</u> <u>Ecrire(i) ;</u> <u>JUSQU'A i &gt;= 10 ;</u>

## IV. LES TYPES CONSTRUITS

A partir des types de données de base (caractère, nombre et booléen), le programmeur peut construire de nouveaux types de données pour pouvoir manier des informations plus élaborées.

### 1. Le type tableau

#### a) Définition

Un tableau est une série de données de même type rassemblées pour former un seul bloc. Ce bloc est placé dans une seule variable. Chaque élément du tableau est repéré par un nombre entier ou une valeur d'un ensemble discret qu'on appelle indice du tableau.

Tableau	18	16	15	17	16	18	16
Indice	1	2	3	4	5	6	67

Ici, le tableau est une série de nombres qui sont des notes.

L'élément n° 1 du tableau a pour valeur 18, l'élément n° 2 16 et ainsi de suite.

#### b) Déclaration d'une variable de type tableau:

Syntaxe :

Variable <NomTableau>[ <Taille>] de <TypeEleme> ;

Exemple :

Variable Notes[100] de entier ;

#### c) Accès aux éléments d'un tableau :

Pour désigner les notes du tableau précédent, on utilisera Notes[1], Notes[2], Notes[3],... et on lit Notes de 1 ou Notes indice 1.

On peut les utiliser comme des simples variables soit en les faisant entrer dans une expression, soit en y affectant des valeurs.

Ansi :

Notes[1] ← 18 ;

x ← Notes[1] ;

Est une séquence d'instructions correcte

#### d) Les algorithmes sur les tableaux.

Agir sur un ou des éléments d'un tableau exige qu'on parcoure les éléments de ce tableau à travers une variable utilisée comme indice.

Pour les exemples, on a les variables déclarées e la manière suivantes :

Variable t[N] de entier ;

i: entier ;

- Parcours d'un tableau et saisie des éléments

POUR i DE 1 A 100 FAIRE

Ecrire ("Entez t[", i, "] ");

Lire(t[i]);

FIN ;

- Parcours d'un tableau et affichage des éléments

POUR i DE 1 A 100 FAIRE

Ecrire( t[i] );

FIN ;

## 2. Le type structure ou enregistrement

### a) Définition

- Un type enregistrement ou type structuré est un type T de variable v obtenu en juxtaposant plusieurs variables v1 , v2 , . . . ayant chacune un type T1 , T2 , . . . de type de base.
- Autrement dit, un enregistrement ou une structure est un type composite obtenu en juxtaposant plusieurs types de base.
- Les différentes variables vi sont appelées champs de v. Elles sont repérées par un identificateur de champ. Si v est une variable de type structuré T possédant le champ ch1, alors la variable v.ch1 est une variable comme les autres : (type, adresse, valeur)

ch1	ch2	ch3	ch4
v =			

### b) Déclaration de type structuré :

```
Type nom_de_type = structure  
    nomchamp1 :: typechamp1;  
    nomchamp2 :typechamp2;  
Fin;
```

### c) Déclaration de variable structurée :

```
Variable nomvar : nom_de_type;
```

### d) Désignation d'une information composante:

Chaque information composante est désignée (ou est accédée) par le nom de la variable suivi du caractère « . » (point) puis suivi du nom du champ.

Avec les déclarations ci-dessus, les champs sont :

```
nomvar.nomchamp1  
nomvar.nomchamp2
```

Notez : Pas d'espace ni avant ni après le caractère « . ».

## Exemple

Considérons le type Personne qui est la composition de 3 informations : son nom, son sexe et son âge.

Déclaration du type personne

Type Personne = structure

```
    nom : texte(20);  
    sexe : caractère;  
    age : entier;
```

Fin;

Déclaration de la variable monami de type Personne

Variable monami : Personne;

On peut alors faire des affectations telles que :

```
monami.nom ← "Rajao" ;  
monami.sex ← 'm' ;  
monami.age ← 25 ;
```

## 3. Tableau de structure

Un tableau de structure est un tableau dont le type des éléments est un structure.

Exemple : tableau de Personne.

```
Algorithme remplir_tableau;  
Type Personne = structure  
    nom : texte(20);  
    sexe : caractère;  
    age : entier;  
Fin;  
Variable i : entier;  
T : tableau[1..100] de Personne;  
Début  
    POUR i DE 1 A 100 FAIRE  
        Ecrire("T["i,"].nom = ");  
        Lire( T[i].nom );  
        Ecrire("T["i,"].sexe = ");  
        Lire( T[i].sexe );  
        Ecrire("T["i,"].age = ");  
        Lire( T[i].age );  
    FIN;  
Fin.
```

## V. LES SOUS-PROGRAMMES

Certains problèmes conduisent à des programmes longs, difficiles à écrire et à comprendre. On les découpe en des parties appelées **sous-programmes** ou **modules**.

Les **fonctions** et les **procédures** sont des modules (groupe d'instructions) indépendants désignés par un nom. Elles ont plusieurs **intérêts** :

- permettent de "factoriser" les **programmes**, c'ad de mettre en commun les parties qui se répètent
- permettent une **structuration** et une **meilleure lisibilité** des programmes
- **facilitent la maintenance** du code (il suffit de modifier une seule fois)
- Ces procédures et fonctions peuvent éventuellement être **réutilisées** dans d'autres programmes.

### 1. Fonctions

Le **rôle** d'une fonction en programmation est similaire à celui d'une fonction en mathématique: elle **retourne un résultat à partir des valeurs des paramètres**

Une fonction s'écrit dans la partie déclaration du programme principal selon la syntaxe générale :

**Fonction** nom\_fonction (paramètres et leurs types) : type\_résultat ;

**<Déclarations>** ;

**Début**

Instructions constituant le corps de la fonction  
**retourne** résultat

**Fin** ;

- Pour le choix d'un nom de fonction il faut respecter les mêmes règles que celles pour les noms de variables
- type\_résultat est le type du résultat renvoyé
- L'instruction **retourne** sert à renvoyer la valeur du résultat

#### Exemple de fonction

La fonction SommeCarre suivante calcule la somme des carrés de deux réels x et y :

**Fonction** SommeCarre (x : réel, y: réel ) : réel ;

**variable** z : réel ;

**debut**

**z ← x<sup>2</sup>+y<sup>2</sup> ;**

**retourne (z) ;**

**Fin** ;

La fonction Pair suivante détermine si un nombre est pair :

```
Fonction Pair (n : entier) : booléen ;  
debut  
    retourne ((n mod 2) = 0) ;  
FinFonction
```

### Appel ou utilisation d'une fonction

L'utilisation d'une fonction se fera par simple écriture de son nom dans le programme principale. Le résultat étant une valeur, elle devra être affecté ou être utilisé dans une expression, une écriture, ...

Exemple :

```
Algorithme exempleAppelFonction  
variables z : réel, b : booléen  
Début  
    b ← Pair(3)  
    z ← 5 * SommeCarre(7,2)+1  
    Ecrire("SommeCarre(3,5)= ", SommeCarre(3,5))  
Fin ;
```

Lors de l'appel Pair(3), le **paramètre formel n** de la déclaration de la fonction Pair(n) est remplacé par le **paramètre effectif 3** de l'instruction d'appel.

Il en est de même avec les paramètres formels x, y de la fonction SommeCarre(x,y) lors des 2 appels SommeCarre(7,2) et SommeCarre(3,5).

## 2. Procédures

Dans certains cas, on peut avoir besoin de répéter une tache dans plusieurs endroits du programme, mais que dans cette tache on ne calcule pas de résultats ou qu'on calcule plusieurs résultats à la fois

Dans ces cas on ne peut pas utiliser une fonction, on utilise une **procédure**

Une **procédure** est un sous-programme semblable à une fonction mais qui **ne retourne rien**

Une procédure s'écrit dans la partie déclaration du programme principal sous la forme :

```
Procédure nom_procédure (paramètres et leurs types) ;  
<Déclarations> ;  
Début  
    <Instructions constituant le corps de la procédure> ;  
Fin ;
```

Remarque : une procédure peut ne pas avoir de paramètres

## Appel d'une procédure

L'appel d'une procédure, se fait dans le programme principal ou dans une autre procédure par une instruction indiquant le nom de la procédure selon l'exemple suivant :

**Algorithme exempleAppelProcédure ;**

...

**Procédure exemple\_proc (...);**

...

**Fin;**

**Début**

**exemple\_proc (...);**

...

**Fin.**

- Lors de l'appel, une procédure est utilisée comme une instruction autonome. En fait, quand on a créé une procédure, c'est comme si on a créé une nouvelle commande reconnue par l'ordinateur.

## Exemple

Procédure qui affiche les valeurs d'un tableau de 10 entiers.

**Procedure afficher(t : tableau{10} de entier) ;**

**Variable i : entier ;**

### 3. Mode de passage des paramètres

Les paramètres servent à échanger des données entre le programme principale (ou la procédure appelante) et la procédure ou fonction appelée.

Les paramètres placés dans la déclaration d'une procédure ou fonction sont appelés **paramètres formels**. Ces paramètres peuvent prendre toutes les valeurs possibles mais ils sont abstraits (n'existent pas réellement)

Les paramètres placés dans l'appel d'une procédure ou fonction sont appelés **paramètres effectifs**. Ils contiennent les valeurs à utiliser à la place des paramètres formels.

Le nombre de paramètres effectifs doit être égal au nombre de paramètres formels. L'ordre et le type des paramètres doivent correspondre.

Il existe deux modes de transmission de paramètres dans les langages de programmation :

#### a) **La transmission par valeur :**

Les valeurs des paramètres effectifs sont affectées aux paramètres formels correspondants au moment de l'appel de la procédure. Dans ce mode le paramètre effectif ne subit aucune modification quelles soient les actions faites par le sous-programme appelant.

Pour passer un paramètre par valeur en pseudo-code, on donne uniquement le nom du paramètre suivi de ":" et de son type.

## Syntaxe

**Procédure** <nomproc>(<nomparam> : <type>);

**Début**

... ;

**Fin;**

## Exemple

```
Algorithme ExempleValeur ;
Variable n : entier ;
Procédure AfficherLeDouble( k : entier ) ;
Début
    k ← 2*k ;
    Ecrire("Le double = ", k) ;
Fin ;
Début {début du programme principal ExempleValeur}
    Ecrire("Entrez un nombre : ") ;
    Lire(n) ;
    Ecrire(" n = ", n) ;
    AfficherLeDouble(n) ;
    Ecrire(" n = ", n) ;
Fin.
```

L'exécution de ce programme donne :

```
Entrez un nombre : 5
n = 5
Le double = 10
n = 5
```

La modification de k en  $2*k$  ne s'applique pas à la variable globale n utilisée comme paramètre effectif car utilise le passage par valeur.

Ainsi, au retour dans le programme principal, n reste égale à 5.

### b) La transmission par adresse (ou par référence) :

Les adresses des paramètres effectifs sont transmises à la procédure appelante. Dans ce mode, le paramètre effectif subit les mêmes modifications que le paramètre formel lors de l'exécution du sous-programme

**Remarque :** le paramètre effectif doit être une variable (et non une valeur) lorsqu'il s'agit d'une transmission par adresse

Pour passer un paramètre par valeur en pseudo-code, on écrit le mot clé **variable** devant le nom du paramètre suivi de ":" et de son type.

#### Syntaxe

**Procedure** <nomproc>(**variable** <nomparam> : <type>);

**Début**

... ;

**Fin;**

#### Exemple

```
Algorithme ExempleParRef;
    Variable n : entier ;
    Procédure DoublerAfficher( variable k : entier ) ;
        Debut
            k ← 2*k ;
            Ecrire("Le double = ", k) ;
        Fin ;
        Début {début du programme principal ExempleValeur}
            Ecrire("Entrez un nombre : ") ;
            Lire(n) ;
            Ecrire(" n = ", n) ;
            DoublerAfficher (n) ;
            Ecrire(" n = ", n) ;
        Fin.
```

L'exécution de ce programme donne :

```
Entrez un nombre : 5
n = 5
Le double = 10
n = 10
```

La modification de k en  $2*k$  s'applique à la variable globale n utilisée comme paramètre effectif car utilise le passage par adresse. La modification faite dans le sous-programme reste permanente.

Ainsi, au retour dans le programme principal, n est devenue égale à 10.

**VI. LES ALGORITHMES DE TRI D'UN TABLEAU**

Le tri consiste à ordonner les éléments d'un tableau dans l'ordre croissant ou décroissant. Il existe plusieurs algorithmes pour trier les éléments d'un tableau.

**1) Tri par sélection**

**Principe**

Pour toutes les valeurs de l'indice  $i$ , à l'étape  $i$ , on sélectionne le plus petit élément parmi les  $(n - i + 1)$  éléments du tableau les plus à droite. On l'échange ensuite avec l'élément  $i$  du tableau.

9	4	1	7	3
---	---	---	---	---

- **Etape 1:** on cherche le plus petit parmi les 5 éléments du tableau. On le trouve en position 3, et on l'échange alors avec l'élément en 1 :

1	4	9	7	3
---	---	---	---	---

- **Etape 2:** on cherche le plus petit élément, mais cette fois à partir du 2ème élément. On le trouve en dernière position, on l'échange avec le deuxième:

1	3	9	7	4
---	---	---	---	---

- **Etape 3:** On échange 9 et 4.

1	3	4	7	9
---	---	---	---	---

- Aux étapes 4 et 5, l'échange se fait sur l'élément lui-même car 7 et 9 sont déjà à leurs places respectives.

Le tri est alors terminé.

## Algorithme du tri par sélection

Supposons que le tableau est noté T et sa taille N et les indices varient de 1 à N.

```
Pour i allant de 1 à N-1
début
    indice_ppe ← i ;
    Pour j allant de i + 1 à N
        Si T[j] < T[indice_ppe] alors
            indice_ppe ← j ;
        temp ← T[indice_ppe] ;
        T[indice_ppe] ← T[i] ;
        T[i] ← temp ;
Fin ;
```

## 2) Tri par insertion

### Principe:

- Le tableau T formé par les éléments ( $a_1, a_2, \dots, a_n$ ) est décomposé en deux parties :
  - une partie triée ( $a_1, a_2, \dots, a_k$ ) et
  - une partie non-triée ( $a_{k+1}, a_{k+2}, \dots, a_n$ );
  - l'élément  $a_{k+1}$  est appelé élément frontière (c'est le premier élément non trié).
- On prend l'élément  $a_{k+1}$  et on l'insère dans sa place dans la partie triée que l'on parcourt de  $k$  à 1.

## Algorithme du tri par sélection

```
Pour i de 2 à n Faire      {la partie non encore triée ( $a_i, a_{i+1}, \dots, a_n$ )}
début
    v ← Tab[ i ] ; { l'élément frontière :  $a_i$  }
    j ← i ;          { le rang de l'élément frontière }
    Tant que Tab[ j-1 ] > v Et j > 0 Faire {on est sur la partie triée ( $a_1, \dots, a_i$ )}
    début
        Tab[ j ] ← Tab[ j-1 ] ; { on décale l'élément}
        j ← j-1;           { on passe au rang précédent}
    Fin ;
    Tab[ j ] ← v {on recopie  $a_i$  dans la place libérée }
Fin ;
```

## 3) Tri bulles

### Principe:

- On admet qu'un tableau est trié s'il n'a pas deux éléments consécutifs non ordonnés.
- Ainsi, on parcourt le tableau du début à la fin et on intervertit les places de tout couple consécutif non ordonné.
- On refait le parcours jusqu'à ce que tout couple consécutif soit ordonné.

L'algorithme tri bulles sous forme de procédure

```
procédure Tri_Bulles (variable Tab : tableau[1..n]) ;  
    Variables i, j, v : Entiers ;  
        echange: booléen;  
    début  
        echange  $\leftarrow$  VRAI;  
        Tant que echange Faire  
            début  
                echange  $\leftarrow$  FAUX;  
                Pour i de 1 à n-1 Faire  
                    Si Tab[i]>Tab[i+1] Alors  
                        début  
                            v  $\leftarrow$  Tab[ i ] ;  
                            Tab[i]  $\leftarrow$  Tab[i+1];  
                            Tab[i+1]  $\leftarrow$  v;  
                            echange  $\leftarrow$  VRAI;  
                        Fin ;  
                    Fin ;  
                Fin ;  
            Fin ;
```

## VII. LE LANGAGE PASCAL

Le langage PASCAL, du nom du célèbre mathématicien de même nom, a été créé par WIRTH au début des années 70. Son avantage est qu'il représente de la façon la plus proche les notions algorithmiques de base.

En fait, c'est avec pascal qu'on a défini la méthode de programmation dite « programmation structurée »

Avec cette méthode, les instructions d'un programme sont exécutées une à une du haut vers le bas, guidée par les 5 structures de contrôles algorithmique que nous avons vu plus haut (instructions de base n°4 à 8).

Le PASCAL est un langage compilé, c'est à dire qu'il faut :

- entrer un texte dans l'ordinateur (à l'aide d'un programme appelé EDITEUR),
- le traduire en langage machine (c'est à dire en codes binaires compréhensibles par l'ordinateur) : c'est la compilation et éventuellement l'édition de liens (LINK),
- l'exécuter.

Actuellement, tout logiciel de développement de programme intègre ces 3 fonctions. C'est le cas d'Algopear, AlgExec, Turbo Pascal, et autres.

### 1. Structure générale d'un programme en pascal

Un programme pascal est composé de 3 parties : L'en-tête, la déclaration et le corps.

L'en-tête a la syntaxe :

**PROGRAM <nom\_progr> ;**

Où :

Program est un mot clé obligatoire .

<nom\_progr> est un identificateur (Ne plus écrire les chevrons <>) .

Le caractère « ; » est un mot clé obligatoire indiquant la fin d'un instruction en pascal.

La partie déclaration

Elle comprend la déclaration de : constante, type, variable, procédure et fonction.

Une constante est un identificateur auquel on a assigné une valeur. La syntaxe est :

**CONST <nom\_const> = <valeur> ;**

Exemple

Const pi = 3.1416 ;

Taux = 0.12 ;

La déclaration de variable est précédée du mot clé VAR, suivi par une liste de variable de même type séparée par « , » (virgule), puis du mot clé « : » puis du type de ces variables et terminé par « ; ».

On passe à la ligne suivante s'il y a une autre liste de variable.

La syntaxe est :

**VAR <liste\_var\_1> : <type1> ;  
          <liste\_var\_2> : <type2> ;  
          ... ;**

## Exemple

```
Var n, i, j : integer;  
      x, y : real;  
      nom, nat : string;
```

On vient de déclarer 3 variables n, i, j de type integer, 2 variables x, y de type real et 2 autres variables nom, nat de type string.

Nous aborderons plus tard les autres déclarations.

## Le corps du programme

C'est une suite d'instructions encadrées par BEGIN et END selon la forme :

```
BEGIN  
      <instructions>;  
END.
```

END est terminé par un « . »..

## Un programme pascal aura donc la forme :

```
Program <nom_prog>;  
Const <liste_const>;  
Var <liste_var>;  
Begin  
      <instructions>;  
End.
```

## 2. Types de variables

Les types de base en pascal sont :

- INTEGER pour entier.
- REAL pour réel . Exemples : 1.5 , 0.12 , 2.0 sont des real.
- BOOLEAN pour booléen, les valeurs sont TRUE (Vrai) et FALSE (Faux).
- CHAR pour caractère. Une valeur char est un caractère encadré par « ' » (le caractère apostrophe). Exemple : 'a', '+', ',', '9' sont des valeurs de type char.
- STRING pour chaîne de caractère. Une valeur de type string est une suite de caractères (éventuellement vide) encadrée par « ' ». Exemple : '' (apostrophe suivi de apostrophe pour la chaîne vide), 'a', 'azerty', 'une phrase' sont des constantes de type string.

## Les opérateurs

Chaque type de base est doté d'un ensemble d'opérateurs.

- Opérateur sur integer : +, -, \*, div (division division entière), mod (modulo ou le reste de la division). Ils donnent tous des résultats entiers, et nécessitent deux arguments entiers.
- Opérateurs sur les real : + - \* /
- Comparaison : 2 nombres, qu'ils soient integer ou real, sont comparables avec les opérateurs de comparaison qui sont : =, <, <=, >, >=, <> (différent).
- Opérateur de string : & (concaténation de 2 string).
- Opérateurs sur boolean : AND (et), OR (ou), NOT (NON)

### **3. L'affectation**

L'affectation se fait avec le mot := (« : » suivi de « = »). La syntaxe est :

<nom\_var> := <expression>;

#### **Exemples**

```
n := 0;  
long := 6;  
large := 3  
aire := long * large;  
nom := 'Rakoto';
```

Avec ces affectations, la valeur de l'expression à droite de := est placée dans la variable à gauche de :=.

#### **Remarque**

Si on a ;

```
A := 4;  
B := 6;  
C := B;  
A := A+B;
```

La dernière affectation signifie : « placer dans A l'actuelle valeur de A plus celle de B ». Après cette affectation, on aura donc A=10.

La 3ème met dans C la valeur 6 mais B garde également sa valeur 6. Ainsi, l'affectation n'a aucun effet sur la ou les variables de la partie droite.

Les affectations :

```
3 := 2+1;  
5 := B - 1;  
A+B := 10;
```

Sont toutes incorrectes car la partie gauche ne sont pas des variables mais des valeurs ou des expressions.

#### **REMARQUE**

L'affectation et l'a lecture sont les seules moyens possibles pour valoriser une variable. Donc, avant d'être utilisée dans une expression ou affichée, une variable doit subir une affectation ou être lue. On dit qu'on INITIALISE la variable.

## 4. Lecture et écriture

a) La lecture d'une valeur au clavier se fait par l'instruction READ OU READLN.

Syntaxe :

```
READ(<nom_var>) ;  
READLN'<nom_var> ;
```

READLN passe de curseur à la ligne suivante après la lecture d'une donnée. Cette lecture est terminée par la touche [Entrer].

On lit toujours une variable.

Exemple

```
Read(a) ;  
Readln(x) ;
```

Sont correctes si a et x sont des variables déclarées.

```
Read('a') ;  
Readln(5) ;  
Read(x+y) ;
```

Sont toutes incorrectes. Les 2 premières car elles tentent de lire des valeurs et non pas de variable et le troisième car elle tente de lire une expression au lieu d'une variable.

## b) Ecriture avec WRITE et WRITELN

Syntaxe

```
WRITE (<exprssion1>, <exprssion2>, ..., <exprssionN>) ;  
WRITELN (<exprssion1>, <exprssion2>, ..., <exprssionN>) ;
```

Les différentes <expressions> sont séparées par « , » (virgule) dès qu'il y en a plus d'une.

Une <expression> peut être :

- Une constante string ou chaine entre apostrophes. Exemple 'Bonjour !'.
- Une constante numérique : 6, 514, 3.14, ...
- N'importe quelle variable quelque soit son type.
- N'importe quelle expression valide.

WRITELN écrit et passe à la ligne suivante tandis que WRITE reste juste derrière le dernier caractère écrit.

Exemple de programme simple avec read et write.

```
Program rectangle ;  
Var long, large, surf : integer;  
Begin  
    Write('Entrez la longueur : ');  
    Readln(long);  
    Write('Entrez la largeur : ');  
    Readln(large);  
    surf := long * large ;  
    Writeln('La surface du rectangle ', long, ' x ', large, ' est ', surf) ;  
    Writeln('Le périmètre est ', (long+large)*2) ;  
End.
```

Voici un exemple d'exécution de ce programme.

```
Entrez la longueur : 4  
Entrez la largeur : 3  
La surface du rectangle 4x3 est 12  
Le perimetre est 14
```

Remarquez qu'on écrit sans caractères accentués. Ceux ci ne sont pas supportés par pascal.

Dans l'affichage de la surface, les expressions dans l'instruction writeln sont :

- La constante texte 'La surface du rectangle' .
- La variable long.
- La constante texte 'x' .
- La variable large.
- La constante texte ' est ' .
- La variable surf.

Dans l'affichage du périmètre, elles sont :

- La constante texte 'Le perimetre est ' .
- L'expression (long+large)/2.

## 5. La structure alternative

Elle réalise une structure SI ... ALORS .. SINON ....

Syntaxe en pascal

Forme 1 :

```
IF <condition> THEN  
    <instructions>;
```

Exécute <instructions> si <condition> vaut TRUE et saute <instructions> si <condition> vaut FALSE.

Si <instructions> est composée de plus d'une instruction (2 affectations par exemple), elle doit être encadrée par BEGIN et END ; (terminé par « ; »).

Exemple

```
IF n<0 THEN  
    BEGIN  
        x := x+1 ;  
        y := y + 1 ;  
    END ;
```

Forme 2 :

```
IF <condition> THEN  
    <instructions1>  
ELSE  
    <instructions 2>;
```

Exécute <instructions1> si <condition> vaut TRUE et <instructions 2> si <condition> vaut FALSE.

Si <instructions 1> est composée de plus d'une instruction (2 affectations par exemple), elle doit être encadrée par BEGIN et END mais le END ne doit pas être terminé par « ; ».

Si <instructions> est composée de plus d'une instruction (2 affectations par exemple), elle doit être encadrée par BEGIN et END ; (terminé par « ; »).

En aucun cas, il ne doit pas y avoir de « ; » devant ELSE.

Exemple

```
IF n<0 THEN  
    BEGIN  
        ValAbs := -n ;  
        Writeln(ValAbs) ;  
    END  
ELSE  
    BEGIN  
        ValAbs := n ;  
        Writeln(ValAbs) ;  
    END;
```

Forme 3 : IF imbriqués

On peut avoir des IF imbriquées aussi bien dans la clause THEN que dans la clause ELSE. Le IF imbriqué est un IF normal, donc peut prendre l'une des 3 formes de IF.

```
IF <condition1> THEN  
    IF <condition2> THEN  
        <instructions11>  
    ELSE  
        <instructions12>  
ELSE  
    IF <condition2> THEN  
        <instructions21>  
    ELSE  
        <instructions22>;
```

## Exemple :

Détermination du frais en ariary selon le type de voiture (1 ou 2) et la distance en km dans une zone suburbaine.

```
IF typevoiture = 1 THEN
    IF distance <20 THEN
        Frais := 1000
    ELSE
        Frais := 2000
ELSE
    IF distance <20 THEN
        Frais := 800
    ELSE
        Frais := 1500;
```

## 6. La structure de choix multiple

Le choix multiple se fait avec l'instruction CASE ... OF dont la syntaxe est :

```
CASE <nom_var> OF
    liste_de_cas1:
        instruction1;
    liste_de_cas2:
        instruction2;
    ....
    liste_de_casN:
        instructionN
END ;
```

Si dans un cas, les instructions dépassent une instruction, on doit les encadrer par begin ... end ;.

## Exemple

Un programme qui affiche le résultat d'un test de personnalité.

```
Program LirePoint ;
Var point : integer ;
Begin
    Write( 'Vous avez eu combien de point ? ' );
    Readln(point) ;
    Case point of
        0, 1, 2 :
            Writeln('Vous etes paresseux') ;
        3 :
            Writeln('Vous pouvez faire des efforts') ;
        4,5 :
            Writeln('Vous etes bon travailleur') ;
        Else
            Writeln('Vous avez mal calcule les points') ;
    End ;
End.
```

## 7. La boucle FOR ... DO

FOR ... est une version pascal de POUR ... FAIRE ...

Sa syntaxe est :

Version incrémentée (compteur monte par pas de +1)

```
FOR <compteur> := <borneinf> TO <borbesup> DO  
    <instructions>  
END ;
```

Version décrémentée (compteur descend par pas de -1)

```
FOR <compteur> := <bornSup> DOWNTO <borbeINF> DO  
    <instructions>  
END ;
```

### Fonctionnement :

- La valeur <borneinf> est affectée à la variable <compteur>;
- <instructions> est exécutée;
- <compteur> est **incrémentée**, c'est-à-dire, qu'on lui affecte son ancienne valeur + 1 ( comme si on a fait : compteur := compteur +1 );.
- Si <compteur> est encore <= <bornesup>, <instructions> est exécutée et on revient au tiret précédent.
- Si <compteur> est devenu > <bornesup>, on passe à l'instruction après FIN ;
- Si instructions dépasse une instruction, on doit les encadrer par begin ... end ;.

### Exemple :

Calcul de x à la puissance n où x est un réel non nul et n un entier positif ou nul

```
Program CalculPuissance ;  
Var x, puiss : real  
    n, i : integer  
Begin  
    Write(' Entrez la valeur de x : ') ;  
    Readln (x) ;  
    Write (' Entrez la valeur de n : ') ;  
    Readln (n) ;  
    puiss ← 1 ;  
    For i := 1 To n Do  
        puiss := puiss*x ;  
    Writeln (x, ' à la puissance ', n, ' est égal à ', puiss) ;  
End.
```

## REMARQUE

Il est **interdit de modifier la valeur** du compteur et des bornes à l'intérieur d'une boucle FOR. En effet, une telle action perturbe le nombre d'itérations prévu par la boucle FOR et donnera une erreur.

## 8. La boucle WHILE ... DO

WHILE ... DO est la version pascal de TANT QUE ... FAIRE...

### Syntaxe :

```
WHILE <condition> DO  
    <instructions>;
```

### Fonctionnement :

- <condition> (dite condition de contrôle de la boucle) est évaluée avant chaque itération
- si la condition est vraie, on exécute <instructions> (corps de la boucle), puis, on retourne tester la condition. Si elle est encore vraie, on répète l'exécution, ...
- si la condition est fausse, on sort de la boucle et on exécute l'instruction qui est après « ; » .
- Si instructions dépasse une instruction, on doit les encadrer par begin ... end ;

### Remarque :

- Le nombre d'itérations dans une boucle WHILE n'est pas connu au moment d'entrée dans la boucle. Il dépend de l'évolution de la valeur de condition
- Une des instructions du corps de la boucle doit absolument changer la valeur de condition de vrai à faux (après un certain nombre d'itérations), sinon le programme tourne indéfiniment et on aura ce qu'on appelle une boucle infinie.

### Premier exemple de boucle infinie :

```
i := 2;  
WHILE i > 0 DO  
    i := i+1;
```

Ici, on part de  $i=2$  puis on l'incrémente à chaque passage dans la boucle. Donc  $i$  restera à jamais  $>0$  et on aura une boucle infinie.

### Deuxième exemple de boucle infinie :

```
i := 1;  
WHILE i < 10 DO  
    Writeln(i);
```

Ici, on part de  $i=1$  puis on ne modifie pas cette valeur lors des passages dans la boucle. Donc  $i$  restera à jamais égal à 1 qui est  $< 10$  et on aura une boucle infinie.

## Exemple de boucle WHILE correcte :

Un algorithme qui détermine le premier nombre entier N tel que la somme de 1 à N dépasse strictement 100

```
Program PremierN;  
Var som, i : integer;  
BEGIN  
    i := 0 ;  
    som:= 0 ;  
    WHILE (som <=100)  
        BEGIN  
            i := i+1  
            som := som+i  
        END ;  
        WRITELN (' La valeur cherchée est N= ', i);  
Fin
```

## Deuxième version du même algorithme.

La différence se trouve au niveau de l'ordre des instructions et des valeurs initiales.

```
Program PremierN;  
Var som, i : integer;  
BEGIN  
    som := 0;  
    i := 1 ;  
    WHILE (som <=100)  
        begin  
            som := som + I;  
            i := i+1 ;  
        end;  
        Writeln(' La valeur cherchée est N= ', i-1);  
END.
```

Ici, la valeur du N recherchée est i-1 et non pas i comme dans la première version.

## Relation avec la structure FOR ...

Une boucle POUR peut toujours être transformée en WHILE et on aura :

```
compteur ← borneinf ;  
WHILE compteur <= bornsup DO  
BEGIN  
    Instructions ;  
    compteur ← compteur +1 ;  
END;
```

Par contre, il y a des boucles WHILE qu'on ne peut pas réaliser avec FOR.

Exemple :

```
write('Entrez B : ');
readln(B);
write('Entrez A : ');
readln(A);
WHILE A < (B+4)/2 DO
BEGIN
    write('C'est normal');
    write('Entrez B : ');
    readln(B);
    write('Entrez A: ');
    readln(A);
END;
writeln("La situation est devenue anormale");
```

Ici, la condition de boucle  $A < (B+4)/2$  n'exprime le parcours d'un intervalle par une variable. Donc, cette boucle ne peut pas être exprimée avec POUR.

### **9. La boucle REPEAT ... UNTIL**

C'est la version REPETER ... JUSQU'A de pascal.

Syntaxe

```
REPEAT
    instruction1;
    instruction2;
    ....;
    instructionN
UNTIL condition;
```

À la différence des autres structures, REPEAT n'a pas besoin de begin ... end car ses instructions sont déjà bien délimitées par repeat et until.

Exemple

Un algorithme qui détermine le premier nombre entier N tel que la somme de 1 à N dépasse strictement 100 (version avec répéter jusqu'à)

```
Program PremierNRep;
Var som, i : integer;
Begin
    som := 0;
    i := 0;
    REAPET
        i := i+1;
        som := som+i;
    UNTIL som > 100
    writeln (' La valeur cherchée est N= ', i);
End.
```

## VIII. LES TYPES DEFINIS PAR L'UTILISATEURS PASCAL

On peut créer des nouveaux types autres que les types de base prédéfinis par pascal. Cela se fait dans la partie déclaration, par le mot clé TYPE puis on passe à la définition de ce nouveau type.

Une fois défini, le type s'utilise, dans la partie déclaration de variable, comme n'importe quel type.

### 1. Les intervalles

On déclare un intervalle par :

**TYPE <nom\_type> = <borninf> .. <bornsup>;**

#### Exemple

TYPE AgeEtudiant = 16..60;

NomDisk = 'C'..'M';

#### REMARQUE:

On peut créer un intervalle uniquement à partir des types INTEGER, CHAR ou BOOLEAN.

### 2. Les ensembles

Un ensemble est une "collection" d'éléments de même type.

#### Déclaration d'un type ensemble

**TYPE <nom\_type> = (<elem1>, <elem2>, ..., <elemN>);**

Les éléments de l'ensemble sont énumérés exhaustivement entre parenthèses.

#### Déclaration d'une variable de type ensemble

**VAR <nom\_var> : SET OF <nom\_type>;**

#### Exemple

On veut représenter des vendeurs et leurs domaines d'action.

Voici la déclaration de type et de variable :

TYPE produits=(velos,motos,autos,accessoires);

VAR vendeur1,vendeur2 , vendeur3 : SET OF produits;

On "remplit" un ensemble (ou on affecte une valeur dans un ensemble) en donnant ses éléments entre crochets de la manière suivante :

vendeur1:=[velos, motos];

vendeur2:=[motos, accessoires];

L'ensemble vide est : []

On peut faire les opérations suivantes avec des ensembles :

- UNION : réalisé avec le symbole « + »:

vendeur3 := vendeur1+vendeur2 ;

En résultat, on aura vendeur3 =[velos,motos,accessoires]

- INTERSECTION : réalisé avec le symbole « \* »:

vendeur3 := vendeur1\*vendeur2 ;

En résultat, on aura vendeur3 =[motos]

- COMPLEMENT : réalisé avec le symbole « - »:

vendeur3 :=: vendeur1-vendeur2 ;

En résultat, on aura vendeur3 = [velos]

Et avec :

vendeur3 := vendeur2-vendeur1 ;

On aura vendeur3 = [accessoires]

On dispose également des opérateurs de comparaison avec les ensembles. Ce sont:

= ,  $\neq$  ,  $\leq$  (inclus) ,  $\geq$  (contenant).

L'opérateur IN teste l'appartenance d'un élément. Si X vaut motos, alors

X IN vendeur1 et

[motos] $\leq$ vendeur1

sont équivalents..

### 3. Les tableaux

En Pascal, la déclaration d'un tableau s'écrit :

**Var <nom\_tableau> : array [<indice\_min> .. <indice\_max>] Of <type\_eléments>;**

Où : <indice\_min> .. <indice\_max> est un intervalle (d'entier ou de char).

#### Exemple

```
Var Prenom : array [1..1000 ] Of String;
```

```
Age : array[1..1000] of [16..60];
```

Pour Age, les éléments sont des entiers de 16 à 60 inclus.

#### Accès aux éléments

Chaque élément d'un tableau est désigné par le nom du tableau suivi du numéro (indice) de l'élément entre crochet « [ ] ».

Avec les déclarations e l'exemple ci-dessus, on peut avoir :

```
Prenom[1] := 'Abela' ;
```

```
Age[1] := 18 ;
```

```
Writeln('le nom de l''etudiant num 1 est ',Prenom[1]) ;
```

```
Writeln('Et son age est de ', Age[1], ' ans ') ;
```

Ce qui affichera :

```
Le nom de l'etudiant num 1 est Abela  
Et son age est 18 ans
```

#### Parcours des éléments d'un tableau.

On peut parcourir indifféremment les éléments d'un tableau avec l'une des trois structures de boucle : FOR, WHILE, REPEAT. Il suffit de parcourir toutes les indices avec un variable.

#### Exemple avec FOR

```
For i :=1 to 1000 do
```

```
Begin
```

```
    Write('Entrez le nom du num ',i) ;
```

```
    Readln(Prenom[i]) ;
```

```
    Write('Entrez l''age du num ',i) ;
```

```
    Readln(Age[i]) ;
```

```
End;
```

Cette boucle fait la saisie de tous les prenoms et ages du 1<sup>er</sup> au 1000<sup>e</sup> élément.

## Exemple avec WHILE

```
i := 1;  
while i<=1000 do  
Begin  
    Write('Entrez le nom du num ',i);  
    Readln(Prenom[i]);  
    Write('Entrez l'age du num ',i);  
    Readln(Age[i]);  
    i := i+1;  
End;
```

## Exemple avec REPEAT

```
i := 1;  
REPEAT  
    Write('Entrez le nom du num ',i);  
    Readln(Prenom[i]);  
    Write('Entrez l'age du num ',i);  
    Readln(Age[i]);  
    i := i+1;  
UNTIL i>100;
```

## 4. Les enregistrements

### Déclaration de type enregistrement

```
Type <nom_struct> = record  
    <nomchamp1> : <typechamp1>;  
    <nomchamp2> : <typechamp2>;  
    ...;  
end;
```

### Déclaration de variable de type enregistrement

```
Var nomvar : nom_de_type;
```

### Désignation d'une information composante:

Chaque information composante est désignée (ou est accédée) par le nom de la variable suivi du caractère « . » (point) puis suivi du nom du champ.

Avec les déclarations ci-dessus, les champs sont :

```
nomvar.nomchamp1  
nomvar.nomchamp2
```

Exemple

```
Program UsePersonne ;
TYPE personne = RECORD
    Nom : STRING ;
    Age : INTEGER ;
END ;
VAR p1, p2 : personne ;
a,a1,a2 : INTEGER ;
BEGIN
    WRITE('En quelle annee sommes-nous ? ') ;
    READLN(a) ;
    WRITE('Entrez le nom de la personne 1 : ') ;
    READLN(p1.nom) ;
    WRITE('Entrez l'age de la personne 1 : ') ;
    READLN(p1.age) ;
    WRITE('Entrez le nom de la personne 2 : ') ;
    READLN(p2.nom) ;
    WRITE('Entrez l'age de la personne 1 : ') ;
    READLN(p2.age) ;
    a1 = a - p1.age ;
    a2 = a - p2.age ;
    WRITELN(p1.nom, ' est née en ', a1) ;
    WRITELN(p2.nom, ' est née en ', a2) ;
    IF p1.age > p2.age THEN
        WRITELN(p1.nom, ' est l'aîné de ', p2.nom)
    ELSE
        IF p1.age > p2.age THEN
            WRITELN(p1.nom, ' est le cadet de ', p2.nom)
        ELSE
            WRITELN(p1.nom, ' est de même âge que ', p2.nom) ;
    END.
```

Voici 3 exemples d'exécution différente de ce programme.

p1 plus âgé que p2.

```
En quelle annee sommes-nous ? 2015
Entrez le nom de la personne 1 : Naivo
Entrez l'age de la personne 1 : 28
Entrez le nom de la personne 2 : Ranivo
Entrez l'age de la personne 2 : 25
Naivo est née en 1987
Ranivo est née en 1990
Naivo est l'aîné de Ranivo
```

p1 moins âgé que p2.

```
En quelle année sommes-nous ? 2015
Entrez le nom de la personne 1 : Beby
Entrez l'âge de la personne 1 : 22
Entrez le nom de la personne 2 : Fanja
Entrez l'âge de la personne 2 : 25
Beby est née en 1993
Fanja est née en 1990
Beby est le cadet de Fanja
```

p1 de même âge que p2.

```
En quelle année sommes-nous ? 2015
Entrez le nom de la personne 1 : Jao
Entrez l'âge de la personne 1 : 22
Entrez le nom de la personne 2 : Koto
Entrez l'âge de la personne 2 : 22
Jao est née en 1993
Koto est née en 1993
Jao est de même âge que Koto
```

### IX. LES SOUS-PROGRAMMES PASCAL

On peut regrouper un ensemble d'instructions sous un même nom. On forme alors un sous-programme. On utilise un sous-programme :

- chaque fois qu'une même suite d'instructions doit être répétée plusieurs fois dans un programme,
- quand une suite d'instruction forme une action globale. Le programme est alors plus clair et les erreurs plus facilement détectables.

En Pascal, il y a 2 types de sous-programme : les procédures et les fonctions.

#### 1) Procédure

Une procédure est un bloc de programme. Elle se compose d'un en-tête et d'un corps. Elle est définie dans une section appropriée de la partie déclaration du programme principal et son appel ne peut précéder sa définition.

La définition d'une procédure comprend successivement les parties suivantes:

- L'en-tête de procédure introduite par le mot-clé **procedure**, suivi du nom de la procédure et se termine par la liste des paramètres entre () .
- La déclaration des variables de la procédure qu'on appelle variables locales contrairement à celles du programme principal qu'on appelle variables globales.
- Le corps de la procédure encadré par « **begin ... end ;** ».

Voici la syntaxe de définition d'une procédure :

```
procedure <nom_proc>(<liste_param>) ;  
  <déclarations contantes, types, variables et sous-programmes>;  
  Begin  
    <instructions>;  
  End ;
```

### Appel d'une procédure

Il se fait dans le programme appelant en utilisant le nom de la procédure, suivi éventuellement de ses paramètres, comme une instruction pascal.

Exemple de programme avec définition et appel d'une procédure.

```
Program GererTableau ;  
Var t=array[1..10] of integer;  
  i, rep : integer;  
Procedure AfficheMenu ;  
Begin  
  Writeln (' Menu général' );  
  Writeln ('1 - Saisir les données' );  
  Writeln ('2 - Afficher les données' );  
  Writeln ('3 - Rechercher une information' );  
  Writeln ('0 - Quitter' );  
End ;  
Begin {Début du programme principal GererTableau}  
  Repeat  
    AfficheMenu ; {appel de la procédure}  
    Readln(rep);  
    Until rep = 0 ;  
    Writeln('Au revoir') ;  
End.
```

## 2) Fonction

L'en-tête de fonction a la même forme que l'en-tête d'une procédure mais

- est introduit par le mot-clé function,
- se termine par la mention du type de la fonction ; seuls sont autorisés les types simples (entiers, réels, booléens, caractères et les chaînes).

Dans le corps de la fonction, le renvoie du résultat de la fonction s'effectue par **affectation de l'expression résultat au nom de la fonction**.

Voici la syntaxe de définition d'une fonction:

```
function <nom_proc>(<liste_param>) : <type_résultat>;  
  <déclarations contantes, types, variables et sous-programmes>;  
  Begin  
    <instructions>;  
    <nom_proc> := <expression du resultat>;  
  End ;
```

Exemple: fonction qui calcule le pgcd de 2 entiers

```
Function pgcd ( a ,b : Integer ) : Integer ;
Var r : Integer ;
Begin
    While b<>0 Do
        Begin
            r := a Mod b ;
            a := b ;
            b := r
        End;
        pgcd := a      {retour de la fonction}
    End ;
```

### Appel d'une fonction

L'appel d'une fonction se fait dans le programme principal en utilisant la fonction dans une expression. Pour cela, on écrit le nom de la fonction avec ses paramètres ou arguments, comme un terme d'une expression.

#### Exemple

Si on a défini une fonction f avec un paramètre réel et renvoyant un nombre réel, on peut écrire l'affectation :

y := 2 \* f(x) + 1 ;

où y et x sont des variables réelles. f a été définie par :

```
function f(x :real) :real;
begin
    f := (x-1)*x;
end;
```

### Exemple de programme complet avec fonction et procédure.

```
Program DeuxNombres ;
Var m, n, rep : integer;
Function pgcd ( a ,b : Integer ) : Integer ;
Var r : Integer ;
Begin
    While b<>0 Do
        Begin
            r := a Mod b ;
            a := b ;
            b := r
        End;
        pgcd := a      {retour de la fonction}
    End ;
Function LePlusGrand(a,b :integer) : integer;
Begin
    if a>b then
        LePlusGrand := a
    else
        LePlusGrand := b;
End;
```

```
Procedure menu ;
Begin
    Writeln (' Menu général' );
    Writeln ('1 - Saisir les données' );
    Writeln ('2 – Afficher le plus grand' );
    Writeln ('3 – Afficher le pgcd' );
    Writeln ('0 - Quitter' );
    Write(' Choix : ');
End;

Begin {début du corps de DeuxNombres}
    Repeat
        menu;
        readln(rep) ;
        case rep of
            1 :
                Begin
                    Write('Entrez m :');
                    Readln(m);
                    Write('Entrez n :');
                    Readln(n);
                End;
            2 :
                Writeln('Le plus grand est ', LePlusGrand(m,n) );
            3 :
                Writeln('Le pgcd est ', pgcd(m,n) );
        End ;
        Until rep =0 ;
        Writeln(' Merci et au revoir') ;
End.
```

Voici un exemple d'exécution de ce programme

```
menu général
1 - Saisir les données
2 - Afficher le plus grand
3 - Afficher le pgcd
0 - Quitter
Choix : 1
```

Entrez m : 15  
Entrez n : 18

```
menu général
1 - Saisir les données
2 - Afficher le plus grand
3 - Afficher le pgcd
0 - Quitter
Choix : 2
```

Le pgcd est 18

```
menu général
1 - Saisir les données
2 - Afficher le plus grand
3 - Afficher le pgcd
0 - Quitter
Choix : 3
```

```
menu général
1 - Saisir les données
2 - Afficher le plus grand
3 - Afficher le pgcd
0 - Quitter
Choix : 0
Merci et au revoir
```

### 3) Passage de paramètre par valeur

Deux notions sont à bien distinguer :

- Le paramètre formel sur lequel le sous-programme (procédure ou fonction) appelé va travailler.
- Le paramètre effectif qui est une variable du programme principal et avec lequel ce dernier va appeler le sous-programme.

Dans le mode de passage par valeur, le paramètre transmis au sous-programme appelé est la valeur de l'argument. C'est une copie locale et non pas le paramètre formel original, il n'y a pas de risque de modification de la variable globale.

Dans ce mode de transmission, en Pascal, le paramètre formel ,dans la définition du sous-programme, n'est pas précédé du mot-clé Var .

#### Exemple

```
Program ExempleAvecParametreTransmisParValeur ;  
  
Procedure Ajouter ( L : Integer ) ;  
Begin  
    L := L + 1 ;  
    Writeln ( 'Resultat dans procedure : ' , L )  
End ;  
  
var G : Integer ;  
  
Begin  
    G := 27 ;  
    Ajouter ( G ) ;  
    Writeln ( 'Resultat dans programme : ' , G )  
End.
```

Ce programme produit à l'exécution les résultats suivants :

Resultat dans procedure : 28 Resultat dans programme : 27
--

La variable globale nommée G n'est pas modifiée ; le mécanisme est le suivant :

- Avant l'appel, la variable G vaut 27 et la variable L n'existe pas ( puisqu'elle n'est pas mentionnée dans la partie déclaration du programme principal ).
- A l'appel de la procédure, la variable L est créée dynamiquement et on lui affecte la valeur de la variable G, à savoir 27. Au début de la procédure, L vaut donc 27.
- Pendant le déroulement de la procédure, la variable L est modifiée ( l'affichage de L le prouve ), mais ceci n'affecte en rien la variable G.
- Après la sortie de la procédure, lorsque l'on retourne au programme principal, la variable G n'a pas changé et vaut donc toujours 27 lorsqu'on la réécrit sur l'écran. La variable L n'existe plus ; elle a été "volatilisée" à la sortie de la procédure puisqu'elle était interne à celle-ci.

## 4) Passage de paramètre par adresse ou par référence

Il existe en Pascal une autre manière de transmettre les paramètres : le passage par référence. Dans ce mode, il n'y a pas de recopie locale de l'argument mais le sous-programme appelé travaille directement sur l'original et les modifications effectuées sur les paramètres formels sont répercutées aux arguments. Les variables correspondantes du bloc appelant sont donc modifiées.

C'est dans la définition du sous-programme qu'on précise le mode de transmission par référence en précédant le nom du paramètre formel par le mot réservé **Var**.

Au niveau de l'appel, il n'y a aucune différence entre les passages par référence et par valeur.

```
Program ExempleAvecParametreTransmisParReference ;
```

```
Procedure Ajouter (Var L: Integer) ;
Begin
    L := L + 1;
    Writeln ( 'Resultat dans procedure : ', L )
End;

Var G : Integer ;

Begin
    G := 27;
    Ajouter ( G );
    Writeln ( 'Resultat dans programme : ', G )
End.
```

```
Resultat dans procédure : 28
Resultat dans programme : 28
```

La variable G est donc modifiée dès que l'on touche au paramètre L de la procédure.  
Le processus en est le suivant:

- Avant l'appel, la variable G vaut 27 et la variable L n'existe pas (puisque n'est pas mentionnée dans la partie déclaration du programme principal).
- A l'appel de la procédure, la variable L est créée mais ce n'est pas une "vraie" variable : il s'agit en fait d'une manipulation qui consiste à répercuter tout ce qui concerne la variable L sur la variable G. En d'autres termes, pour les connaisseurs, la variable L est un pointeur contenant l'adresse de la variable G.

Ces deux variables sont équivalentes et toute modification de la variable L sera en réalité une modification de la variable G. Tout se passe comme si on avait simplement redonné un nouveau nom à la variable G. La validité de ce nouveau nom est restreinte à la procédure.

## X. LES TRIS EN PASCAL

### 1) Tri par sélection

Cette procédure utilise un paramètre transmis par référence qui est le tableau à trier. Comme il est interdit d'utiliser une variable non valorisée N dans la déclaration du tableau, nous avons choisi 10 pour l'exemple mais on peut utiliser n'importe quelle taille du tableau.

```
Procedure TriSelection(var T : array[1..10] of integer);
Var i, j, indice_ppe, temp : integer;
begin
    For i := 1 To N-1 Do
        begin
            indice_ppe := i ;
            For j := i + 1 To N Do
                If T[j] < T[indice_ppe] Then
                    indice_ppe := j ;
                temp := T[indice_ppe] ;
                T[indice_ppe] := T[i] ;
                T[i] := temp ;
        end ;
    end ;
```

Lors de l'appel, si tab est un array[1..10] of integer déjà rempli, alors l'instruction qui va trier tab en appellant cette procédure est :

```
TriSelection(tab) ;
```

Il en est de même pour les deux méthodes suivantes.

### 2) Tri par insertion

```
Procedure TriInsertion(var T : array[1..10] of integer);
Var i, j, v : integer;
begin
    For i := 2 To 10 Do
        Begin
            v := T[i] ;
            j := i ;
            While (Tab[j-1] > v) AND (j > 0) Do
                begin
                    Tab[j] := Tab[j-1] ;
                    j := j-1 ;
                End ;
            Tab[j] := v ;
        End ;
    End .
```

3) Tri bulle

```
procedure TriBulle (var Tab : array[1..10] of integer);
  Var i, j, v : integer;
      echange: boolean;
begin
  echange := true;
  While echange Do
  begin
    echange := False;
    For i := 1 To 9 Do
      If Tab[i]>Tab[i+1] Then
      begin
        v := Tab[ i ];
        Tab[i] := Tab[i+1];
        Tab[i+1] := v;
        echange := True;
      End;
    End ;
  End ;
```

Remarquez qu'ici, la boucle For n'a pas besoin de Begin ... End car elle ne contient qu'une seule instruction : l'instruction IF.