ANNEE UNIVERSITAIRE 2002 – 2003
COURS : ALGORITHMIE ET PROGRAMMATION
AUTEUR : FABRICE GONTARD

OUTILS DE BASE

Dans le cours précédent, nous avons donné la définition de l'algorithmie. Nous avons précisé que pour formaliser une solution à un sous problème, nous utiliserions une notation spécifique : le pseudo code issu d'un pseudo langage. Dans ce deuxième cours, nous allons utiliser cette syntaxe pour expliquer le principe des différents outils de base de l'algorithmie.

1. PREMIER PROGRAMME

Avant d'entrer dans le vif du sujet, commençons par mettre en place un premier algorithme. Celui ci va nous permettre de voir globalement les différentes parties d'un programme. Dans un deuxième temps, nous détaillerons plus précisément ces différentes parties afin d'expliquer les outils de l'algorithmie utilisés.

Enoncé : Ecrire un programme qui permet de calculer la somme de deux valeurs saisies au clavier.

```
Programme sommeDeuxValeurs
Var
        Valeur1, Valeur2 : NOMBRE /* Valeurs saisies au clavier */
        Somme : NOMBRE /* Contient le résultat de la somme */
Début
        /* Saisies des deux valeurs */
        Afficher « Entrez 2 valeurs »
        Saisir Valeur1
        Saisir Valeur2

        /* Effectuer la somme des deux valeurs */
        Somme ← Valeur1 + Valeur2

        /* Afficher le résultat */
        Afficher «Le résultat est », Somme
Fin
```

D'une manière générale, un algorithme :

- Commence toujours pour le mot **Programme** suivi du nom du programme. Le fait de nommer les algorithmes permet de les différencier ;
- Possède un bloc de déclaration de variables, repéré par le mot **Var** et qui contient la liste des variables utilisées dans l'algorithme (cf. le paragraphe 4 de ce cours) ;
- Possède un bloc d'instruction encadré par les mots **Début** et **Fin** contenant la liste des instructions que l'algorithme déroulera

séquentiellement. En fait, le programme commence son exécution sur le mot « Début » et se termine sur le mot « Fin ».

Ainsi, nous obtenons :

Programme NomDuProgramme
Var
    Déclaration des variables
Début
    Instructions
Fin

2. COMMENTAIRES

Les commentaires améliorent la lisibilité générale des programmes. Ils ne donnent lieu à aucun traitement, mais sont essentiels pour clarifier les différentes actions auprès de ceux qui liront votre algorithme. Il faut prendre l'habitude de les mettre en place au fur et à mesure de l'avancement du travail car bien souvent, à la fin, soit on ne rappelle plus de toutes les particularités de l'algorithme soit on n'a pas le courage de les ajouter.
En pseudo code, les commentaires commencent par « /* » et se terminent par « */ ».

Exemple : / Voici des commentaires*/*

Dans le programme précédent, nous avons mis en place cinq commentaires.

```
Programme sommeDeuxValeurs
Var
    Valeur1, Valeur2 : NOMBRE /* Valeurs saisies au clavier */
    Somme : NOMBRE /* Contient le résultat de la somme */
Début
    /* Saisies des deux valeurs */
    Afficher « Entrez 2 valeurs »
    Saisir Valeur1
    Saisir Valeur2

    /* Effectuer la somme des deux valeurs */
    Somme ← Valeur1 + Valeur2

    /* Afficher le résultat */
    Afficher «Le résultat est », Somme
Fin
```

3. NOTION DE TYPE

Les données manipulées par un programme peuvent être de natures différentes (entiers, réels, caractères, chaînes de caractères, booléen…). En fait en programmation, on parle plutôt de *type* de la donnée que de la nature de la donnée.

Le type d'une donnée manipulée en algorithmie ou dans les langages de programmation, spécifie la taille occupée par la donnée en mémoire, les opérations qui lui sont applicables ainsi que l'intervalle de valeurs autorisées.

Voici une liste non exhaustive des types utilisés en algorithmie :

- **NOMBRE** : représente le type ENTIER, la taille mémoire utilisée est de 2 octets et l'intervalle des valeurs autorisées est [**-32 768 , +32 767**] ;
- **LONG :** représente le type ENTIER LONG, la taille mémoire utilisée est de 4 octets et l'intervalle des valeurs autorisées est [**-2 147 483 648, +2 147 483 647**] ;
- **REEL** : représente le type REEL, la taille mémoire utilisée est de 4 octets. Ce type autorise les décimales. L'intervalle des valeurs autorisées est sensiblement [$\mathbf{-1.0\ 10^{38}}$, $\mathbf{-1.0\ 10^{-38}}$] ∪ [$\mathbf{1.0\ 10^{-38}}$, $\mathbf{-1.0\ 10^{38}}$] avec une précision de 6 décimales. Attention, l'intervalle défini ici peut varier suivent les langages de programmation ;
- **BOOLEEN** : représente le type LOGIQUE (Vrai / Faux). La taille mémoire utilisée est de 1 octet ;
- **CARACTERE** : représente le type des caractères issus de la table ASCII, la taille mémoire utilisée est de 1 octet. La table ASCII est une table normalisée qui associe à chaque caractère un code appelé code ASCII. De manière interne, l'ordinateur travaille avec le code et non avec la représentation du caractère.

4. NOTION DE VARIABLE

Dans notre exemple, nous avons déclaré trois variables : Valeur1,Valeur2 et Somme. En fait, l'intérêt d'une variable est de pouvoir mémoriser une information dont l'algorithme ou le programme aura besoin au cours d son exécution. De plus, son contenu peut varier à n'importe quel moment. En résumé, il faut déclarer une variable lorsque vous avez besoin de mémoriser une information.

Une variable se caractérise par :

- Une *étiquette* (un nom). Dans certains ouvrages, on parlera également d'identificateur ;
- Un *emplacement mémoire*. L'adresse en mémoire est associée au non de la variable. Manipuler le nom de la variable, c'est faire référence à son adresse mémoire. Rassurez-vous, c'est le compilateur qui réalise cette association, pas le programmeur ;
- Une taille mémoire déterminée par le type de la variable (NOMBRE, CARCTERE…) ;
- Elle ne possède aucune valeur initiale a priori. Attention, il ne faut jamais partir du principe qu'une variable est initialisée automatiquement à zéro. Certains langages de programmation le font, d'autres pas. Prenons donc le principe général qu'elle n'est jamais initialisée à zéro, cela peut éviter bien des surprises.

4.1 Déclaration d'une variable

Une variable mémorise une information d'un type donné. Mais en réalité, cela n'est possible que s'il y a une opération préalable qui consiste à déclarer la variable afin de permettre au programme, en cours d'exécution, de réserve une zone mémoire d'une certaine taille et de lui associer un nom pour pouvoir y accéder ultérieurement.
En algorithme, la déclaration des variables se fait dans le bloc matérialisé par le mot **Var**.

Exemple :

```
Programme TestDéclaration
Var
   LaValeur : NOMBRE
   CarLu : CARACTERE
Début
   ……
   ……
   ……
Fin
```

Nous venons de déclarer deux variables, l'une de type NOMBRE et l'autre type CARACTERE. La première alloue un emplacement de deux octets et l'autre d'un octet.

Remarques :

- Par convention, le nom d'une variable doit avoir la première lettre de chaque mot important la composant en majuscule ;

  Exemple : LaValeur et non pas lavaleur ni la_valeur

- Le nom des variables doit être significatif (parlant). Evitez d'appeler une variable « s » pour Somme ou « t » pour TotalAchat ;

- Il faut toujours déclarer une variable, même si certains langages de programmation peuvent les déclarer pour vous. Le type pris par défaut ne sera pas forcément le plus adapté ;

- Les variables se déclarent toujours en début de programme, même si, encore une fois, certains langages de programmation sont permissifs. En effets, lors des recherches d'erreurs (bugs) ou lors des phases de maintenance, la définition des variables est plus facilement localisée.

4.2 Affectation d'une variable

L'opération d'affectation consiste à placer dans la zone mémoire d'une variable (dont on précise le nom), une valeur compatible avec le type de la variable. La variable concernée et la valeur affectée doivent être de même type.
En pseudo code, nous matérialiserons l'affectation en dessinant une flèche qui part de la valeur affectée pour aller jusqu'au nom de la variable. Cela permet d'indiquer par un dessin, le sens du mouvement.

Exemple :

```
Programme TestDéclaration
   Var
     LaValeur : NOMBRE
Début
   ……….
     LaValeur ← 5
   ……….
Fin
```

Ces deux lignes permettent, pour la première, de déclarer une variable de type NOMBRE et pour la deuxième, d'affecter la valeur 5 dans la variable spécifiée.

Après la déclaration, il y a réservation d'une zone mémoire de 2 octets sans valeur initiale.

LaValeur

?????

Après l'opération d'affection, la zone mémoire Contient la valeur 5.

LaValeur

5

Exemple :

```
Programme TestDéclaration2
   Var
     LaValeur : NOMBRE
Début
   ……….
     LaValeur ← « Bonjour »
     LaValeur ← 5.12
   ……….
Fin
```

Les deux lignes d'affectation sont fausses car le type de la donnée affectée ne correspond pas au type de la variable LaValeur. La première ligne tente d'affecter une chaîne de caractères et la deuxième, un réel à une variable de type Nombre, c'est impossible.

5. CONSTANTE

5.1 Définition d'une constante

Une constante est un objet qui :

- **Ne possède pas d'emplacement mémoire** réservé. La valeur de la constante est directement placée dans le code du programme utilisant. En effet, où se trouve la constante 1 en mémoire ? Où se trouve la constante 2 en mémoire ? En fait, comme une constante ne possède pas d'emplacement mémoire, elle ne peut pas voir sa valeur modifiée ;
- Possède **un type** déterminé par sa valeur (NOMBRE, CARACTERE…).

Exemples :

- 3 est une constante de type NOMBRE ;
- 25.2 est une constante de type REEL ;
- VRAI ou FAUX sont des constantes de type BOOLEEN ;
- 'A' est une constante caractère. Elle se place entre quotes ;
- « Bonjour » est une constante chaîne de caractères. Elle se place entre guillemets.

5.2 Constante nommée

Une constante nommée est une constante à laquelle on associe un nom afin de rendre plus lisible un programme, mais aussi pour simplifier la maintenance. Par exemple, on peut utiliser la valeur **3.1416** dans un programme à chaque fois que l'on souhaite utiliser la valeur de **pi** mais on peut également définir le nom **PI** auquel on associe la valeur 3.1416. Cela nous permet d'employer le PI à chaque fois que l'on fait référence à la valeur de pi. Le compilateur remplacera chaque occurrence du nom PI par sa valeur lors de la phase de compilation.

En algorithmie, la déclaration des constantes nommées se fait dans un bloc matérialisé par le mot **Const**. Chaque nom de constante est, par convention, écrit en majuscule. Pour associer un nom de constante à sa valeur, il suffit d'employer la syntaxe :

NomConstante = ValeurConstante

Exemple : Ecrire un programme calculant la surface d'un cercle à partir de la valeur de rayon, saisie au clavier.

```
Programme CalculSurface
Const
    PI = 3.1416
Var
    Rayon, Surface : REEL
Début
    /* Saisir de la valeur du rayon */
    Afficher « Entrez la valeur du rayon »
    Saisir Rayon

    /* Effectuer le calcul de la surface du cercle */
    Surface ← PI * Rayon * Rayon

    /* Afficher le résultat */
    Afficher « La surface est », Surface, « cm² »
Fin
```

Exemple : Sachant que le prix d'une vache est de 250 € et que celui d'un mouton est de 120 €, écrire un algorithme qui vous demande de saisir le nombre de vaches et le nombre de moutons puis qui affiche le prix du troupeau ainsi constitué.

```
Programme PrixTroupeau
Var
    NBVaches, NBMoutons : NOMBRE
    PrixTroupeau : LONG
Début
    /* Saisie des deux valeurs*/
    Affiche « Entrez le nombre de vaches »
    Saisir NBVaches

    Affiche « Entrez le nombre de moutons »
    Saisir NBMoutons

    /* Effectuer le calcul du prix du troupeau */
    PrixTroupeau ← NBVaches * 250 + NBMoutons * 120

    /* Afficher le résultat */
    Afficher « Le prix du troupeau est : », PrixTroupeau, « euros »
Fin
```

Autre version en utilisant des constantes nommées :

```
Programme PrixTroupeau
Const
    PRIX_VACHE = 250
    PRIX_MOUTON =120

Var
    NBVaches, NBMoutons : NOMBRE
```

```
    PrixTroupeau : LONG
Début
    /* Saisie des deux valeurs*/
    Affiche « Entrez le nombre de vaches »
    Saisir NBVaches

    Affiche « Entrez le nombre de moutons »
    Saisir NBMoutons

    /* Effectuer le calcul du prix du troupeau */
    PrixTroupeau ← NBVaches * PRIX_VACHE + NBMoutons * PRIX_MOUTON

    /* Afficher le résultat */
    Afficher « Le prix du troupeau est : », PrixTroupeau, « euros »
Fin
```

Cette deuxième version est plus lisible et plus facile à maintenir car si les prix change, il suffit de modifier la valeur des constantes à un endroit déterminé. Cela évite de perdre du temps à rechercher dans tout le programme les endroits à modifier au risque d'en oublier.

Exemple : Ecrire un programme qui demande à l'utilisateur de saisir le prix en euro d'une bouteille d'eau minérale de 1,5 L. Dans un deuxième temps, le programme affiche le prix au litre, le prix d'un pack d'eau, sachant qu'il y a 6 bouteilles, dont une est offerte, ainsi que le prix d'une palette de packs d'eau, sachant qu'il y a 50 packs dans une palette dont 5 sont offerts.

```
Programme CalculPrix
Var
    PrixBouteille, PrixLitre, PrixPack : REEL
Début
    /* Saisir le prix d'une bouteille d'eau */
    Affiche « Entrez le prix d'une bouteille d'eau de 1,5L »
    Saisir PrixBouteille

    /* Effectuer les calculs */
    PrixLitre ← (PrixBouteille*2)/ 3
    PrixPack ← PrixBouteille*5

    /* Afficher le résultat * /
    Afficher « Le prix au litre est de », PrixLitre, « Euro(s) »
     Afficher « Le prix d'un pack est de », PrixPack, « Euro(s) »
    Afficher « Le prix d'une palette est de », PrixPack*45, « Euro(s) »
Fin
```

Autre version en utilisant des constantes nommées :

```
Programme CalculPrix
Const
NB_BOUTEILLES_PACK = 6
NB_BOUTEILLES_OFFERTES = 1
NB_PACKS_PAR_PALETTE = 50
NB_PACKS_OFFERTS = 5
NB_BOUTEILLES_PAYANTES = NB_BOUTEILLES_PACK - NB_BOUTEILLES_OFFERTES
NB_PACKS_PAYANTS = NB_PACKS_PAR_PALETTE - NB_PACKS_OFFERTS
Var
    PrixBouteille, PrixLitre, PrixPack : REEL
Début
    /* Saisir le prix d'une bouteille d'eau */
    Affiche « Entrez le prix d'une bouteille d'eau de 1,5L »
    Saisir PrixBouteille

    /* Effectuer les calculs */
    PrixLitre ← (PrixBouteille*2)/ 3
    PrixPack ← PrixBouteille*5

    /* Afficher le résultat * /
    Afficher « Le prix au litre est de », PrixLitre, « Euro(s) »
     Afficher « Le prix d'un pack est de », PrixPack, « Euro(s) »
    Afficher « Le prix d'une palette est de », PrixPack* NB_PACKS_PAYANTS, « Euro(s) »
Fin
```

Dans cette deuxième version, nous voyons que des constantes nommées peuvent servir à déterminer la valeur d'autres constantes nommées. Ainsi, si on modifie la valeur de l'une des quatre premières constantes car les conditions de vente ont changé, les autres constantes sont réévaluées automatiquement au début de la phrase de compilation du programme.

6. OPERATEURS ET EXPRESSIONS

Une expression est une alternance d'*opérandes* et d'*opérateurs.* Elle fournit toujours un résultat et un seul.

Voici un exemple d'expression : **1 + 2**
Dans cette expression, **1** et **2** sont des opérandes et + est un opérateur. En résumé, les opérandes sont les éléments sur lesquels porte l'opérateur et un opérateur décrit l'action que l'on souhaite réaliser sur les opérandes se trouvant de part et d'autre.

Autre exemple : **Valeur + 1**

Dans cette expression, on prend le contenu de la variable de nom « Valeur » et on ajoute « 1 ».

Les opérandes peuvent être des valeurs ou des variables. Dans ce dernier cas, il y aura une opération préliminaire, qui n'est pas à la charge du programmeur, qui consiste à remplacer le nom de la variable par son contenu au moment du calcul afin de pouvoir obtenir le résultat de l'expression dans un deuxième temps (cf . exemple précédent).

6.1 Opérateur mathématiques

Addition : +	Multiplication : *
Soustraction : -	Division : /

Dans le cas de la division, le résultat obtenu sera du type « Réel », c'est à dire éventuellement avec des décimales, si l'un des opérateurs est du type « Réel ». Ainsi :

5.0 / 2.0	donne 2.5
5.0 / 2	donne 2.5
5 / 2.0	donne 2.5
5 / 2	donne 2

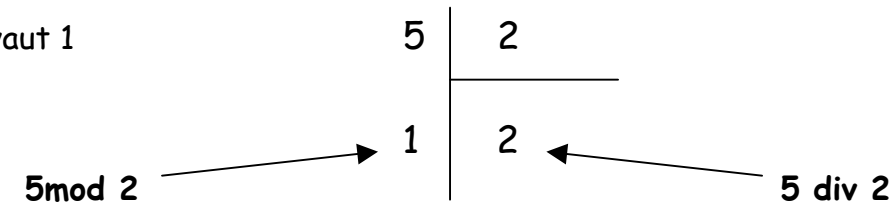
Il existe encore deux autres opérateurs particuliers :

Division entière : **div**
Reste de la division entière : **mod**

Exemples :

5 div 2 vaut 2 (on obtient le même résultat que 5 / 2 à la seule différence près que l'opérateur **div** est plus explicite sur le fait que le résultat sera de type entier)

5 mod 2 vaut 1



Remarque :

En général, une expression est associée à une opération d'affection afin de mémoriser le résultat obtenu dans une variable en vue d'un traitement ultérieur. Ainsi a ← a + 1 augmente le contenu de la variable « a » de « 1 ».
En fait, on prend l'ancienne valeur de « a » et on lui ajoute « 1 ». Le résultat obtenu est réaffecté à la variable « a » ce qui a pour conséquence d'écraser sa valeur précédente. En effet, une variable ne peut contenir qu'une et une seule valeur à la fois.

Somme ← Valeur1 + Valeur2 prend le contenu de la variable Valeur1 que l'on additionne au contenu de Valeur2. Le résultat obtenu est placé dans la variable Somme.

6.2 Expression logique et opérateurs de comparaison

Une expression logique, également appelée « condition », est une expression faisant intervenir des opérateurs de comparaison et ayant pour résultat VRAI ou FAUX (TRUE ou FALSE). Elle permet de comparer deux valeurs de même nature à savoir 2 entiers, 2 réels, 2 caractères ou 2 chaînes de caractères.

Remarque :

Les valeurs VRAI et FAUX sont des valeurs de type booléen.

Voici la liste des opérateurs de comparaison utilisables en algorithmie :

Egalité : =	Supérieur : >
Différent : <>	Inférieur ou égal : <=
Intérieur : <	Supérieur ou égal : >=

Exemples d'expressions logiques :

2 < 3	réponse : VRAI
10 >= 15	réponse : FAUX
'A' < 'E'	réponse : VRAI

Dans ce dernier cas, la comparaison se fait par rapport au code ASCII des caractères, sachant que dans la table ASCII, 'A' a pour code la valeur 65 et 'E' a

pour code la valeur 69. En fait, l'ordre alphabétique est respecté dans les codes ASCII attribués aux lettres.

'a' > 'A' réponse : VRAI

La comparaison se fait par rapport au code ASCII des caractères, sachant que dans la table ASCII, 'a' a pour code la valeur 97 et 'A' a pour code la valeur 65. Les majuscules et les minuscules ont des codes ASCII différents. Les majuscules commencent à partir de la valeur 65 et les minuscules commencent à la valeur 97.

« Bonjour » > « Au revoir » réponse : VRAI

La comparaison se fait par rapport à l'ordre alphabétique.

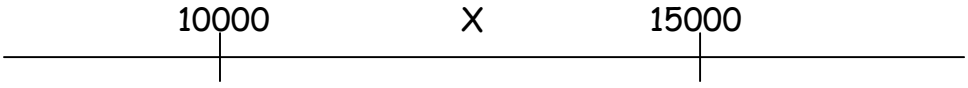
Supposons que nous ayons deux variables Val1 et Val2 de type NOMBRE contenant respectivement les valeurs 7 et 5.

Val1 > Val2 réponse : VRAI. En effet, on compare le contenu des 2 variables.

6.3 Combiner des expressions logiques ou conditions multiples

Nous verrons dans le paragraphe concernant les instructions de contrôle qu'il peut être parfois intéressant de combiner plusieurs expressions logiques pour valider ou non une condition plus générale.

On souhaite par exemple savoir si la valeur contenue dans ma variable de nom **LaValeur** est comprise entre 10000 et 15000. Pour cela, il faut regarder si cette valeur est supérieure ou égale à 10000 et inférieure ou égale à 15000.



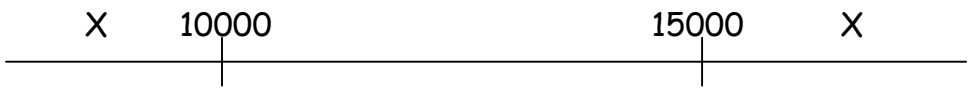
Ainsi l'expression logique générale doit être :

LaValeur >= 10000 **Et** LaValeur <= 15000

Les deux conditions doivent être vérifiées pour certifier que le contenu de la variable LaValeur est entre 10000 et 15000. Nous avons employé un opérateur

logique Et qui impose que les deux conditions de part et d'autre de l'opérateur soient vraies pour avoir un résultat vrai.

On souhaite maintenant savoir si la valeur contenue dans la variable de nom **LaValeur** n'est pas comprise entre 10000 et 15000. Pour cela, il faut regarder si cette valeur est inférieure à 10000 ou supérieure à 15000.



Ainsi l'expression logique générale doit être :

LaValeur < 10000 **ou** LaValeur > 15000

Dans ce deuxième cas, il suffit que l'une des deux conditions soit vérifiée pour certifier que le contenu de la variable LaValeur n'est pas compris entre 10000 et 15000. Nous avons employé un opérateur logique Ou qui demande que l'une des deux conditions de part et d'autre de l'opérateur soient vraie, et seulement l'une d'elles, pour avoir un résultat vrai.

Voici la table de vérité du Et

Et	Vrai	Faux
Vrai	Vrai	Faux
Faux	Faux	Faux

La seule manière d'avoir VRAI et d'avoir les deux conditions à VRAI.

Voici la table de vérité du Ou

Et	Vrai	Faux
Vrai	Vrai	Vrai
Faux	Vrai	Faux

La seule manière d'avoir FAUX et d'avoir les deux conditions à FAUX.

Exemples :

Nous disposons de 4 variables de type NOMBRE :

Val1, Val2, Val3, Val4 : NOMBRE

Nous réalisons les affectations suivantes :

Val1 ← 2
Val2 ← 10
Val3 ← 15
Val4 ← 5

Analysons les expressions logiques suivantes :

(Val1 < Val2) Et (Val3 < Val4)
Cette expression a pour résultat : FAUX. En effet la première condition est vraie et la deuxième condition est fausse, ce qui donne : VRAI Et FAUX donc FAUX.

(Val1 < Val2) Ou (Val3 < Val4)
Cette expression a pour résultat : VRAI. En effet la première condition est vraie et la deuxième condition est fausse, ce qui donne : VRAI Ou FAUX donc VRAI.

(Val1 < Val2) Et ('A' < 'B')
Cette expression a pour résultat : VRAI. En effet la première condition est vraie et la deuxième condition est fausse, ce qui donne : VRAI Et VRAI donc VRAI.

(Val1 < Val2) Et (Val3 < Val4) Ou (« Gorki » < « pedra »)
Cette expression a pour résultat : VRAI. L'évolution des différentes expressions logiques s'effectue de la gauche vers la droite. La première condition est vraie, la deuxième condition est fausse, ce qui donne un premier résultat intermédiaire : FAUX. En effet, VRAI Et FAUX donne FAUX. Ce résultat intermédiaire est combiné avec le résultat de la troisième expression logique qui vaut : VRAI. Le résultat global est donc FAUX Ou VRAI soit VRAI.

Remarque :
Les parenthèses autour des conditions sont facultatives mais elles facilitent la lecture.
Dans le dernier exemple, nous avons vu que l'évaluation des expressions logiques s'effectue de la gauche vers la droite. Il est tout de même possible d'influencer l'ordre des évaluations en ajoutant des parenthèses supplémentaires.

> ((40 < 30) Et (10 < 20)) Ou ('A' < 'B')
> Cette expression a pour résultat : VRAI. Le groupe de conditions situé à gauche donne le résultat intermédiaire FAUX, la dernière condition donne le résultat VRAI. Le résultat global est donc FAUX Et VRAI soit FAUX.
>
> (40 < 30) Et ((10 < 20) Ou ('A' < 'B'))
> Cette expression a pour résultat : FAUX. La première condition donne le résultat intermédiaire FAUX, le groupe de conditions qui suit donne un résultat VRAI. Le résultat global est donc FAUX Et VRAI soit FAUX.

6.4 Négation

L'algorithmie propose un opérateur qui permet de calculer la négation d'une condition. Il s'agit de l'opérateur Non. Voici la table de vérité de cet opérateur :

> Non **VRAI** donne **FAUX**
> Non **FAUX** donne **VRAI**

Dans certains cas, cet opérateur permet de simplifier l'écriture des conditions. Reprenons un exemple vu précédemment qui permettait de savoir si la valeur contenue dans la variable de nom **LaValeur** n'est pas comprise entre 10000 et 15000.
Il est facile de mettre en place la condition qui détermine si le contenu de la variable est dans l'intervalle. Cette condition est :

> LaValeur >= 10000 Et LaValeur <= 15000

Pour déterminer maintenant si la valeur est en dehors de l'intervalle, il suffit de prendre le contraire de la condition précédente, ce qui donne **Non** (LaValeur >= 10000 et LaValeur <=15000).

En effet, si je ne suis pas dans l'intervalle, c'est que je suis en dehors.

7. INSTRUCTIONS SIMPLES

Pour faire vivre un algorithme, il est nécessaire de mettre en place des instructions afin de préciser les actions que l'ordinateur doit exécuter. Il existe deux familles d'instructions : les *instructions simples* et les *instructions de contrôle.*

Une instruction simple est une instruction élémentaire, non décomposable. Bien souvent dans un langage de programmation, une instruction simple est suivie d'un point-virgule. En algorithmie, cela n'est pas nécessaire.

Exemple :

LaValeur ← 5
Cette instruction ne peut pas être décomposée en sous-instructions.

7.1 Instruction d'entrée / sortie

Ces instructions permettent un échange d'information entre le programme et l'utilisateur. L'instruction de lecture permet à l'utilisateur de saisir une donnée au clavier afin que le programme la traite. L'instruction d'écriture permet au programme d'afficher à l'écran des informations ou des résultats à destination de l'utilisateur.

7.1.1 Instruction de lecture des données au clavier

Cette instruction permet à l'utilisateur de saisir des données qui seront stockées dans des variables en vue d'un traitement ultérieur. Voici sa syntaxe :

Saisir <Liste de variables>

Les différentes variables placées dans la liste seront séparées les unes des autres par une virgule.

Exemple : Soient deux variables de type NOMBRE

Valeur1, Valeur2 : NOMBRE
Saisir Valeur1
Saisir Valeur2

Ou

Saisir Valeur1, Valeur2

Remarque : Dans ce dernier cas, les deux valeurs saisies doivent être séparées par un espace.
Ces deux formulations sont identiques. Dans les deux cas :

- A la rencontre de l'instruction **Saisir,** le programme effectue une pause et affiche un curseur clignotant indiquant à l'utilisateur qu'il doit effectuer une saisie ;
- Une fois que l'utilisateur a saisi les données souhaitées, il appuie sur la touche « Entrée », ce qui permet au programme de reprendre son exécution ;
- Les données saisies sont alors affectées aux variables dont le nom est précisé derrière l'instruction **Saisir**. Si la nature des données saisies ne correspond pas à la nature des variables affectées, une erreur se déclenche. En général, les langages de programmation proposent des moyens pour pallier au « plantage » du programme en cas d'erreur de saisie.

7.1.2 Instruction d'affichage des informations à l'écran

Cette instruction affiche à l'écran du texte, le résultat d'une expression ou encore le contenu d'une variable. Afficher de l'information à l'écran est capital pour la convivialité des programmes, sinon l'utilisateur ne sait pas ce que le programme attend de lui, ni où il en est dans son exécution. Voici sa syntaxe :

Afficher <Liste de textes et de variables>

Les différents éléments placés dans la liste seront séparés les uns des autres par une virgule. Les textes seront placés entre guillemets. Si le nom d'une variable est placé dans la liste des éléments, le programme affiche son contenu à l'écran.

Exemple : Dans l'algorithme qui suit, l'utilisateur va saisir un entier puis le programme affichera la valeur saisie.

```
Programme TestSaisieAffichage
Var
    ValeurLue : NOMBRE /* Valeur saisie au clavier */
Début
    /* Afficher un texte pour l'utilisateur */
    Afficher « Entrez une valeur numérique »
```

```
    /* Saisir la valeur au clavier */
    Saisir ValeurLue

    /* Afficher à l'écran la valeur lue */
    Afficher « La valeur saisie est », ValeurLue
Fin
```

En détaillant ce programme :

- La première instruction affiche un texte à l'écran ;
- La deuxième instruction permet à l'utilisateur d'entrer une donnée dans la variable ValeurLue ;
- La troisième instruction affiche un texte suivi du contenu de la variable ValeurLue.

8. INSTRUCTION DE CONTROLE

Les instructions de contrôler le déroulement du programme en fonction des résultats d'un test de condition.

8.1 Structure conditionnelle : Si

```
Si Condition Alors
        Instruction « Alors »
Fin si
```

Ce qui se traduit par : Si la condition est vraie alors
Exécuter les instructions du bloc « Alors »

Que les instructions du bloc **Alors** soient exécutées ou non, le programme continue normalement son exécution après l'instruction **Fin si**. Le programme ne revient plus sur le test du **Si** une fois la condition testée. Un **Si** n'est pas une répétitive.

Exemple :

```
Programme ValeurAbsolue
Var
    ValeurLue : NOMBRE /* Valeur saisie au clavier * /
Début
    Afficher « Entrez une valeur numérique »
    Saisie ValeurLue

    /* Effectuer le test pour savoir si la valeur lue est négative */
    Si ValeurLue < 0 Alors
```

```
            ValeurLue ← - ValeurLue /* On inverse le signe */
        Fin si

        Afficher « La valeur absolue est », ValeurLue
    Fin
```

L'intérêt de l'instruction conditionnelle **Si** est que s'il n'y a pas d'instruction à exécuter dans le cas où la condition est fausse, nous ne sommes pas obligés de mettre du code pour rien. C'est le cas dans notre exemple, si la valeur lue est positive, il n'y a rien à faire, par contre si elle est négative, il faut inverser le signe pour la rendre positive.

Remarques :

- La condition d'un **Si** peut être multiple (cf. paragraphe 6.3 dans ce cours) ;
- Les instructions conditionnelles peuvent être imbriquées (cf. paragraphe suivant) ;
- Pour faciliter la lecture, n'oubliez pas l'indentation (décalage) des instructions du bloc **Alors**.

8.2 Structure alternative : le SiSinon

```
Si Condition Alors
    Instruction « Alors »
Sinon
    Instruction « Sinon »
Fin si
```

Ce qui se traduit par :

```
Si la condition est vraie alors
    Exécuter les instructions du bloc « Alors »
Sinon
    Exécuter les instructions du bloc « Sinon »
```

Que les instructions du bloc **Alors** ou **bloc Sinon** soient exécutées ou non, le programme continue normalement son exécution après l'instruction **Fin si**. Le programme ne revient plus sur le test du **SiSinon** une fois la condition testée. Un **SiSinon** n'est pas une répétitive.

Exemple : Nous souhaitons calculer la différence entre deux valeurs entrées au clavier. La différence devra toujours être positive.

```
Programme DifférencePositive
Var
    Valeur1, Valeur2 : NOMBRE /* Valeurs saisies au clavier */
    Résultat : NOMBRE /* Contient le résultat à afficher */
Début
    Afficher «Entrez deux valeurs numériques »
    Saisir Valeur1
    Saisir Valeur2

    /* Effectuer le calcul de la différence positive */
    Si Valeur1 < Valeur2 Alors
        Resultat ← Valeur2 – Valeur1
    Sinon
        Resultat ← Valeur1 – Valeur2
    Fin si

    Afficher « La différence est de », Resultat
Fin
```

Remarques :

- La condition d'un **SiSinon** peut être multiple (cf. paragraphe 6.3 dans ce cours) ;
- Le bloc des instructions « Alors » commence après le mot **Alors** et se termine sur le **Sinon**. Le bloc des instructions « Sinon » commence après le mot **Sinon** et se termine sur le **Fin si** ;
- Pour faciliter la lecture, n'oubliez pas l'indentation des instructions du bloc « Alors » et du bloc « Sinon ».

Il est possible d'imbriquer des structures alternatives et conditionnelles afin de pouvoir traiter différents cas de figure. Il n'y a pas de limitation dans la profondeur des imbrications sauf peut-être par rapport à la lisibilité du programme.

Exemple : L'utilisateur saisit deux entiers. Le programme affiche une phrase d'information sur la comparaison des deux valeurs saisies.

```
Programme Comparaison2Entiers
Var
    Valeur1, Valeur2 : NOMBRE /* Valeurs saisies au clavier */
Début
    Afficher « Entrez deux valeurs numériques »
    Saisir Valeur1
    Saisir Valeur2

    /* Effectuer le test de comparaison */
    Si Valeur1 < Valeur2 Alors
       Afficher « La valeur la plus grande est », Valeur2
```

```
        Sinon
            Si Valeur1 > Valeur2 Alors
                    Afficher « La valeur la plus grande est », Valeur1
            Sinon
                    Afficher « Les deux valeurs sont égales »
            Fin si
        Fin si
Fin
```

Remarques :

- L'indentation des différents blocs est primordiale ;
- N'oubliez pas les différents **Fin si**.

Conseil : dès que vous tapez le **Si**, tapez le **Fin si** associé pour ne pas le perdre de vue.

8.3 Structure répétitive : Tant que Faire

Tant que Condition **Faire**
    Instructions
Fin tant que

Ce qui se traduit par : Tant que la condition est vraie
Exécuter les instructions

Cette fois-ci, il s'agit d'une instruction de répétition, appelée également une boucle. Le programme va tester avant chaque tour de boucle si la condition est vraie, dans l'affirmative il exécute les instructions jusqu'au **Fin tant que**, puis il revient au test de la condition. Lorsque la condition devient fausse, la boucle **Tant que Faire** se termine et le programme continue normalement son exécution après l'instruction qui suit le **Fin Tant que**.

Exemple : Nous souhaitons que l'utilisateur saisisse une valeur positive. Pour s'en assurer, nous allons boucler tant que la valeur saisie n'est pas correcte.

```
Programme SaisirPositif
Var
    ValeurLue : NOMBRE /* Valeur saisie au clavier */
Début
    Afficher « Entrez une valeur numérique positive »
    Saisir ValeurLue

    /* Boucler tant que la valeur lue n'est pas correcte */
    Tant que ValeurLue < 0 Faire
        Afficher « Erreur ; Entrez une valeur positive »
        Saisir ValeurLue
    Fin tant que
```

```
    Afficher « La valeur saisie est », ValeurLue
Fin
```

Exemple : On demande à l'utilisateur de saisir une série de nombres entiers positifs ou négatifs. La saisie s'arrête lorsque l'utilisateur tape 0 (zéro). Lorsque la saisie est terminée, le programme affiche la somme des valeurs.

```
Programme  FaireSomme
Var
    ValeurLue : NOMBRE /* Valeur saisie au clavier */
    LaSomme : NOMBRE /* Contient la somme des valeurs lues * /

Début
    /* Initialiser la variable de cumul */
    LaSomme ← 0

    /* Effectuer la saisie et le cumul */
    Afficher « Entrez une série de valeurs »
    Saisir ValeurLue

    /* Boucler tant que la valeur lue n'est pas zéro et faire le cumul */
    Tant que  ValeurLue <> 0 Faire
        LaSomme ← LaSomme + ValeurLue
        Saisir ValeurLue
    Fin tant que

    Afficher « La somme des valeurs saisies est », LaSomme
Fin
```

Remarques :

- Nous déclarons deux variables car nous avons besoin de mémoriser la valeur saisie ainsi que la somme à chaque tour de boucle ;
- La première instruction réalise l'initialisation de la variable **LaSomme** car n'oubliez pas qu'une variable n'a pas de valeur déterminée au départ. Elle contient n'importe quoi. Il faut s'assurer que l'on démarre la somme à partir de zéro. Attention, il est vrai que certains langages de programmation initialisent les variables à 0 comme Visual Basic, mais n'oublions pas que dans le cadre de l'algorithmie, nous nous détachons de toutes les spécificités des langages. De toute manière, il est toujours plus prudent de réaliser cette opération, elle n'est pas gourmande en temps d'exécution ;
- La première instruction à l'intérieur du **Tant que** réalise le cumul des valeurs saisies à chaque tour de boucle. En effet, on prend le contenu actuel de la variable **LaSomme**, on lui ajoute le contenu

de la variable **ValeurLue** et on réaffecte le tout à la variable **LaSomme** afin d'obtenir un nouveau cumul ;

- La deuxième instruction à l'intérieur du **Tant que** permet la saisie d'une nouvelle valeur afin de la traiter. Sans cette ligne, le contenu de la variable **ValeurLue** ne changerait jamais, ce que donnerait un programme qui boucle de manière infinie. En effet, s'il n'y a pas de nouvelle saisie, jamais le contenu de **ValeurLue** ne pourra devenir zéro.

Exemple : On demande à l'utilisateur de saisir une série de nombres entiers positifs ou négatifs. La saisie s'arrête lorsque l'utilisateur tape la valeur 0 (zéro). Lorsque la saisie est terminée, le programme affiche le nombre de valeurs positives et le nombre de valeurs négatives saisies.

```
Programme Comptage
Var
    ValeurLue : NOMBRE /* Valeur saisie au clavier */
    CptPositifs, CptNégatifs : NOMBRE /* 2 Compteurs de valeurs */

Début
    /* Initialiser les variables compteurs */
    CptPositifs ← 0
    CptNégatifs ← 0

    /* Effectuer la saisie */
    Afficher « Entrez une série de valeurs »
    Saisir ValeurLue

    /* Boucler tant que la valeur lue n'est pas zéro et faire le dénombrement */
    Tant que ValeurLue <> 0 Faire
        Si ValeurLue > 0 Alors
            CptPositifs ← CptPositifs + 1
        Sinon
            CptNégatifs ← CptNégatifs + 1
        Fin si
        Saisir ValeurLue
    Fin tant que

    Afficher « Le nombre de valeurs positives est : », CptPositifs
    Afficher « Le nombres de valeurs négatives est : », CptNégatifs
Fin
```

Remarques : Dans cet exemple, nous avons imbriqué un **Si** dans un **Tant que**, ce qui permet à chaque tour de boucle de vérifier si la valeur est positive ou négative et d'incrémenter la variable concernée. Il est possible également d'imbriquer des **Tant que**.

Attention pour le bon fonctionnement d'une boucle :

1. Il faut vérifier qu'à un moment on peut entrer dans la boucle pour y exécuter les instructions, ceci afin d'éviter les boucles inutiles.
2. Il faut vérifier également qu'à un moment on sort de cette boucle. Il faut éviter les boucles infinies .

Exemples :

```
Programme SaisirValeurPositive
Var
   ValeurLue : NOMBRE /* Valeur saisie au clavier */
Début
   Afficher « Entrez une valeur numérique positive »

   Tant que ValeurLue < 0 Faire
      Saisir ValeurLue
   Fin tant que

   Afficher « La valeur saisie est », ValeurLue
Fin
```

On n'entrera peut-être jamais dans la boucle car quelle est la valeur initiale de la variable **ValeurLue** ?

Rappel : une variable n'a pas de valeur initiale définie par défaut. Ainsi, on n'est pas maître du test de la condition dans cet exemple. C'est une boucle à éliminer ou alors il faut initialiser la variable.

```
Programme SaisirValeurPositive
Var
   ValeurLue : NOMBRE /* Valeur saisie au clavier */
Début
   ValeurLue ← 0
   Afficher « Entrez une valeur numérique positive »

   Tant que ValeurLue < 0 Faire
      Saisir ValeurLue
   Fin tant que

   Afficher « La valeur saisie est », ValeurLue
Fin
```

Cette fois-ci, on n'entrera jamais dans la boucle **Tant que**. En effet, on a initialisé la variable **ValeurLue** à zéro en croyant bien faire. Mais en fait, la valeur initiale n'étant pas négative dès le départ, on n'entre pas dans la boucle.

Il ne faut pas initialiser une variable avec n'importe quelle valeur. Comment résoudre ce problème. Il y a deux solutions.

```
Programme SaisirValeurPositive
Var
   ValeurLue : NOMBRE /* Valeur saisie au clavier */
Début
   Saisir ValeurLue
   Afficher « Entrez une valeur numérique positive »

   Tant que ValeurLue < 0 Faire
      Saisir ValeurLue
   Fin tant que

   Afficher « La valeur saisie est », ValeurLue
Fin
```

Dans cette première solution, on effectue une saisie et si celle-ci est mauvaise, on exécute les instructions de la boucle **Tant que**. On entrera donc bien un jour dans la boucle. Quand ? Dès que l'utilisateur tapera une valeur incorrecte dès le début.

```
Programme SaisirValeurPositive
Var
   ValeurLue : NOMBRE /* Valeur saisie au clavier */
Début
   ValeurLue ← 1
   Afficher « Entrez une valeur numérique positive »

   Tant que ValeurLue < 0 Faire
      Saisir ValeurLue
   Fin tant que

   Afficher « La valeur saisie est », ValeurLue
Fin
```

Dans cette seconde solution, on affecte à la variable une valeur négative pour rendre le test de la condition du **tant que** vrai dès le départ et donc exécuter les instructions de la boucle.

L'une et l'autre des solutions sont correctes. Dans certains algorithmes, l'une sera plus facile à mettre en œuvre que l'autre et dans d'autres cas se sera l'inverse.

```
Programme AfficherValeurs
Var
```

```
    LaValeur : NOMBRE
Début
    LaValeur ← 10
    Tant que LaValeur > 1 Faire
        LaValeur ← LaValeur + 1
        Afficher LaValeur
    Fin tant que
Fin
```

On ne sortira jamais de cette boucle. En effet, en partant de 10 et en augmentant la valeur à chaque tour de boucle, on sera toujours supérieur à 1.

C'est une boucle à éliminer ou à corriger, par exemple, en diminuant la valeur de la variable **LaValeur** à chaque tour de boucle.

Attention, les cas ne seront pas toujours aussi simples.

Remarques :

- La condition d'un **Tant que Faire** peut être multiple (cf. paragraphe 6.3 dans ce cours) ;
- L'indentation des différents blocs est primordiale.

Il faudra toujours vérifier que :

- Les variables intervenant dans le test de la condition sont correctement initialisées avant le premier test de la condition afin d'être sûr d'exécuter un jour les instructions du **Tant que** ;
- Que le test de la condition est logique pour s'assurer qu'on n'a pas une boucle infinie ;
- Que les instructions contenues à l'intérieur du **Tant que** fassent évoluer les variables intervenant dans la condition du **Tant que** afin de sortir un jour de la boucle. Il faut éviter les boucles infinies.

8.4 Structure répétitive : Faire Tant que

```
Faire
    Instructions
Tant que Condition
```

Ce qui se traduit par : Exécuter les instructions
Tant que la condition est vraie

On exécute une première fois les instructions puis on effectue le test de la condition pour éventuellement reprendre l'exécution des instructions. Lorsque la condition devient fausse, la boucle **Faire Tant que** se termine et le programme continue normalement son exécution après l'instruction **Tant que Condition**. Cette instruction de contrôle évite de devoir mettre en place une ou plusieurs instructions préliminaires nous assurant qu'on entrera bien un jour dans la boucle (cf. paragraphe précédent).

Reprenons notre exemple sur la saisie d'un nombre obligatoirement positif.

```
Programme SaisirValeurPositive
Var
   ValeurLue : NOMBRE /* Valeur saisie au clavier */
Début
   Afficher « Entrez une valeur numérique positive »
   Faire
         Saisir ValeurLue
   Tant que ValeurLue < 0

   Afficher « La valeur saisie est », ValeurLue
Fin
```

Remarques :

- Dans un **Tant que faire**, on exécute de 0 à « n » fois les instructions du bloc considéré. En effet, dès le premier test, la condition peut être fausse, auquel cas on n'exécute pas les instructions. A l'opposé, on peut exécuter un certain nombre de fois les instructions du bloc, tant que la condition reste vraie ;
- Dans un **Faire Tant que**, on exécute de 1 à « n » fois les instructions du bloc considéré. En effet, le programme exécuter de nouveau un certain nombre de fois les instructions, tant que la condition reste vraie ;
- La condition d'un **Faire Tant que** peut être multiple (cf. paragraphe 6.3 dans ce cours).

8.5 Structure répétitive : Répéter Jusqu'à

```
Répéter
   Instructions
Jusqu'à condition
```

Ce qui se traduit par : Exécuter les instructions
Jusqu'à ce que la condition soit VRAIE

On exécute une première fois les instructions puis on effectue le test de la condition pour éventuellement reprendre l'exécution des instructions. Lorsque la condition devient vraie, la boucle **Répéter Jusqu'à** se termine et le programme continue normalement son exécution après l'instruction **Jusqu'à condition**. C'est le contraire de la boucle précédente.

Reprenons notre exemple sur la saisie d'un nombre obligatoirement positif.

```
Programme SaisirValeurPositive
Var
   ValeurLue : NOMBRE /* Valeur saisie au clavier */
Début
   Afficher « Entrez une valeur numérique positive »
   Faire
         Saisir ValeurLue
   Tant que ValeurLue >= 0

   Afficher « La valeur saisie est », ValeurLue
Fin
```

Remarques :

- La différence entre un **Faire tant que** et un **Répéter jusqu'à** ne se situe qu'au niveau de l'écriture de la condition. En effet, dans le premier cas, la condition doit devenir fausse pour quitter la boucle alors que dans le second cas, il faut qu'elle devienne vraie ;
- Dans un **Répéter jusqu'à**, on exécute de 1 à « n » fois les instructions du bloc considéré. En effet, le programme exécute au moins une fois les instructions avant de faire le test de la condition, puis on peut exécuter de nouveau un certain nombre de fois les instructions, jusqu'à ce que la condition devienne vraie ;
- La condition d'un **Répéter jusqu'à** peut être multiple (cf. paragraphe 6.3 dans ce cours).

8.5 Structure répétitive : Pour

```
Pour NomVariableDeBoucle ← ValDébut à ValFin Par Pas de ValPas
      Instructions
Fin Pour
```

Ce qui se traduit par : Pour une variable de boucle dont le contenu varie de ValDébut à ValFin par Pas de ValPas
Exécuter les instructions

L'instruction de contrôle **Pour** permet d'exécuter les instructions un nombre déterminé de fois. Ce nombre est connu avant de commencer à exécuter le **Pour**. La boucle ne s'arrêtera que lorsque le nombre de répétition des instructions demandé a été réalisé. Lorsque le nombre de tours de boucle est terminé, le programme continue normalement son exécution après l'instruction après l'instruction **Fin Pour**.

Exemple : Afficher la table de multiplication par 10

```
Pour i ← 1 à 10 par pas de 1
        Afficher i * 10
Fin Pour
```

Le contenu de la variable i va varier de 1 à 10 par « pas » de 1. A chaque tour de boucle, on exécute les instructions, à savoir afficher la valeur de **i** multipliée par 10.

Dans une instruction **Pour**, la valeur du « Pas d'incrémentation » est de 1 par défaut, donc notre exemple peut se réduire à :

```
Pour i ← 1 à 10
        Afficher i * 10
Fin Pour
```

Exemple : Afficher les valeurs impaires situées entre 1 à 99. Pour cela, nous allons afficher les valeurs de deux en deux en commençant à 1.

```
Programme AfficheValeursImpaires
Var
        Valeur : NOMBRE /* Valeur saisie au clavier */
Début
        Pour Valeur ← 1 à 99 par pas de 2
                Afficher Valeur
        Fin Pour
Fin
```

Remarques :

- La variable utilisée dans l'instruction **Pour** qui permet de passer en revue les différentes valeurs prévues s'appelle la **variable de boucle** ;
- La variable de boucle doit être déclarée dans la liste des variables. En général, le nom d'une variable de boucle se réduit simplement à une lettre, par exemple **i** ;
- La valeur initiale de la variable de boucle lui est affectée avant le premier tour de boucle. Il s'agit de ValDébut ;
- Avant chaque tour de boucle, la valeur de la variable de boucle est comparée à la valeur de fin afin de déterminer s'il y a encore un tour de boucle à faire ;
- La valeur de la variable de boucle d'incrémente ou se décrémente automatiquement à la fin de chaque tour de boucle, c'est-à-dire **après** avoir exécuter les instructions du bloc **Pour** ;
- La valeur d'incrémentation ou de décrémentation dépend du **Pas** ;
- A l'intérieur des instructions du bloc **Pour**, la valeur de la variable de boucle est consultable mais pas modifiable. En effet, si on souhaite modifier la valeur de la variable de boucle en cours d'exécution, c'est que l'on veut modifier le nombre de tours de boucle. Dans ce cas, il faut employer un **Tant que Faire** (cf. explication à la fin de ce paragraphe) ;
- Les instructions s'exécutent tant que le nombre de répétition demandé n'est pas réalisé. La boucle **Pour** s'arrête lorsque la valeur de la variable de boucle dépasse la valeur de fin.

Exemple : Afficher les 10 premiers entiers en ordre décroissant.

```
Programme AfficheDixEntiers
Var
    Valeur : NOMBRE /* Valeur saisie au clavier */
Début
    Pour Valeur ← 10 à 1 par pas de -1
        Afficher Valeur
    Fin pour
Fin
```

Nous pouvons remarquer dans cet exemple que pour mettre en place une boucle **Pour** décroissante, la valeur d'initialisation doit être supérieure à la valeur de fin. De plus la valeur de **Pas** est négative.

Que se serait-il passé dans le cas suivant ?

```
Programme AfficheDixEntiers
Var
    Valeur : NOMBRE /* Valeur saisie au clavier */
Début
    Pour Valeur ← 1 à 10 par pas de -1
```

```
        Afficher Valeur
    Fin pour
Fin
```

Dans ce cas, la boucle **Pour** s'arrête dès le départ car on ne peut pas atteindre 10 en partant de 1 et en décrémentant la variable de boucle à chaque tour de boucle.

Donc, dans une boucle **Pour** qui décroît, la valeur de départ doit toujours être supérieure à la valeur de fin et le pas négatif.

Exemple : Affiche les « n » premiers entiers sachant que la valeur « n » sera saisie par l'utilisateur.

```
Programme  AffichePlusieursEntiers
Var
    I : NOMBRE /* Variable de boucle */
    ValeurLue : NOMBRE /* Variable saisie au clavier */
Début
    Afficher « Entrez la valeur limite »
    Saisir ValeurLue
    Pour i ← 1 à ValeurLue
        Affiche Valeur
    Fin Pour
Fin
```

A travers cet exemple, nous pouvons voir que la valeur de départ et la valeur d'arrivée de la boucle **Pour** peuvent être extraits du contenu d'une variable, voire d'une expression de calcul.

Attention, dans le cas où la borne de début ou de fin est un nom de variable ou une expression de calcul, celle-ci n'est évaluée qu'**une seule fois** avant le premier tour de boucle.

Exemple : Afficher un certain nombre d'entiers. Les valeurs de la borne de départ et de la borne de fin seront saisies au clavier.

```
Programme  AffichePlusieursEntiers
Var
    I : NOMBRE /* Variable de boucle */
    ValDep, ValFin : NOMBRE /* Variables saisies au clavier */
Début
    Afficher « Entrez la valeur de départ »
    Saisir ValDep
    Afficher « Entrez la valeur de fin »
    Saisir ValFin
    Pour i ← ValDep à ValFin
        Afficher Valeur
```

```
    Fin Pour
Fin
```

Exemple : Afficher un certain nombre d'entiers. Les valeurs de la borne de départ et de la borne de fin seront saisies au clavier. La valeur du « pas » sera déduite de la comparaison des deux bornes. Ainsi si la valeur de la borne de départ est inférieure ou égale à la valeur de la borne de fin, le « pas » sera de 1. Dans le cas contraire, il sera de –1.

```
Programme  AffichePlusieursEntiers
Var
    I : NOMBRE /* Variable de boucle */
    ValDep, ValFin : NOMBRE /* Variables saisies au clavier */
Début
    Afficher « Entrez la valeur de départ »
    Saisir ValDep
    Afficher « Entrez la valeur de fin »
    Saisir ValFin
    Si ValDep <= ValFin Alors
        LePas ← 1
    Sinon
        LePas ← -1
    Fin si
    Pour i ← ValDep à ValFin par pas de LePas
        Affiche Valeur
    Fin Pour
Fin
```

Une instruction **Pour** est en fait une réécriture simplifiée d'une instruction **Tant que**. C'est pourquoi, il est possible de réécrire une boucle **Pour** à l'aide d'un **Tant que**. Voyons cela à travers l'exemple suivant :

```
Pour i ← 1 à 10
    Afficher i*10
Fin Pour
```

Rappel : La valeur initiale de la variable de boucle est donnée avant le premier tour de boucle. La condition de fin est vérifiée avant chaque tour de boucle. La valeur de la variable de boucle s'incrémente ou se décrémente automatiquement à la fin de chaque tour de boucle.

Ré-écrivons les lignes précédentes à l'aide d'une instruction **Tant que** :

```
i ← 1
Tant que i <= 10 Faire
```

```
    Afficher i*10
    i ← i+1
Fin tant que
```

Il faut donc placer une initialisation de la variable de boucle avant le **Tant que**, ajouter une condition pour vérifier que l'on n'a pas atteint la valeur de fin et incrémenter la variable de boucle à la fin des instructions.

```
Pour i ← 100 à 0 par pas de –2
    Afficher i*10
Fin Pour
```

Sera ré-écrit de la manière suivante :

```
i ← 100
Tant que >= 0 Faire
    Afficher i*10
    i ← i-2
Fin tant que
```

Quant faut-il employer un « Pour » à la place d'un « Tant que » et vice-versa ?

On emploie un **Tant que** lorsque l'on ne connaît pas d'avance le nombre de tours de boucles à effectuer ou que l'on souhaite pouvoir quitter la boucle n'importe quand. Au contraire, on emploie un **Pour** lorsque l'on connaît d'avance le nombre de tours de boucles à effectuer. Attention, il n'est pas possible de quitter une boucle **Pour** avant son terme, sinon c'est qu'il fallait employer un **Tant que**.

8.6 Structure de choix : Selon

```
Selon ExpressionDeTest Faire
    Cas Expression : Instructions
    Cas Expression : Instructions
    Cas Expression : Instructions
    ……
    ……
    Cas Sinon : Instructions
Fin Selon
```

Une instruction de structure de choix évite des successions de « Si ».

Exemple : Afficher une appréciation sur un élève en fonction de sa moyenne.

```
Programme AfficheAppréciation
Var
    LaMoyenne : NOMBRE /* Variable saisie au clavier */
Début
    Afficher « Entrez la moyenne de l'élève »
    Saisir LaMoyenne
    Selon LaMoyenne Faire
        Cas >=0 Et <10 : Afficher « Mention refusée »
        Cas >=10 Et <12 : Afficher « Mention Passable »
    Cas >=12 Et <14 : Afficher « Mention Assez Bien »
    Cas >=14 : Afficher « Mention Très Bien »
    Cas Sinon : Afficher « Erreur de moyenne »
    Fin Selon
Fin
```

Remarque :

Certains langages de programmation n'acceptent que des valeurs constantes et non des expressions logiques pour mettre en place les différents « Cas » dans une instruction de « Structure de choix », comme dans notre exemple. Le langage C fait partie de cette catégorie.

9. Exercices d'applications

9.1 PPCM

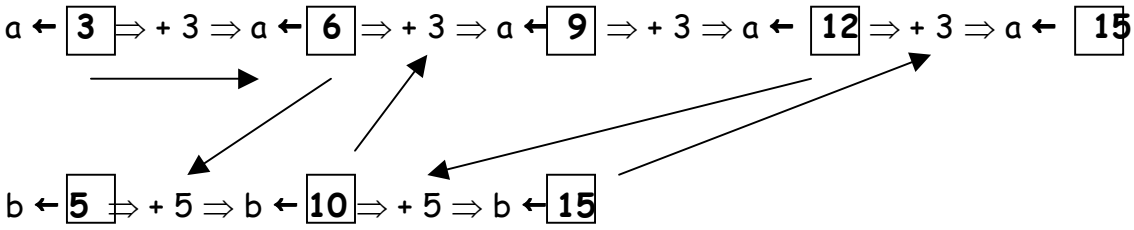
Enoncé : Ecrire un programme qui calcule le Plus Petit Commun Multiple entre une valeur « a » et une valeur « b » saisies au clavier.

Exemple :

a = 2 et b = 3 $\Rightarrow$ ppcm = 6
a = 3 et b = 12 $\Rightarrow$ ppcm = 12

Principe : Cet algorithme consiste à faire converger les valeurs « a » et « b » vers une valeur commune en augmentant la valeur « a » si elle est plus petite que la valeur « b » ou la valeur « b » si elle est plus petite que la valeur « a ».

Prenons un exemple : l'utilisateur saisit 3 et 5.



Le PPCM de **3** et **5** est **15.**

9.2 Température

Enoncé : On sait que : l'eau gèle à 0°; l'eau salée gèle à – 3° ; le fuel gèle à – 5° ; l'ordinaire gèle à – 13° ; le super gèle – 23°.

Ecrire un programme qui demande une température et sort la liste des liquides qui gèlent à cette température. Dans un deuxième temps, modifier le programme en utilisant des constantes nommées.

Exemple :

L'utilisateur saisit : -7
Le programme affiche : Eau gèle
Eau salée gèle
Fuel gèle

9.3 Afficher la plus grande valeur

Enoncé : Saisir trois nombres entiers quelconques a, b, c. L'algorithme doit afficher la plus grande des 3 valeurs saisies.

9.4 Afficher la somme de N valeurs

Enoncé : L'utilisateur saisit un nombre « N » au clavier. L'algorithme effectue la somme des nombres compris entre 1 et N puis affiche le résultat à l'écran.

9.5 Factorielle

Enoncé : Saisir un nombre au clavier et affiche la factorielle de la valeur.

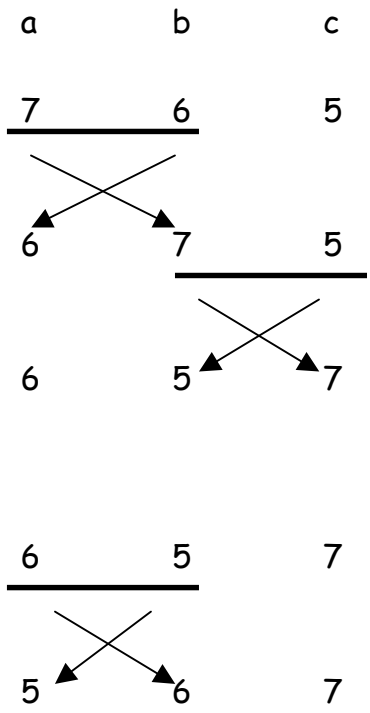
Exemple : Factorielle (4), notée 4 !, est égale à 4x3x2x1.

9.6 Tri de 3 valeurs

Enoncé : Saisir trois nombres entiers quelconques a, b, c. L'algorithme trie physiquement les trois nombres puis affiche les valeurs de a, b, c(les données apparaîtront alors dans l'ordre croissant).

Principe : Nous allons utiliser le principe du tri à bulle pour trier les trois valeurs. Ce tri consiste à prendre une liste de nombres, ouis en comparant les valeurs de la liste 2 par 2, à effectuer des permutations éventuelles de manière à amener la valeur la plus grande de la liste en fin de liste. On recommence avec la liste moins le dernier élément car il est déjà en place et ce jusqu'à ce que la liste à trier ne se compose plus que d'un élément.

Exemple : L'utilisateur saisit la valeur **7** dans la variable **a**, **6** dans la variable **b** et **5** dans la variable **c**. Attention, l'algorithme doit fonctionner sur n'importe quelle valeur.



1 re comparaison

2 e comparaison

La plus grande valeur, pour la série considérée, est en place. On recommence sur la même liste moins le dernier élément.

1 re comparaison avec la nouvelle série

La liste est triée

9.7 Carré d'un nombre par la méthode des impairs

Enoncé : Calculer le carré d'un nombre saisi au clavier par la méthode des impaires.

Principe : Cette méthode consiste à additionner les « n » premiers impairs pour trouver le carré de la valeur « n ».

Exemple :

$2^2 = 1 + 3 = 4$ */* On additionne les deux premiers impairs */*
$3^2 = 1 + 3 + 5 = 9$ */* On additionne les trois premiers impairs */*
$4^2 = 1 + 3 + 5 + 7 = 16$ */* On additionne les quatre premiers impairs */*
$5^2 = 1 + 3 + 5 + 7 + 9 = 25$ */* On additionne les cinq premiers impairs */*

9.8 Compter le nombre de valeurs paires et impaires

Enoncé : On demande à l'utilisateur de saisir une série de nombres entiers positifs. La saisie s'arrête lorsque l'utilisateur tape la valeur 0 (zéro). Lorsque la saisie est terminée, le programme affiche le nombre de valeurs paires et le nombre de valeurs impaires saisies.

Principe : Pour savoir si une valeur est paire, il suffit de regarder si le reste de la division de la valeur considérée par 2 est égal à zéro. Il suffit d'employer l'opérateur **mod** (cf. paragraphe 6.1 dans ce cours).

9.9 Afficher la plus grande et la plus petite valeur saisie

Enoncé : On demande à l'utilisateur de saisir une série de nombres entiers ou négatifs. La saisie s'arrête lorsque l'utilisateur tape la valeur 0 (zéro). Lorsque la saisie est terminée, le programme affiche la valeur la plus grande et la valeur la plus petite des valeurs saisies. Dans cette exercice, on demande pas de mémoriser les valeurs saisies mais la plus grande et la plus petite.

9.10 Afficher la différence entre deux heures

Enoncé : On demande à l'utilisateur de saisir deux heures en donnant à chaque fois les heures, les minutes, les secondes. On considère que les heures saisies font partie de la même journée. L'algorithme affiche la différence entre les deux heures.

Exemple : Si l'utilisateur saisit :

14 30 20
14 51 10

La différence est de 0h 20m 50s.

Principe : On traduit en seconde les deux saisies puis on calcule la différence entre les deux valeurs obtenues. Cela nous donne le nombre de secondes qui séparent les deux heures. Dans un deuxième temps, on traduit la différence en nombre d'heures, de minutes et de secondes.

9.11 Mini calculette

Enoncé : Mettre en place une mini calculette qui ne réalise que 4 opérations : +,-,*,/.
L'utilisateur entre deux valeurs et un opérateur, le programme affiche le résultat. Nous considérons que l'utilisateur saisit les valeurs correctement, donc il n'y a pas de gestion d'erreur.