

# ALGORITHMIQUE

Support de cours

afpa ©	auteur	centres	lecteur	formation	module	séq/item	type doc	millésime	page 1
	JC.Corre	Grenoble Nancy	B Manet	APII	1		sup. cours	01/96- v1.0	ALGOSC.DOC

## Sommaire

<b>1. GENERALITES SUR L'ALGORITHMIQUE.....</b>
<b>1.1 INTRODUCTION.....</b>
<b>1.2 NOTIONS D'OBJETS ET D'ACTIONS.....</b>
1.2.1 Les objets.
1.2.2 les actions.
<b>1.3 STRUCTURE GENERALE D'UN ALGORITHME.....</b>
<b>2. : LE LANGAGE ALGORITHMIQUE.....</b>
<b>2.1 LES ELEMENTS DU LANGAGE ALGORITHMIQUE.....</b>
2.1.1 Caractères utilisés.
2.1.2 Mots réservés et symboles.
2.1.3 Identificateurs.
2.1.4 Commentaires.
<b>2.2 LE TYPE DES DONNEES ET LEURS OPERATEURS.....</b>
2.2.1 Les entiers.
2.2.2 Les caractères.
2.2.3 Les booléens.
2.2.4 Les réels.
2.2.5 Les tableaux.
2.2.6 Les types énumérés.
2.2.7 Les enregistrements.
2.2.8 Les pointeurs.
2.2.9 Types non nommés.
<b>2.3 INSTRUCTIONS.....</b>
2.3.1 Instruction d'affectation.
2.3.2 Instructions alternatives.
2.3.3 Instructions répétitives.
<b>2.4 OUTILS D'ENTREE/SORTIE.....</b>
2.4.1 Les entrées/sorties à la console.
2.4.2 Les entrées/sorties fichier.
<b>2.5 PROCEDURES ET FONCTIONS.....</b>
2.5.1 Les procédures.
2.5.2 Les fonctions.
<b>2.6 STRUCTURE DE PROGRAMME.....</b>
2.6.1 Structure Générale d'un programme.

# 1 GÉNÉRALITÉS SUR L'ALGORITHMIQUE.

## 1.1 INTRODUCTION.

Vous voilà face à votre client et à son problème. La première étape consiste à vous mettre d'accord avec le client sur le travail à fournir. Il est notoire qu'une fois le problème bien compris, un pas décisif vers la solution est fait dans la mesure où il existe une solution informatique au problème posé. Cette phase est **l'analyse fonctionnelle**.

L'étape suivante consiste à concevoir l'application. Cela veut dire modéliser l'application, la décomposer de manière descendante, et mettre en place les moyens qui permettront de trouver une solution au problème posé. Cette phase est la **conception préliminaire**.

La phase suivante est celle qui vous intéresse ici : la **conception détaillée**. Il s'agit d'exprimer de manière détaillée comment résoudre les différentes difficultés rencontrées. Pour cela il existe plusieurs formalismes :

- Graphiques : Organigrammes, réseaux de Pétri, Grafcet.
- Textuels : Algorithmes, recettes, rapports.

Chaque formalisme a ses particularités, et répond bien à un type de problème donné. L'algorithmique s'adapte bien aux problèmes informatiques étudiés par analyse descendante. Son formalisme a été repris dans un grand nombre de langages structurés comme le C, le Pascal, l'Ada...

**L'algorithme** va permettre d'exprimer comment résoudre les problèmes, en se centrant sur la nature du travail, c'est à dire sans soucis des spécificités dues à la machine ou au langage de programmation.

L'algorithmique peut se définir suivant quatre axes :

- La définition des objets que l'on va manipuler.
- L'utilisation des objets définis (les actions).
- La présentation des algorithmes (les commentaires et l'indentation).
- L'esprit dans lequel les algorithmes sont construits.

afpa ©	auteur	centre	lecteur	formation	module	séq/item	type doc	millésime	page 3
	JC.Corre	Grenoble Nancy	B Manet	APII	1		sup. cours.	12/16 - v1.0	ALGOSC.DOC

Ce document aborde les problèmes de formalisme, et montre la manière de définir des objets et de les manipuler. Il vous montre aussi comment rendre compréhensible un programme en mettant des commentaires **utiles** c'est à dire judicieusement choisis aux endroits critiques.

La présentation d'un algorithme sera vue de manière induite par les différents exemples abordés. L'esprit dans lequel les problèmes d'algorithmique seront abordés, fait l'objet principal du module de programmation. Ce point n'est donc pas traité dans le présent document.

L'algorithme terminé, il reste la phase de codage à réaliser. La traduction de cet algorithme en un programme se fait de manière quasiment automatique, sauf pour les points faisant appel aux spécificités de la machine ou du langage de programmation utilisé. Le programmeur a besoin de toute son énergie et de toute sa concentration pour passer ces difficultés, ainsi l'algorithme a permis de traiter séparément les problèmes dus à conception du produit de ceux dus à son implémentation. C'est pourquoi l'algorithmique est une étape nécessaire à la réalisation d'applications d'informatique industrielle.

En observant les mêmes règles de style dans la programmation que dans l'algorithme, le programmeur aboutit à une application conviviale et maintenable (c'est à dire compréhensible et modifiable par une tierce personne) à condition que les documents des phases antérieures soient complets et avenants.

## 1.2 NOTIONS D'OBJETS ET D'ACTIONS.

### 1.2.1 Les objets.

Les objets forment l'ensemble des éléments qui sont manipulés dans un algorithme.

Il y a différents types d'objets :

- Les types.
- Les constantes.
- Les variables.
- Les procédures et fonctions.

#### 1.2.1.1) Les types

En algorithmique il est obligatoire de classer les objets dans des familles. Ces familles s'appellent des types. Deux objets de la même famille seront interchangeables.

afpa ©	auteur	centres	lecteur	formation	module	séq/item	type doc	millésime	page 4
	JC.Corre	Grenoble Nancy	B Manet	APII	1		sup. cours.	12/16 - v1.0	ALGOSC.DOC

Un type est défini par un nom (ou identificateur) et une référence à des types connus (c'est à dire des types prédéfinis dans le langage algorithmique, ou des types construits à partir de ces types). Il est donc possible de définir des types par rapport à des types existants.

Un type est caractérisé par :

- L'ensemble des valeurs que les objets de ce type peuvent prendre.
- L'ensemble des actions que l'on peut faire sur les objets de ce type.

Le rôle d'un type est de permettre de classer les objets dans des familles et de permettre ainsi des manipulations uniquement entre objets d'une même famille. Pour employer une image, le type enlève au programmeur la possibilité "d'enfoncer une vis avec un marteau".

### ***1.2.1.1 Les constantes.***

Elles sont définies par un identificateur, et par une valeur. Le nom représente la manière de faire référence à la valeur. La valeur représente le contenu de notre constante. Cette valeur est **invariante**.

Une constante a un type, qui est défini par le type de la valeur qui lui est associée.

Le rôle d'une constante est de noter des repères, des dimensions, des références invariants au cours d'un programme.

### ***1.2.1.2 Les variables.***

Elles sont définies par un identificateur pour pouvoir les référencer, par un type pour savoir à quelle famille elles appartiennent (donc quelles opérations on peut faire dessus), et un contenu, c'est à dire l'information qu'elles contiennent.

Le rôle de la variable est de stocker les valeurs de certaines informations pour pouvoir les relire, les comparer, les modifier au cours d'un programme.

### ***1.2.1.3 Les procédures et fonctions.***

Ce sont des outils qui seront définis par le programmeur (un marteau par exemple). Puis il définit sur quels types d'objets travaille cet outil (sur des clous pour notre marteau).

Pour les fonctions il donne en plus le type de l'information calculée par la fonction (sinus est une fonction qui, quand nous lui donnons un réel, rend un réel).

Une procédure ou une fonction est définie par un identificateur qui permet de l'appeler, les types des objets qu'elle va manipuler (les paramètres), les actions qu'elle effectue sur les objets qui lui seront donnés à l'appel de la procédure (bien faire la différence entre " un marteau enfonce des clous " et "j'enfonce ce clou avec le marteau ").

La fonction est en plus définie par le type de la valeur qu'elle a calculée.

afpa ©	auteur	centre	lecteur	formation	module	séq/item	type doc	millésime	page 5
	JC.Corre	Grenoble Nancy	B Manet	APII	1		sup. cours.	12/16 - v1.0	ALGOSC.DOC

### 1.2.2 les actions.

Les actions sont toutes les opérations qui pourront être réalisées sur les objets définis dans le programme. Plusieurs catégories d'actions seront distinguées.

Les actions d'observation permettent de comparer deux objets de même type. Sont-ils égaux? L'un est-il plus grand que l'autre? Cette dernière action ne peut se faire que si le type est ordonné, c'est à dire que l'on peut classer les objets de ce type du plus grand au plus petit.

Les actions de modification donnent une valeur à une variable. Cette valeur peut être celle d'une autre variable, une constante, le résultat de l'appel d'une fonction, ou le résultat d'opérations entre plusieurs de ces objets.

Les actions alternatives permettent d'effectuer des actions (de quelque catégorie que ce soit) suivant certaines conditions.

Les actions répétitives permettent d'itérer des actions. C'est la réalisation d'une condition qui permet de mettre fin à cette itération. Notons que si les actions qui sont réalisées dans l'action répétitive ne modifient pas la condition de terminaison, notre algorithme est faux.

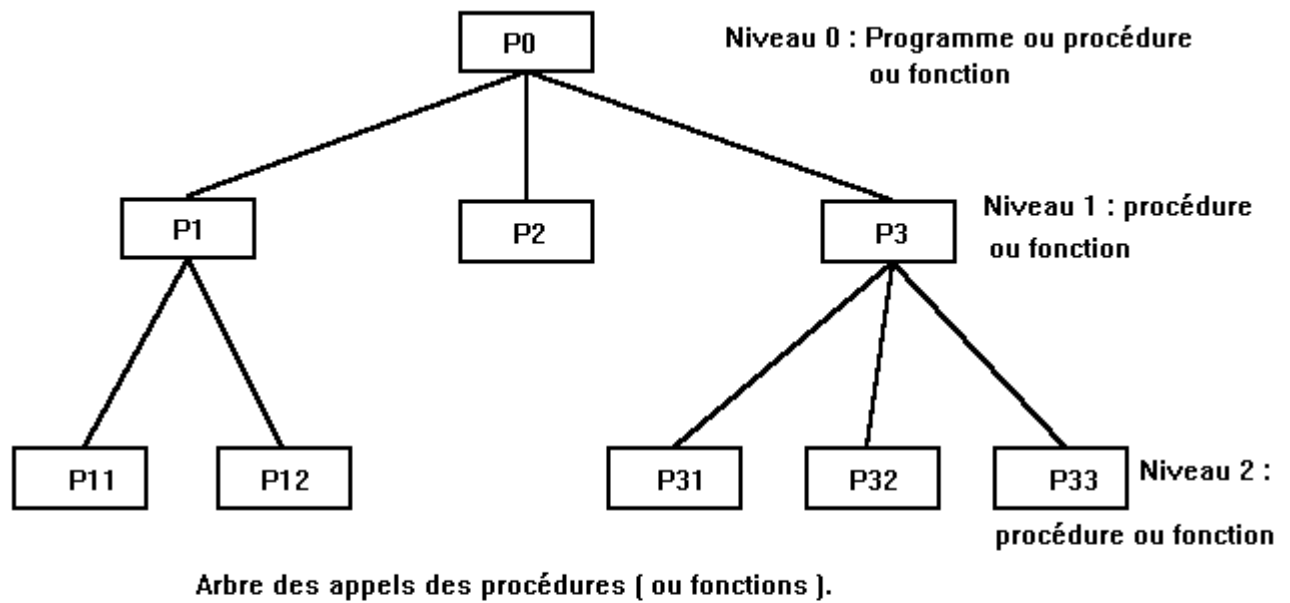
Les actions complexes représentent l'appel d'une procédure. C'est donc le déroulement des actions comprises dans la définition d'une procédure.

Bien sûr ces notions seront développées largement dans le formalisme de l'algorithmique

### 1.3 STRUCTURE GÉNÉRALE D'UN ALGORITHME.

L'algorithme est l'aboutissement d'une analyse, puis d'une conception descendante. Il peut être représenté par un graphe arborescent, où chaque case représente soit le programme principal (pour la racine uniquement) soit une procédure, soit une fonction.

afpa ©	auteur	centres	lecteur	formation	module	séq/item	type doc	millésime	page 6
	JC.Corre	Grenoble Nancy	B Manet	APII	1		sup. cours.	12/16 - v1.0	ALGOSC.DOC



Voici la structure générale que va avoir un bloc (qu'il soit programme, procédure ou fonction).

```

// définition des objets du bloc

Interface de bloc      // programme procédure ou fonction

// commentaire sur le rôle du bloc

constantes          // définition des constantes du bloc

types              // définitions des types du bloc

variables         // définition des variables du bloc

procédures ...        // définitions des procédures appelées par le bloc

fonctions ...         // définition des fonctions appelées par le bloc

// Définition des actions du bloc

début
  
```

actions	// actions qui seront effectuées
	// lors de l'activation du bloc
<b><u>fin</u></b>	

Remarque : la notation // bla-bla désigne les commentaires présents dans l'algorithme.

Voici le résultat de cette structure de bloc appliquée à l'arbre qui a été défini précédemment:



programme P0 ...  
constantes ...  
types ...  
variables ...

procédure P1 ...  
constantes ...  
types ...  
variables ...

procédure P11 ...  
constantes ...  
types ...  
variables ...  
// il n'y a pas de procédure ou fonction ici  
début  
    actions de P11  
fin

procédure P12 ...  
constantes ...  
types ...  
variables ...  
// il n'y a pas de procédure ou fonction ici  
début  
    actions de P12  
fin

début  
    actions de P1  
fin

procédure P2 ...  
constantes ...  
types ...  
variables ...  
// il n'y a pas de procédure ou de fonction ici  
début  
    actions de P2  
fin

**procédure** P3 ...

**constantes** ...

**types** ...

**variables** ...

```
procédure P31
constantes ...
types ...
variables ...
// il n'y a pas de procédure ou fonction ici
début
    actions de P31
fin
```

```
procédure P32 ...
constantes ...
types ...
variables ...
// il n'y a pas de procédure ou fonction ici
début
    actions de P32
fin
```

```
procédure P33 ...
constantes ...
types ...
variables ...
// il n'y a pas de procédure ou fonction ici
début
    actions de P33
fin
```

```
début
    actions de P3
fin
```

```
début
    actions de P0
fin
```

Cette manière de faire est très fidèle à l'esprit de conception qui a été défini, mais induit une difficulté de relecture des algorithmes pour savoir quelles actions se rapportent à quelles définitions, car les actions peuvent être séparées de leur définition (C.F. les définitions des procédures P1 et P3).

D'autre part ceci ne respecte pas l'approche descendante des problèmes. En effet il faut d'abord complètement décrire le problème de plus haut niveau puis décrire les problèmes de niveau suivant puis ainsi de suite.

Il sera ultérieurement étudié comment garder le principe de l'arborescence, tout en décrivant chaque bloc de manière monolithique.

Un bloc est une boîte étanche; un objet défini dans un bloc n'est pas connu dans les blocs extérieurs à ce bloc. Mais il est connu dans le bloc et dans tous les blocs contenus dans ce bloc.

Les règles détaillées sont précisées dans ce document.

## 2: LE LANGAGE ALGORITHMIQUE.

### 2.1 LES ÉLÉMENTS DU LANGAGE ALGORITHMIQUE.

Les différents mots du langage algorithmique sont définis ci-après. Il s'agit des éléments de base du langage à partir desquels seront construits les algorithmes.

#### 2.1.1 Caractères utilisés.

Les majuscules                      A ... Z

Les minuscules                    a ... z

Les chiffres                        0 ... 9

Les signes                         \_ = < > ' ( ) [ ] \* + - / , : . Espace

Aucune distinction entre les majuscules et les minuscules n'est faite.

#### 2.1.2 Mots réservés et symboles.

Les mots réservés sont les mots prédéfinis du langage algorithmique. Ces mots ne pourront pas être employés pour définir d'autres objets du langage.

alors	autrecas	booléen	caractère
choix	constantes	créer	de
début	détruire	div	écrire
enregistrement	entier	entrée	et
faire	faux	fermer	fichier
fin	finchoix	finenregistrement	finfichier
finsi	fintantque	fonction	indexé
jusqu'à	lire	mod	non
null	ou	ouvrir	pointeur
positionner	procédure	programme	quelconque
réel	répéter	retourner	si
sinon	sortie	sur	tableau
tantque	types	variables	vrai

Les symboles prédéfinis sont des compositions de signes qui ont un sens particulier dans le langage algorithmique.

:=      <>      >=      <=      //      ->

### **2.1.3 Identificateurs.**

Les identificateurs sont les noms qui seront donnés aux différents objets déclarés dans un algorithme. Comment sont-ils définis?

- C'est une suite de chiffres, de lettres (minuscules ou majuscules) et de soulignés, de longueur quelconque, commençant forcément par une lettre.

Exemple:

Ad98

nb5667kjl\_dif

WXCXCXC

r4567

g\_t\_s

- On ne fait pas la distinction entre les majuscules et les minuscules.

Exemple :

table\_mesure, Table\_Mesure, TABLE\_mesure représentent le même identificateur.

- Les identificateurs sont discriminants sur l'ensemble des caractères les formant.

Exemple :

table\_des\_mesures\_hautes et table\_des\_mesures\_haute sont deux identificateurs différents.

- Les identificateurs sont différents des mots réservés du langage algorithmique.

Exemple :

fin ne représente jamais autre chose que la fin d'un programme ou d'une procédure (ou fonction).

### **2.1.4 Commentaires.**

Un commentaire dans un algorithme commence par 2 barres inclinées (//).

Ce qu'il y a dans un commentaire ne change rien au déroulement de notre algorithme. Par contre les commentaires seront précieux pour exprimer le rôle de chaque objet défini dans notre algorithme, et pour expliquer le sens général de nos traitements.

Les commentaires se placent entre deux mots quelconques du langage algorithmique, ou en fin de ligne, ou sur une ligne. Un commentaire n'est jamais placé sur plusieurs lignes.

Une étude sur la qualité des algorithmes dit qu'un algorithme doit comporter au minimum 50% de commentaires **utiles**. Un commentaire utile est un commentaire qui apporte une réelle information sur la nature d'un objet ou d'un traitement; les

commentaires ne doivent pas paraphraser le code, mais apporter des éclaircissements sur celui-ci.

## 2.2 LE TYPE DES DONNÉES ET LEURS OPÉRATEURS.

La définition d'un type de données permet de définir une famille d'objets qui prendront tous leurs valeurs dans le même domaine, qui auront les mêmes propriétés, et qui auront le même comportement quand ils seront utilisés.

Ainsi quand une variable est définie comme un entier, il est sous-entendu qu'elle a les valeurs des entiers signés, et qu'elle respecte les propriétés et les opérations des entiers.

Exemple :

nombre : entier // définition d'un entier

nombre + 12 // nombre peut être utilisé

// comme un entier.

Deux objets qui sont du même type seront interchangeables du point de vue syntaxique. Tout ce qui peut être fait à l'un, peut être fait à l'autre. Par contre cela n'a pas toujours de sens du point de vue de la sémantique.

### 2.2.1 Les entiers.

Les entiers ne seront pas limités par leur taille. Les constantes entières seront toujours exprimées en base 10.

Exemple :

856 98877665567786543 12 sont des entiers.

Les opérateurs possibles sur les entiers sont les suivants :

- Les opérateurs de comparaison = > < >= <= <>
- Les opérateurs de calcul + - \* **div** **mod**

Les opérateurs de comparaison ont pour opérandes deux entiers et donnent un résultat booléen. L'opérateur <> est l'opérateur différent, les autres opérateurs ayant leur signification habituelle.

Les opérateurs de calcul ont pour opérandes deux entiers et donnent un résultat entier. L'opérateur **div** est la division entière, l'opérateur **mod** est le reste de la division entière, les autres opérateurs ayant leur signification habituelle.

Exemples :

$$7 \text{ div } 3 = 2$$

$$7 \text{ mod } 3 = 1$$

L'ordre de priorité des opérateurs de calcul est l'ordre habituel d'évaluation des opérateurs, à savoir que \*, **mod**, **div** sont plus prioritaire que + et -.

Exemple :

$$2 + 3 * 4 = 14$$

Si l'ordre d'évaluation de l'expression doit être forcée, il faut parenthéser les expressions.

Exemple :

$$(2 + 3) * 4 = 20$$

Quand on mélange les opérateurs de calcul et les opérateurs de comparaison, il est vivement conseillé de parenthéser vos expressions.

Exemples :

$$2 * 3 <= 4 + 1 \quad \text{est une horreur}$$

$$(2 * 3) <= (4 + 1) \quad \text{est tellement plus lisible...}$$

### **2.2.2 Les caractères.**

Ils servent principalement à interfacer des périphériques. Ils seront représentés entre des apostrophes.

Exemple :

'a' 'B' sont deux caractères.

Tous les caractères imprimables de la table ASCII pourront être représentés. Il est à noter que les caractères 'a' et 'A' sont deux caractères différents.

Le caractère '2' qui est un chiffre, et le nombre 2 qui est un entier seront bien sûr différenciés. Quant à la variable A et au caractère 'A', ils n'ont rien à voir.

Les seules opérations permises sur les caractères sont les opérations de comparaison. Les opérateurs permis sont les suivants : < > = >= <= <>.

L'ensemble des caractères est un ensemble ordonné correspondant à celui de la table ASCII.

Exemple :

'A' < 'a'



### 2.2.3 Les booléens.

Les booléens servent à exprimer un état binaire : partout où une information n'a que deux états possibles un booléen est utilisé. Les opérateurs de comparaison délivrent des résultats booléens, car ils n'ont que deux états : soit la condition est remplie, soit la condition n'est pas remplie.

Les deux valeurs possibles pour les booléens sont vrai et faux.

Les opérations possibles sur les booléens sont les opérations de comparaison et les opérations logiques les plus élémentaires.

- Les opérations de comparaison :  $<$   $>$   $=$   $<>$   $>=$   $<=$  Il est à noter que vrai  $>$  faux.
- Les opérations logiques et ou non.

Nota : ici aussi il y a un ordre de priorité d'exécution des opérateurs.

- non (opérateur unaire) est le plus prioritaire.
- et
- ou
- $<$   $>$   $=$   $<>$   $<=$   $>=$  sont les moins prioritaires.

Bien sûr les expressions booléennes complexes seront parenthésées, pour fournir une bonne compréhension de notre expression.

**Note:** Les deux précisions qui suivent ne seront lues que quand les instructions auront été comprises.

1) Le résultat d'une comparaison quelconque est un booléen (c'est à dire que le résultat de la comparaison est soit vrai soit faux). Prenez l'exemple suivant:

...

i : entier // i est un entier quelconque

bool : booléen // bool est un booléen vrai quand i est plus grand que cinq

...

**Si** (i > 5) **Alors**

bool := vrai

**Sinon**

bool := **faux**

## **Finsi**

Ceci peut en conséquences s'écrire de manière plus concise:

bool := (i > 5)

Le seul critère de choix entre les deux formulations est celui de la lisibilité. Cette lisibilité est fortement liée à la pratique que l'on a de la programmation. La première formulation semble plus compréhensible pour un débutant, tandis que la deuxième est plus concise pour un programmeur chevronné.

2) Le **et** et le **ou** algorithmiques ne sont pas le **et** et le **ou** logique habituel. En effet ils ne sont pas commutatifs. Les opérateurs logiques algorithmiques doivent être compris comme **et-alors** et **ou-sinon**. Le fonctionnement de chaque opérateur sera détaillé, puis il sera mis en évidence l'intérêt d'opérateurs non commutatif sur un exemple.

a **et** b : si a est faux la condition est fausse, sans évaluer b (notez que b pourrait dans ce cas ne pas avoir de valeur, ne pas être calculable!).

si a est vrai, le résultat de l'expression est b (qui sera alors évalué, et qui doit donc être calculable).

a **ou** b: si a est vrai la condition est vraie, sans évaluer b (qui pourrait ici aussi ne pas être calculable).

Si a est faux, le résultat de l'expression est b (qui sera alors évalué, et qui doit donc être calculable).

Cette manière d'interpréter le **et** et le **ou** donne aux deux opérandes un rôle qui n'est pas commutatif. Prenons un exemple plus parlant:

**Si** (i <> 0) **et** ((3 div i) >= 1) **Alors** ...

Comme notre **et** n'est pas commutatif, on vérifiera d'abord que i est non nul, et seulement si il n'est pas nul, la division par i sera faite. Si l'opérateur **et** avait été commutatif il ne serait pas possible d'écrire ceci de cette façon.

Il est à noter que le jour où le programmeur transcrit un algorithme dans un langage de programmation, il est extrêmement important de vérifier si le langage supporte ou non un tel **et** et **ou** non commutatifs.

### 2.2.4 Les réels.

Les réels permettent de représenter des valeurs décimales, sans limitation d'écriture, mais sans la forme exponentielle.

Exemples :

238.45      425.0      32.47      sont des réels.

Mais 425 est un entier...

Il y a deux sortes d'opérateurs sur les réels, les opérateurs de comparaison et les opérateurs de calcul.

- Les opérateurs de comparaison sont = > < <> >= <= . Le résultat de la comparaison est un booléen.
- Les opérateurs de calcul sont + - \* /. La division est l'opérateur /. Les opérateurs de calcul ont la priorité habituelle des opérateurs.

On pourra faire un calcul entre un entier et un réel, avec les opérateurs des réels, mais le résultat sera uniquement un réel.

### 2.2.5 Les tableaux.

Voici l'exemple d'un tableau de caractères, communément appelé chaîne de caractères. Par la suite, l'exemple sera généralisé à tous les types de tableaux.

Les chaînes de caractères

Une chaîne de caractères est un tableau où chaque case contient un caractère. Chaque case est numérotée de 1 jusqu'à la taille du tableau.

'S'	'A'	'L'	'U'	'T'		
1	2	3	4	5	...	n

Déclaration du tableau :

```
constante n = 80      // taille du tableau  
type chaîne = tableau [ n ] caractères  
                                 // type des tableaux de caractères traités
```

**variable**

```
texte : chaîne            // chaîne contenant le texte  
cible : caractère        // caractère pour le travail demandé  
i : entier                // indice de parcours de texte
```

- Mettre le ième caractère du tableau dans la cible ?

cible := texte [ i ]

- Mettre la cible dans le ième et le i+1ème caractères de texte ?

texte [ i ] := cible

texte [ i + 1 ] := cible

ou

texte [ i ] := cible

i := i + 1

texte [ i ] := cible

- Regarder un élément du tableau texte ?

si texte [ i ] = cible alors .....

si texte [ i ] = texte [ i + 1 ] alors .....

Généralisation à tous les tableaux.

Tous ce qui a été vu précédemment est valable pour des tableaux d'entiers, des tableaux de booléen, des tableaux de réels.

Plus généralement, si t est un type défini dans un programme ou une procédure, le programmeur pourra définir un tableau de t.

Exemple :

types tabent = tableau [ 10 ] de entier

matrice = tableau [ 10 ] de tabent

Le type matrice est défini comme un tableau de tableau d'entier. Ceci permet de définir des tableaux à plusieurs dimensions.

Comment affecter un élément d'un tel tableau ?

...

variables i : entier

j : **entier**

matcarrée : matrice

...

matcarrée [ i ] [ j ] := 2      // matcarrée est de type matrice

   // matcarrée [ i ] est de type tabent

   // matcarrée [ i ] [ j ] est de type entier

On pourra aussi prendre la notation matcarrée [ i, j ] qui a exactement la même signification.

### **2.2.6 Les types énumérés.**

Le formalisme doit permettre de garder des informations qui caractérisent des objets, sans les dimensionner. Par exemple une couleur, un état (arrêt, marche, panne). Il serait souhaitable de garder ces informations sous la forme la plus proche de celle employée couramment, afin que le programme soit le plus compréhensible possible à la relecture.

Pour représenter cette énumération de caractéristiques possibles, voici la définition d'un type énuméré:

...

**types** état\_moteur = (arrêt, marche, panne)

...

**variables** état\_alternateur : état\_moteur

...

état\_alternateur := marche

...

**si** état\_alternateur = panne **alors**

...

Ce type énuméré possède une autre caractéristique, c'est qu'il est ordonné. Le premier élément cité dans la liste de l'énumération est le plus petit, le dernier est le plus grand. En fait vous avez déjà traité un type énuméré ordonné, sans le savoir : c'est le type booléen (**faux**, **vrai**). Voici un autre exemple :

...

**types** gabarit = (petit, moyen, grand)

```

...

variables taille : gabarit

...

taille := moyen

...

si  taille > petit alors

...

```

Il est conseillé d'utiliser des types énumérés chaque fois que c'est possible, cela aide énormément à la relecture des programmes, et cela évite des erreurs de codage d'informations.

Les opérateurs sur les types énumérés sont les opérateurs de comparaison > < = >= <= <>

### **2.2.7 Les enregistrements.**

On désire conserver des informations sur une personne, concernant son nom et son âge, ces données représentant la personne.

Il est possible de gérer une chaîne de caractères pour le nom et un entier pour l'âge. Or il serait préférable de pouvoir mettre en évidence la liaison qui existe entre ces deux données. L'idéal étant de pouvoir les regrouper en une seule information.

```

type  info = enregistrement  // information sur une personne

        nom : chaîne  // nom de la personne

        âge : entier    // âge de la personne

finenregistrement

```

```

variables  personne : info  // une personne que l'on va traiter

...

début

...

```

```
personne.nom := 'toto'
```

```
personne.âge := 27
```

```
...
```

```
i := personne.âge
```

```
...
```

**fin**

L'information est composée de deux champs, un champ nom et un champ âge.

Une agence matrimoniale désire conserver certaines informations pour se constituer un fichier. Pour les hommes elle ne s'intéresse bien sûr qu'à leur salaire, et pour les dames qu'à leurs mensurations.

Comment construire une information de ce type?

**types** info = **enregistrement** // client ou cliente de l'agence

nom : chaîne // nom de la personne

âge : **entier** // âge de la personne

**choix** sexe : **booléen sur**

**vrai** : salaire : **entier**

// salaire si c'est un homme

**faux** : mensurations : **enregistrement**

// mensurations si c'est une femme

poitrine : **entier**

// tour de poitrine

taille : **entier**

// tour de taille

hanches : **entier**

// tour de hanches

**finenregistrement**

**finchoix**

## finenregistrement

...

### variables

personne : info // une personne

...

### début

...

personne.âge := 28

...

lire(personne.sexe) // on lit le sexe de la personne

...

si personne.sexe alors

personne.salaire := 5000 // personne du sexe masculin

sinon // personne du sexe féminin

personne.mensuration.poitrine := 90

personne.mensuration.taille := 60

personne.mensuration.hanches := 90

finsi

...

fin

Il est à noter que le choix a été fait sur un booléen. Il aurait pu être fait sur un champ de type quelconque.

La caractéristique homme ou femme, dans l'exemple précédent, est représentée par un booléen. Il serait préférable de trouver une représentation au niveau de l'algorithme plus facile à lire.

Exemple :

types ange = (masculin, féminin)

info = enregistrement



...

**choix** sexe : ange **sur**

masculin : ...

féminin : ...

**finchoix**

**finenregistrement**

...

**si** personne.sexe = masculin **alors** ...

...

### **2.2.8 Les pointeurs.**

Jusqu'à maintenant les objets déclarés étaient :

**Statiques** dans le cas d'un programme : Les objets sont définis une fois pour toute à l'appel du programme et leur durée de vie est celle du programme.

**Automatiques** dans le cas d'une procédure : Les objets sont définis à chaque appel de la procédure et leur durée de vie est celle de la procédure. La procédure ne peut donc pas garder une valeur qu'elle se transmet appel après appel sans passer l'information par le programme qui l'utilise.

Dans ces deux cas tous les objets utilisés sont définis avant la partie instruction en nombre et en taille. Cela ne correspond pas toujours aux besoins rencontrés dans le monde informatique. Quand une application doit gérer un nombre d'informations qui est très variable dans le temps, il peut être dommage de réserver une place de stockage d'information pouvant contenir le nombre maximum d'informations que l'on aura à gérer. D'une part, il est souvent difficile d'estimer ce nombre maximum, d'autre part cette place ne sert pas à grand chose sauf dans des cas extrêmes.

D'où l'idée de pouvoir manipuler des variables dynamiques, que l'on crée quand on en a le besoin, et que l'on libère quand elles ne servent plus. Ainsi le programmeur s'affranchit des problèmes de nombres maximum de données à traiter, et ne sera, sur la machine, plus limité que par la capacité mémoire virtuelle.

Jusqu'à maintenant les données sont référencées par un identificateur qui est défini avant les instructions qui les utilisent. Il n'est pas possible de définir un identificateur de manière dynamique. Si bien que les données dynamiques ne seront pas repérées par un identificateur mais par un pointeur.

Un **pointeur** est un objet qui permet de référencer un objet créé de manière dynamique. Il est défini pour référencer des objets d'un type bien précis. Soit un type élément défini dans l'algorithme. Soit un pointeur d'élément, ce pointeur ne pourra référencer des objets dynamiques que s'ils sont du type élément.

Un pointeur est une variable qui a le type pointeur du type de l'information qu'il représente. Cette variable va pouvoir être initialisée de plusieurs manières:

Par affectation, il lui sera donné la valeur d'un autre pointeur du même type.

Par affectation, il lui sera donné la valeur **null** qui indique que le pointeur ne référence pas d'information. Attention un pointeur non initialisé n'a pas la valeur **null**.

Par création, il lui sera donné comme valeur la référence de l'objet qui vient d'être créé.

Les autres opérations permises sur les pointeurs sont les suivantes:

- Comparer l'égalité de deux pointeurs de même type.
- Regarder si un pointeur est égal à la valeur **null**.
- Détruire l'objet dynamique référencé par un pointeur. Attention cette opération n'est permise que si le pointeur référence effectivement un objet. A l'issue de la destruction de l'objet le pointeur n'a pas de valeur définie : il ne vaut pas **null**.

De ce qui vient d'être énoncé, il est bon de tirer quelques leçons :

- Il ne faut pas essayer d'accéder à un objet dynamique quand la valeur du pointeur est nulle ou non définie.
- La seule manière de référencer un objet dynamique est son pointeur. Si la référence à un objet dynamique est perdue (en changeant la valeur du pointeur par exemple) l'algorithme ne pourra plus accéder à l'objet, ni le détruire.

En conclusion les pointeurs vont faire gagner en souplesse d'utilisation, ôtent le souci des limites, mais par contre ils sont d'un emploi délicat quant à l'écriture des algorithmes.

Regardez maintenant comment utiliser les pointeurs.

### Comment déclarer une variable de type pointeur?

Soit un type élément quelconque :

ref : **pointeur de** élément    // pointeur sur un objet dynamique

### Comment créer un objet dynamique?

**créer** (ref)    // ref référence un objet de type élément

## Comment utiliser cet objet dynamique?

ref-> est l'objet élément (quelque soit le type élément).

Si élément est un type entier on peut écrire :

ref-> := 4

**Si** ref->=5 **alors** ....

Si élément est un type tableau d'entiers on peut écrire:

ref->[4] := 56

**tantque** ref->[i] = ref->[i+1] **faire** ...

Si élément est un type enregistrement avec deux champs entiers long et large, on peut écrire:

ref->.long := 4

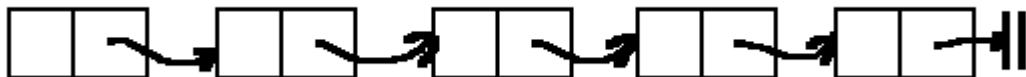
**si** ref->.long = ref->.large **alors** ...

On s'aperçoit vite que l'intérêt de manipuler une donnée dynamique simple est limité car il faut un pointeur par donnée simple ce qui n'apporte rien.

Dans le cas d'un tableau, il peut être intéressant de le créer à un moment particulier mais sa taille sera fixe car se référant à un type déclaré.

L'utilisation optimale des pointeurs se trouve quand des enregistrements doivent être créés. Mais pas n'importe quel enregistrement. Le besoin est de créer des ensembles dynamiques d'éléments. Par exemple une liste dynamique d'éléments.

Exemple :



Voici comment déclarer un enregistrement permettant cela:

**types**

référence = **pointeur de** élément // pointeur sur un élément

info = ... // information que l'on désire stocker

élément = **enregistrement**

valeur : info

suivant: référence

### **finenregistrement**

Notez que l'élément contient la référence à un autre élément (créé ou à créer). Ainsi chaque fois que l'on crée un élément pour la liste on possède une référence supplémentaire. Si on connaît le premier élément de la liste on a la possibilité de parcourir toute la liste de proche en proche.

Notez également que le type référence est défini par rapport à un type non encore déclaré. C'est le seul cas où on s'autorise à utiliser un objet non défini.

### **Comment comparer deux pointeurs?**

Soit p, q deux pointeurs de type référence.

si p = q alors ...

La condition ne sera vraie que si p et q pointent le même élément ou si p et q valent **null**. Si au moins un des deux pointeurs est non défini la condition n'a pas de sens.

si p = **null** alors ...

Si p a explicitement été initialisé à la valeur **null** alors la condition est vraie. Si p référence un objet dynamique alors la condition sera fausse. Si p est non défini la condition n'a pas de sens.

### **Comment détruire un objet dynamique?**

Soit ref un pointeur de type référence.

**détruire**(ref) // destruction de l'élément pointé par ref

La destruction d'un élément n'a de sens que si l'élément est défini.

Soit la suite d'instructions suivante:

ref, pointe : référence

...

**créer**(ref) // on crée un élément pointé par ref

pointe := ref // ref et pointe référencent tous les deux le même élément

**détruire**(ref) // ref et pointe ne référencent plus rien

// il ne faut surtout pas essayer d'utiliser ref ou pointe pour

// accéder à un élément

// notez que détruire(pointe) aurait eu dans ces conditions

// exactement le même résultat

### **2.2.9 Types non nommés.**

Un type nommé est un type défini dans la rubrique **types**, ou un type prédéfini (entier, booléen, caractère, réel). Un type non nommé est toute référence à une construction de type (par exemple tableau [ 10 ] de caractères), sans utiliser un type défini dans une rubrique **types**.

Voici un exemple :

...

**types** tabent = **tableau** [ 10 ] **de entier**

...

**variables** tab1 : tabent // voici une référence à un type nommé

tab2 : **tableau** [ 10 ] **de entier** // voici une référence à un type non nommé

tab1 et tab2 sont deux tableaux qui ont même structure, mais qui sont de types différents.

Partout où le programme demande une variable de type tabent, le programmeur ne pourra pas lui donner tab2. Pour ce faire, il aurait fallu déclarer tab2 de type tabent.

Un type de donnée construit une famille de variables interchangeable entre elles, et garantit ainsi le programmeur de toute confusion entre variables qui, bien qu'ayant la même structure, ont des rôles différents.

## 2.3 INSTRUCTIONS.

Les instructions sont les opérations qui vont permettre de faire interagir les différents objets entre eux. Nous disposerons de trois types d'instructions nous permettant d'accéder à cinq instructions. Cet ensemble réduit d'instructions est suffisant pour traiter l'ensemble des problèmes rencontrés.

### 2.3.1 Instruction d'affectation.

L'instruction d'affectation permet de donner une valeur à une variable, dans la mesure où cette variable est d'un type simple. Les variables de type complexe ne peuvent être affectées directement mais leurs champs ou éléments peuvent être affectés s'ils sont de type simple (entier, booléen, caractère, réel, pointeur).

Exemple :

**variables** **i** : **entier**

...

**i** := 3            // **i** prend pour valeur 3

**i** := **i** + 1        // **i** est incrémenté de 1

**i** := 4 \* **i**        // **i** est multiplié par 4

### 2.3.2 Instructions alternatives.

il y a deux sortes d'instruction alternatives :

#### 2.3.2.1 instruction **si**.

L'instruction **si** permet d'effectuer une ou plusieurs instructions si une certaine condition est remplie (que l'on retrouve dans une proposition du type : si la neige est bonne je vais skier).

Elle permet également d'effectuer un traitement si une condition est remplie, et un autre traitement dans le cas contraire (que l'on retrouve dans une proposition du type : si la neige est bonne je vais skier, sinon je vais au cinéma).

Voici le format type, sous ses deux formes, de l'instruction **si**:

**Si**   < condition >   **Alors**

    < instruction 1 >

    ...

    < instruction n >

**Finsi**

**Si** < condition > **Alors**

< instruction 1 >

...

< instruction n >

**Sinon**

< instruction 1' >

...

< instruction n' >

**Finsi**

Exemple :

**variables** i : **entier**

...

**Si** i > 4 **Alors**

i := i - 1

**Sinon**

i := i + 1

**Finsi**

### ***2.3.2.instruction choix.***

L'instruction choix permet d'associer à différentes valeurs discrètes (il s'agit ici de valeurs particulières, et non de plages de valeurs) des instructions à exécuter. Cette instruction peut être réalisée par une cascade de **si sinon**, mais elle offre une présentation plus agréable à la lecture et à la compréhension.

Voici le format type de l'instruction **choix**:

**Choix** **sur** < variable > **faire**

< valeur 1 > : < instruction 1 1 >

< instruction 1 2 >

...

< instruction 1 n(1) >

...

< valeur m > :      < instruction m 1 >

< instruction m 2 >

...

< instruction m n(m) >

**autrecas**            :    < instruction (m+1) 1 >

...

< instruction (m+1) n(m+1) >

### **Finchoix**

Il est obligatoire de mettre **autrecas**, même dans les cas où il n'y en a pas, car cela donne l'occasion au concepteur de l'algorithme de faire le point sur tous les cas énumérés.

Exemple :

**variables** i : **entier**

...

**Choix**    **sur** i    **faire**

1 :      **écrire** ('1 c'est bien!!')

2 :      **écrire** ('2 c'est mieux!!')

3 :      **écrire** ('3 bonjour les dégâts!!')

**autrecas**            :    **écrire** ('choix impossible!!')

### **Finchoix**

#### **2.3.3 Instructions répétitives.**

Il existe deux formes d'instructions itératives. Ces deux formes sont légèrement différentes, et correspondent à deux types de problèmes différents. Il est important de bien voir la différence entre ces deux formes.

Les instructions itératives se caractérisent par trois choses:



- une éventuelle répétition des actions
- une évolution de notre univers par ces actions
- une condition de terminaison testée sur notre univers.

### 2.3.3. *Instruction Tantque.*

L'instruction **tantque** permet d'exécuter une instruction tant qu'une condition est remplie. Il est préférable que la condition puisse être changée par les instructions, sinon notre algorithme ne terminerait jamais (ceci correspond à une formulation du type : tant que le soleil n'est pas couché, je reste à la plage).

Voici le format type de l'instruction **tantque**:

```
tantque  < condition >  faire      // c'est une condition de continuation
      < instruction1 >
      ...
      < instruction >
```

**fintantque**

Voici un exemple pour voir comment fonctionne cette instruction :

**variables** i : **entier**

...

```
tantque  (i < 3)  faire      // on arrête quand i >= 3
                                     // on met ici le commentaire sur la terminaison
                                     // de la boucle. C'est à dire la condition
contraire
                                     // à celle exprimée dans le tantque.
```

i := i + 1

**écrire** ('coucou')

**fintantque**

**écrire** (' i = ',i)

Prenez le cas où i = 5 avant l'exécution de notre tantque.

La condition n'étant pas remplie on n'effectue pas le tantque, on va donc directement effectuer l'instruction écrire, le résultat est alors:

**i = 5**

Prenez le cas où  $i = 1$  avant l'exécution de notre tantque.

La condition est remplie, on incrémente  $i$ , on écrit coucou, et on vient à nouveau tester la condition. Cette condition est remplie, on incrémente  $i$  et on écrit coucou.  $i$  maintenant vaut 3, et on revient tester notre condition qui n'est plus remplie. on va donc écrire le résultat. L'algorithme aura donc écrit:

**coucou**

**coucou**

**$i = 3$**

### 2.3.3.2 instruction répéter.

L'instruction répéter permet d'exécuter une instruction jusqu'à ce qu'une condition soit remplie. Ceci correspond à une formulation du type : je mange jusqu'à ne plus avoir faim (ce qui implique que l'on mange avant de se poser la question si l'on a faim).

Voici le format type de l'instruction répéter :

**répéter**

< instruction1 >

...

< instruction >

**jusqu'à** < condition >      // c'est une condition de terminaison

Voici un exemple pour voir comment fonctionne cette instruction:

**variables**  $i$  : **entier**

...

**répéter**

$i := i + 1$

**écrire** ('coucou')

**jusqu'à**  $i \geq 3$

// la boucle se termine quand  $i \geq 3$ . ici, c'est la

// condition de terminaison qui est donnée, c'est

// à dire la condition du jusqu'à.

**écrire** ('  $i =$  ',  $i$ )

Prenez le cas où  $i = 5$  avant l'exécution du répéter.

On commence par incrémenter  $i$ , puis on écrit coucou. La condition de terminaison est vérifiée, on écrit donc le résultat.

Les écritures correspondant à cette exécution sont:

**coucou**

**$i = 6$**

Prenez le cas où  $i = 1$  avant l'exécution du répéter.

On commence par incrémenter  $i$ , puis par écrire coucou. On teste la condition de terminaison qui n'est pas vérifiée. On incrémente à nouveau  $i$ , on écrit coucou, on teste la condition de sortie qui est vérifiée. Les écritures correspondant à cette exécution sont :

**coucou**

**coucou**

**$i = 3$**

## **2.4 Outils d'entrée/sortie.**

### **2.4.1 Les entrées/sorties à la console.**

Pour tous les exercices proposés, il existe des actions complexes (qui sont en fait des outils disponibles sur l'ordinateur) qui permettent de lire des informations au clavier et d'écrire des informations à l'écran.

**Lire** (sortie boîte : **quelconque**)

Lire met dans la variable qui sera donnée à l'utilisation de la procédure les informations entrées au clavier.

Boîte représente la variable qui doit être donnée pour récupérer l'information. C'est une variable de type quelconque simple (entier, réel, booléen, caractère). Notez que la définition de paramètres de type quelconque n'interviendra que pour les deux procédures faisant appel au système de l'ordinateur **lire** et **écrire**.

Exemple :

**variable** nombre : **entier** // nombre que l'on va lire

.....

**lire**(nombre) // on lit au clavier la valeur du nombre

**si** nombre = 25 **alors** .....

.....

### **Ecrire (entrée résultat : quelconque)**

Ecrire permet d'afficher à l'écran la valeur qui lui est donnée.

Résultat représente la valeur (variable ou expression) que l'on veut afficher. C'est une expression de type quelconque simple (entier, réel, booléen, caractère).

Exemple :

```
variable calcul : entier // nombre que l'on va afficher
```

.....

```
écrire ('le nombre calculé est : ') // écriture d'un texte
```

```
écrire (calcul) // écriture du résultat
```

```
écrire (2*calcul+3) // écriture d'un autre résultat
```

Cet outil de sortie permet d'écrire plusieurs choses en une seule ligne d'écriture

Exemple :

```
écrire ('le nombre calculé est : ', calcul) // écriture d'un texte
```

### **2.4.2 Les entrées/sorties fichier.**

Les données traitées par les programmes informatiques ont une durée de vie inférieure ou égale à la durée de vie du programme lui-même. C'est à dire que les données traitées par un programme sont perdues quand on arrête le programme.

Quand on désire conserver les informations d'une exécution du programme à l'autre, il faut stocker ces données dans un fichier, ces données auront alors une existence propre.

La gestion des fichiers peut se faire suivant plusieurs logiques, nous en retiendrons deux : la gestion séquentielle des données et la gestion par index.

#### **2.4.2.1 Fichiers séquentiels.**

Les données seront traitées depuis la première donnée entrée jusqu'à la dernière, dans cet ordre immuable.

Nous pouvons ouvrir un fichier, lire un élément du fichier, écrire un élément du fichier, fermer le fichier, et tester la fin du fichier.

- La déclaration d'un fichier à accès séquentiel se fait de la manière suivante :

nomfic : **fichier de** type\_élément

où nomfic est la variable représentant le fichier, type\_élément est le type des éléments du fichier.

- L'ouverture d'un fichier se fait de la manière suivante :

**ouvrir** ( 'fichier.dat' , nomfic )

où fichier.dat représente un nom de fichier sur la mémoire de masse, et nomfic représente le fichier. La procédure **ouvrir** fait le lien entre le fichier et sa représentation interne à l'algorithme.

- La lecture d'un élément du fichier se fait de la manière suivante :

**lire** ( nomfic , varlec )

où varlec représente une variable de type type\_élément. La procédure **lire** affecte la valeur lue dans le fichier à la variable varlec. Il faut avoir testé au préalable que le fichier n'est pas à sa fin.

- L'écriture d'un élément du fichier se fait de la manière suivante :

**écrire** ( nomfic , valecr )

où valecr représente une valeur de type type\_élément. La procédure **écrire** sauvegarde dans le fichier la valeur représentée par valecr.

- La fermeture d'un fichier se fait de la manière suivante :

**fermer** ( nomfic )

La procédure ferme le fichier, c'est à dire que nomfic n'est plus associé à un fichier. Il ne faut plus utiliser nomfic, sauf à ouvrir un fichier, c'est à dire à associer nomfic à un autre fichier ( ou le même ).

- Le test de fin de fichier se fait de la manière suivante :

**Si finfichier** ( nomfic ) **Alors** ...

La fonction teste si le dernier élément du fichier a été lu. Elle rend donc vrai quand il n'y a plus rien à lire dans le fichier.

Précisons le fonctionnement de **lire** et **écrire**. Ces deux fonctions sont exclusives sur une utilisation d'un fichier. Quand un fichier est ouvert, nous pouvons écrire des valeurs dedans, mais si nous voulons les relire, il faut fermer le fichier, puis le rouvrir pour lire les valeurs qui s'y trouvent.

Quand un fichier est lu, la première valeur lue sera la première valeur mise dans le fichier, puis les suivantes dans l'ordre d'insertion.

Quand une valeur est écrite dans un fichier, à la première écriture, le fichier est vidé de son contenu, et la nouvelle valeur est insérée. Les écritures suivantes, dans la même utilisation de ce fichier, s'ajoutent aux valeurs contenues dans le fichier. Il n'est donc pas possible, directement, de rajouter des valeurs à un fichier fermé.

Regardons un exemple simple d'algorithme utilisant un fichier séquentiel :

Nous voulons enregistrer, dans un fichier, des caractères donnés par l'opérateur, jusqu'à ce que celui-ci entre un 'z' ( qui ne sera pas enregistré ). Nous afficherons ensuite le contenu du fichier.

**Programme** essai-fichier // programme d'essai de fichier séquentiel

**Variables** fichecar : **fichier de caractères** // variable représentant le fichier

car : **caractère** // caractère lu au clavier

**Début**

// remplir le fichier avec les caractères de l'opérateur

**ouvrir** ( "toto.dat", fichecar ) // ouverture du fichier toto.dat

**écrire** ( ' donnez des caractères en terminant par un "z" ' )

**lire** ( car )

**Tantque** ( car <> 'z' ) **Faire** // arrêt quand car = 'z'

**écrire** ( fichecar, car ) // ranger le caractère dans le fichier

**lire** ( car )

**Fintantque**

**fermer** ( fichecar ) // fermeture du fichier

// relecture du fichier pour voir son contenu

**ouvrir** ( fichecar, "toto.dat" ) // ouverture du fichier toto.dat

**Tantque non finfichier** ( fichecar ) **Faire** // arrêt en fin de fichier

**lire** ( fichecar, car )

**écrire** ( car )

**Fintantque**

**fermer** ( fichecar ) // fermeture du fichier

**Fin** // les données sont toujours accessibles par un autre programme

// pour la lecture seulement

#### 2.4.2.2 Fichiers indexés.

Nous désirons conserver des informations. Ces informations constituent des enregistrements dont un champ est unique ( numéro de sécurité sociale, nom unique, ... ). Ce champ unique est une clef d'accès à l'enregistrement, c'est à dire que l'on peut retrouver l'enregistrement uniquement en connaissant cette clef.

Nous définirons, ici, un fichier indexé comme un fichier qui gère automatiquement les accès à des enregistrements à clef unique.

Nous pourrons ouvrir un fichier indexé, le fermer, se positionner sur un élément, puis le lire, l'écrire ou le détruire.

- La déclaration d'un fichier indexé se fait de la manière suivante :

**type** type\_élément = **enregistrement**

clef : type\_quelconque

// le premier champ est la clef unique d'accès à la donnée

.....

**finenregistrement**

nomfic : **fichier de** type\_élément

où nomfic est la variable représentant le fichier, type\_élément est le type des éléments du fichier. Ce fichier n'est à priori pas indexé. C'est l'ouverture du fichier, à sa création qui en fera ou non un fichier indexé.

- L'ouverture d'un fichier indexé se fait de la manière suivante :

**ouvrir** ( 'fichier.dat' , nomfic, **indexé** )

où fichier.dat représente un nom de fichier sur la mémoire de masse, et nomfic représente le fichier. La procédure **ouvrir** fait le lien entre le fichier et sa représentation interne à l'algorithme. Ce fichier doit avoir été ouvert à sa création en mode indexé. Un fichier indexé peut être parcouru en mode séquentiel. Pour cela il suffit de l'ouvrir comme un fichier séquentiel, puis de lire séquentiellement ses éléments. Par contre l'écriture ne se fera qu'en mode indexé.

Un fichier créé en mode séquentiel ne pourra jamais être exploité par les outils des fichiers indexés.

- La lecture d'un élément du fichier indexé se fait de la manière suivante :

**variables** index : type\_quelconque

...

**Si** positionner ( nomfic, index ) **Alors** // l'élément existe nous pouvons le lire

lire ( nomfic , varlec )

...

où index représente la clef d'accès à l'élément que l'on désire lire, varlec représente une variable de type type\_élément. La procédure lire affecte la valeur lue dans le fichier à la variable varlec. Il faut avoir testé au préalable que l'élément est présent dans le fichier, par la fonction positionner.

- L'écriture d'un élément du fichier indexé se fait de la manière suivante :

variables index : type\_quelconque

...

**Si** non positionner ( nomfic, index ) **Alors**

// l'élément n'existe pas nous pouvons l'écrire

écrire ( nomfic , valecr )

...

où index représente la clef d'accès à l'élément que l'on désire écrire, valecr représente une valeur de type type\_élément. La procédure écrire sauvegarde dans le fichier la valeur représentée par valecr. Il faut au préalable vérifier que l'élément n'existe pas déjà dans notre fichier.

- La fermeture d'un fichier se fait de la manière suivante :

fermer ( nomfic )

La procédure ferme le fichier, c'est à dire que nomfic n'est plus associé à un fichier. Il ne faut plus utiliser nomfic, sauf à ouvrir un fichier, c'est à dire à associer nomfic à un autre fichier ( ou le même ).

- La destruction d'un élément du fichier indexé se fait de la manière suivante :

variables index : type\_quelconque

...

**Si** positionner ( nomfic, index ) **Alors**

// l'élément existe nous pouvons le détruire

détruire ( nomfic )

...



objet : bien // bien à entrer au fichier du patrimoine

## **Début**

// saisie des nouveaux biens à ajouter au patrimoine

**ouvrir** ( nomfic, fic, **indexé**) // ouverture du fichier de bien en indexé

**écrire** ( 'donnez un identificateur : ')

**lire** ( idf )

**Tantque** idf <> finsaisie **Faire** // arrêt quand idf = 'quit'

**Si** **positionner** ( fic, idf ) **Alors**

**écrire** ( 'le bien est déjà répertorié au patrimoine')

**Sinon**

objet.identif := idf

... // saisie des autres informations du bien

**écrire** ( fic, objet )

**Finsi**

**écrire** ( 'donnez un identificateur : ')

**lire** ( idf )

**Fintantque**

**fermer** ( fic ) // fermeture du fichier patrimoine

// relecture des biens par ordre alphabétique des identificateurs

**ouvrir** ( nomfic, fic) // ouverture du fichier patrimoine en séquentiel

**Tantque** **non finfichier** ( fic) **Faire**

**lire** ( fic, objet ) // lecture d'un bien

... // affichage du bien

**Fintantque**

**fermer** ( fic ) // fermeture du fichier du patrimoine

**Fin**

## 2.5 PROCÉDURES ET FONCTIONS.

### 2.5.1 Les procédures.

Les seules différences avec un programme sont pour l'entête.

**procédure** calcul (**entrée** x : **entier** , **sortie** y : **entier**)

// cette procédure calcule la puissance trois d'un nombre

// x est le nombre que l'on va élever à la puissance

// y est le résultat du calcul

**constante** puissance = 3 // puissance à calculer

**variable** i : **entier** // comptage des puissances

**début**

i := 1 // initialisation des puissances et du résultat

y := 1

**tantque** i <= puissance **faire**

// on arrête quand i > puissance c'est à dire quand toutes les

// puissances ont été traitées

y := y \* x

i := i + 1

**fintantque**

**fin** // calcul

Programme utilisant la procédure :

**programme** cube

// ce programme calcule le cube du nombre que l'opérateur donne au clavier

**variable**

nombre : **entier** // nombre donné par l'opérateur

résultat : **entier** // cube du nombre donné

**procédure** calcul (**entrée** x : **entier** , **sortie** y : **entier**)

// cette procédure calcule le cube du nombre x et met le résultat dans y

**début**

**lire**(nombre) // acquisition du nombre

calcul (nombre , résultat) // calcul du cube

**écrire** ('le résultat est : ', résultat)

// affichage du résultat. Avec l'utilisation

// d'une seule instruction écrire

**fin**

### **2.5.2 Les fonctions.**

Voici l'exemple précédent appliqué à une fonction.

**fonction** calcul (**entrée** x : **entier**) : **entier**

// cette fonction calcule la puissance trois d'un nombre

// x est le nombre que l'on va élever à la puissance

// la fonction retourne le cube du nombre

**constante** puissance = 3 // puissance à calculer

**variable** i : **entier** // comptage des puissances

y : **entier** // calcul intermédiaire

**début**

i := 1 // initialisation des puissances et du résultat

y := 1

**tantque** i <= puissance **faire**

// on arrête quand i > puissance c'est à dire quand toutes les

// puissances ont été traitées

y := y \* x

i := i + 1

**fintantque**

**retourner** (y)

**fin** // calcul

Programme utilisant la fonction :

**programme** cube

// ce programme calcule le cube du nombre que l'opérateur donne au clavier

**variable**

nombre : **entier** // nombre donné par l'opérateur

**fonction** calcul (**entrée** x : **entier**) : **entier**

// cette procédure calcule le cube du nombre x et retourne le résultat

**début**

**lire**(nombre) // acquisition du nombre

**écrire** ('le résultat est : ', calcul (nombre))

// affichage du résultat. avec l'utilisation

// d'une seule instruction écrire

**fin**

## **2.6 STRUCTURE DE PROGRAMME.**

### **2.6.1 Structure Générale d'un programme.**

L'exemple de procédure ou de fonction montre une présentation qui n'est pas compatible avec la structure générale d'un programme vu au chapitre 1.3).

L'idée, ici, est de ne pas imbriquer programme et procédure, afin de garder une certaine lisibilité à nos algorithmes. Cela nécessite pour le programme ou la procédure qui utilise une procédure ou une fonction de posséder le mode d'emploi (ou description, ou prototype) de l'outil qu'il va utiliser. Cela nécessite également

que la procédure ait un nom assez évocateur pour qu'à la première lecture le lecteur puisse comprendre le rôle de la procédure ou de la fonction; pour plus de détails il faut se reporter aux spécifications de la procédure ou de la fonction (c'est à dire à son interface complète ou mode d'emploi).

Ce mode d'emploi comporte le nom de la procédure ou de la fonction, avec tous ses paramètres, et les commentaires qui expliquent comment utiliser la fonction, et son rôle.

Reprenez la présentation de la structure d'un programme au chapitre 1.3) et comparez avec les exemples des chapitres 2.5.1) et 2.5.2).

Notez que quand le mode d'emploi d'un outil est défini, n'importe quel programmeur peut utiliser l'outil, sans savoir comment il est réalisé. Ceci est le fondement de la conception d'applications de manière descendante.

#### 2.6.2) Visibilité des objets.

Le principe est simple : un objet est visible (utilisable) dans l'unité algorithmique qui l'a défini, et dans toutes les unités algorithmiques définies dans celle-ci.

Ceci sera appliqué pour les constantes, les types, les procédures et les fonctions.

Mais il est **interdit** d'utiliser des variables dans une procédure ou fonction, qui ne seraient pas définies dans la procédure ou fonction ou en paramètre de celle-ci. Cette règle est universellement appliquée suite aux méfaits de l'utilisation de variables dites **globales** (c'est à dire ici non définies dans notre environnement). L'aide, pour la mise au point de programme, demandée à toute personne étrangère au développement est conditionnée par le respect de cette règle.

Pour finir, voici la définition des notions de variables globales et locales. Ces deux notions sont relatives. Une variable est dite locale si elle est définie dans l'environnement d'où on la regarde. Elle est dite globale si elle est définie dans un environnement qui est le père, ou un aïeul, de cet environnement. Donc une même variable peut être considérée globale ou locale relativement à l'endroit d'où elle est vue.