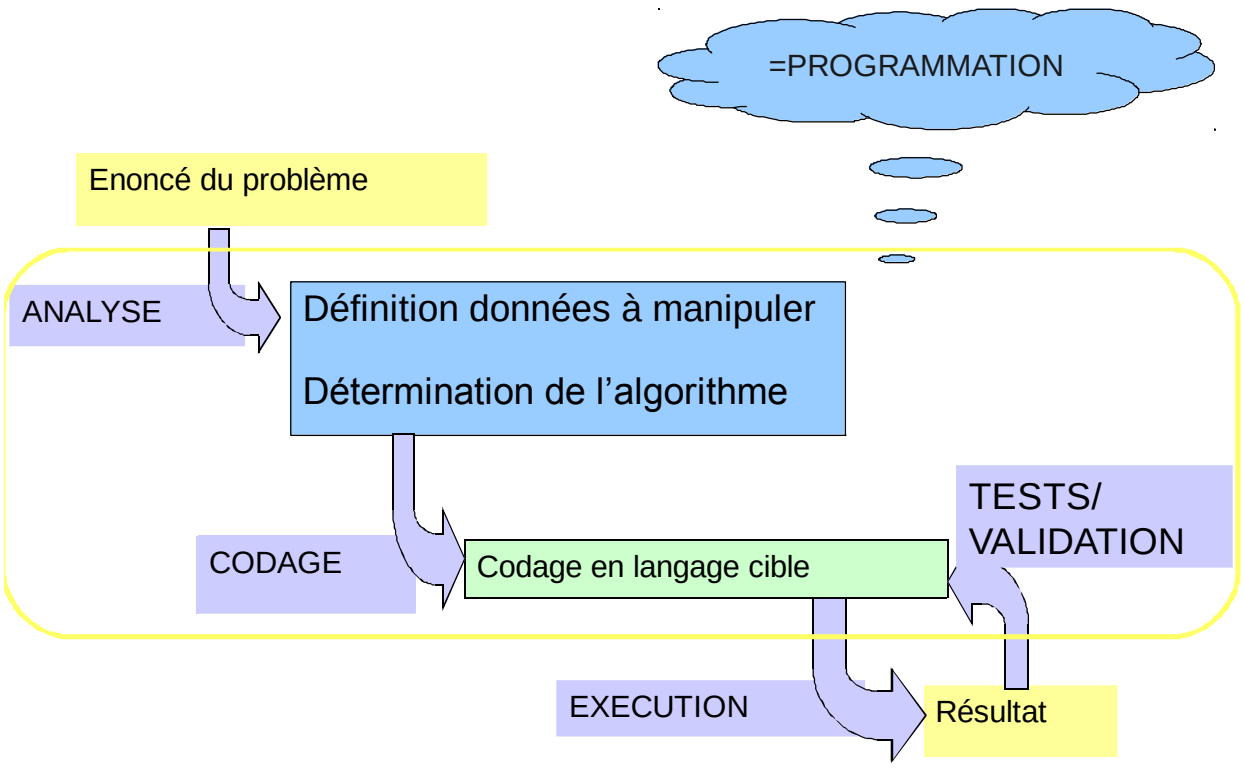
Algorithmique et programmation

1 Introduction à la programmation

La programmation permet d'établir une séquence d'actions pouvant être exécutées par le processeur pour réaliser un travail donné.

La programmation comporte 3 parties :

la résolution du problème passe par :
1. La détermination des objets manipulés et rédaction d'un algorithme pour le traitement à réaliser sur ces objets.
2. L'adaptation de l'algorithme au processeur : codage de l'algorithme dans un langage donné. Compilation éventuelle pour que le programme soit compréhensible par le processeur.
3. L'exécution.



.1.1 Modes de codage : langages de programmation

De très nombreux langages de programmation existent. Le choix du langage se fait en fonction des connaissances du programmeur, des habitudes de l'entreprise et de la bonne adaptation du langage au problème à résoudre.

.1.2 Modes de programmation

Il existe plusieurs modes de programmation :
- procédural ou séquentiel,
- par objet,
- événementiel.

.1.3 Modes d'exécution

Le programme (code) établi peut ensuite être :
- interprété : PHP, scripts Bash, perl, Tcl,...
- compilé puis interprété : Java. La compilation du code produit un "bytecode Java" qui est ensuite exécuté par une machine virtuelle Java.
- compilé : C, Pascal, ...

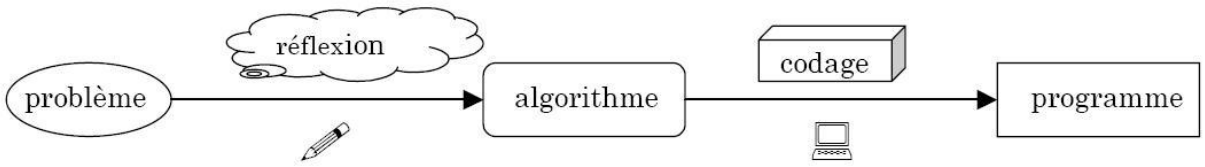
Quel que soit le mode de programmation choisi, la rédaction d'un algorithme est primordiale.

2 Introduction à l'algorithmique

.2.1 Qu'est-ce ?

Un algorithme décrit l'ensemble des opérations nécessaires permettant de résoudre le problème posé.

L'algorithmique, l'art d'écrire des algorithmes, permet de se focaliser sur la procédure de résolution du problème sans avoir à se soucier des spécificités d'un langage de programmation particulier.



.2.2 Pourquoi apprendre l'algorithmique pour apprendre à programmer ?

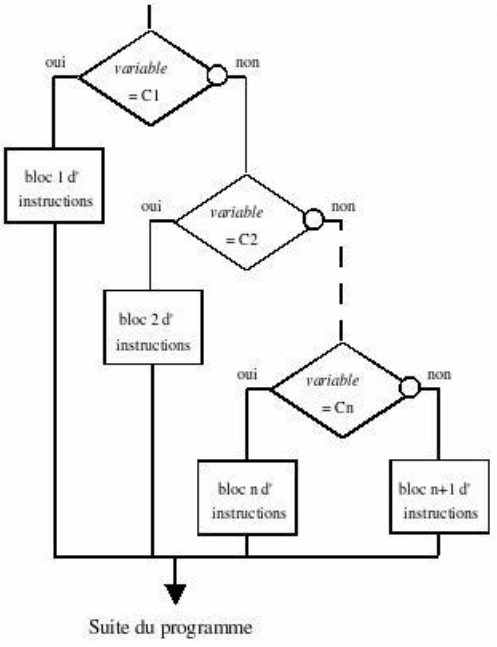
En quoi a-t-on besoin d'un langage spécial, distinct des langages de programmation compréhensibles par les ordinateurs ?

L'algorithmique exprime les instructions résolvant un problème donné indépendamment des particularités de tel ou tel langage.

Apprendre l'algorithmique, c'est apprendre à manier la structure logique d'un programme informatique. Cette dimension est présente quelle que soit le langage de programmation.
Quand on programme dans un langage , on doit en plus gérer les problèmes de syntaxe, ou de types d'instructions, propres à ce langage.
Apprendre l'algorithmique de manière séparée, c'est donc dissocier les difficultés pour mieux les vaincre.

.2.3 Aspect d'un algorithme

Représentation sous la forme d'un organigramme :



Aujourd'hui, cette représentation est quasiment abandonnée, pour 2 raisons :

1. Dès que l'algorithme commence à grossir un peu, ce n'est plus pratique du tout du tout.
2. Cette représentation favorise le glissement vers un certain type de programmation, dite non structurée que l'on tente d'éviter.

Représentation en pseudo-code :

```
VAR
   a, b, c : Entiers
DEBUT
     a ← 5
     b ← 3
     c ← a + b
     a ← 7
     c ← b – a
FIN
```

Une série de conventions appelée « pseudo-code » est utilisée pour écrire un algorithme.
Il ressemble à un langage de programmation authentique dont on aurait éliminé la plupart des problèmes de syntaxe.
Ce pseudo-code est susceptible de varier légèrement d'un ouvrage à l'autre.
Le pseudo-code est purement conventionnel : aucune machine n'est censée le reconnaître tel quel.

.2.4 Contenu d'un algorithme

Les ordinateurs, quels qu'ils soient, ne sont fondamentalement capables de comprendre que quatre catégories d'ordres (en programmation, on emploiera plutôt le terme d'instructions).

Ces 4 familles d'instructions sont :

1. l'affectation de variables ;
2. la lecture (saisie) / écriture (affichage) ;
3. les tests ;
4. les boucles.

Un algorithme informatique se ramène donc toujours au final à la combinaison de 4 briques de base.
Il peut y en avoir quelques unes, quelques dizaines, et jusqu'à plusieurs centaines de milliers dans certains programmes.

3 Les variables

.3.1 Généralités

Les programmes ont pour but de traiter différentes données afin de produire des résultats.
Les résultats peuvent eux-mêmes être des données pour d'autres programmes.

Les données d'un programme doivent être récupérées en mémoire centrale, à partir du clavier ou d'un fichier par exemple, pour pouvoir être traitées par le processeur qui exécute le programme. Ainsi, toutes les données d'un programme sont mémorisées en mémoire centrale, dans des sortes de cases que l'on appelle variables.
Une variable peut être représentée par une case mémoire, qui contient la valeur d'une donnée.
Chaque variable possède un nom unique appelé identificateur par lequel on peut accéder à son contenu.

Exemple :
une variable **prix** de valeur **15.3** (identificateur = prix ; valeur = 15,3)
une variable **nom** de valeur **Dupond** (identificateur = nom ; valeur = Dupond)

Attention : ne pas confondre la variable et son contenu
Une variable est un contenant, c'est à dire une sorte de boîte, alors que le contenu d'une variable est une valeur numérique, alphanumérique ou booléenne, ou de tout autre type.

Deux variables peuvent avoir la même valeur, mais une variable ne peut pas avoir plusieurs valeurs en même temps.

On rencontre deux sortes de variables :
- celles dont la valeur est … variable. Ce sont les "vraies" variables ;
- celles dont la valeur ne change pas et que l'on appellera constantes.

Exemple : **pi** vaut **3.14159** de manière permanente.

.3.2 Déclaration

Pour qu'un programme puisse utiliser une variable, il faut au préalable que cette variable ait été déclarée, c'est-à-dire que le programme lui ait réservé une place en mémoire et ait attribué l'identificateur à cette place.

Toutes les variables n'ont pas besoin de la même place en mémoire. Un grand nombre prend plus de place qu'un caractère. Selon le type de l'objet, il faudra lui réserver plus ou moins de place.
=> il faut déclarer le type des variables et pas seulement leur nom.

=> la déclaration d'une variable indique 2 éléments :
- son identificateur (son nom)
- son type
-

Syntaxe : **identificateur : type**

Exemples :
prix : réel
quant : entier

Un identificateur peut être composé de lettres et de chiffres mais il ne peut pas commencer par un chiffre et ne peut comporter d'espaces.

L'identificateur des variables doit permettre de reconnaître aisément la donnée qu'il contient. Par exemple : pour désigner un nom, l'identificateur **nom** est plus parlant que **a** même si celui-ci est plus rapide à écrire !

.3.3 Types de variables

En algorithmique, on distingue 5 types principaux :

1. les caractères (lettres, chiffres, ponctuation, code des opérations, espace, retour chariot,…)
   Exemples : 'a', 'C', '-',...

2. les chaînes de caractères (ensemble de caractères)
   Exemples : 'bonjour!', 'nom et prenom?', …

3. les entiers (les nombres sans virgule)
   Exemples : 10, -1, 1567, …

4. les réels (les nombres avec virgule et sans virgule)
   Exemples : 203.7 , -4521.9 , 7.09, 7.9, …

5. les booléens (qui n'ont que deux valeurs possibles: soit VRAI, soit FAUX , ou 1 et 0)

.3.4 Affectation de variables

L'affectation consiste tout simplement à placer une valeur dans une variable (ce qui revient à changer le contenu de cette variable).

En pseudo-code, l'instruction d'affectation se note avec le signe ←

Syntaxe : identificateur ← valeur

On évite d'utiliser le symbole =, qui est pourtant fréquemment utilisé dans les divers langages de programmation. (car on peut être amené à écrire x = x+1, ce qui n'a pas de sens en maths!).

La nouvelle valeur est évaluée à partir d'une expression, qui peut être :
- soit une autre variable ou constante,

- soit une valeur,
- soit une combinaison de variables, de valeurs littérales et d'opérateurs.

Exemples :

x ← 5
x ← y
x ← 5 + 8

Remarque : l'expression située à droite de la flèche doit être du même type que la variable située à gauche. Si x est du type caractère, alors :
x ← 5 provoquera une erreur mais pas x ← '5'

.3.5 Opérateurs

Il est possible de réaliser certaines opérations entre les variables. La nature des opérations dépend du type des variables :

Entiers, réels :

opérations	symboles
Arithmétique élémentaire	+, -, *, /
Division entière (euclidienne)	DIV. Le reste est obtenu grâce à l'opérateur MOD (modulo)
exposant	^
comparaisons	<, >, <=, >=, =, <>

Booléens :

opérations	symboles
Et logique	et
Ou logique	ou
Non logique	non
comparaisons	<, >, <=, >=, =, <>

Caractères, chaînes :

opérations	symboles
Concaténation	&
comparaisons	<, >, <=, >=, =, <>

.3.6 Premier exemple d'algorithme

```
PROGRAMME monPremierProgramme
/* les constantes: obligatoire de
leur affecter une valeur dès leur déclaration */
CONST
   taux <-  10 : entier
   phrase <- "bonjour!" : chaîne
// Les variables au sens strict
VAR
   var1, var2, res : réels
   mot : chaîne
DEBUT

/* instructions*/

FIN
```

Un algorithme commence par le mot PROGRAMME suivi de son identificateur (le nom du programme).

Ensuite viennent les déclarations :
- des constantes, annoncées par CONST,
- puis des variables, annoncées par VAR.

Pour déclarer une variable, on indique son identificateur suivi d'un double point et de son type. La valeur des constantes est donnée dès leur déclaration, avec le signe <- précédé de son identificateur.

Le corps du programme commence par DEBUT et se termine par FIN.

Les variables doivent être initialisées en début de programme ou au moins avant leur utilisation.

On peut insérer des commentaires :
- soit entre les balises **/* */** ;
- soit après **//** jusqu'à la fin de la ligne.

4 Lecture et écriture

.4.1 Lecture / saisie

L'instruction de saisie/lecture permet de communiquer des données au programme.

Cette instruction assigne une valeur entrée au clavier dans une variable.
Tant que l'utilisateur n'entre rien au clavier, le déroulement du programme est stoppé.

Syntaxe : **SAISIR var1 [, var2, …] ou LIRE var1 [, var2, …]**

Exemples :

SAISIR x

Cette instruction lit la valeur saisie au clavier et l'affecte à la variable x.

SAISIR x ,y

Cette instruction lit la première valeur saisie au clavier et l'affecte à x, puis lit la deuxième valeur saisie et l'affecte à y.

l'instruction de saisie (ou de lecture sur un périphérique autre que le clavier) est indispensable pour permettre d'utiliser le même programme avec des données différentes sans avoir à changer les valeurs du programme à chaque fois.

.4.2 Écriture / affichage

La plupart des programmes nécessite de communiquer à l'utilisateur un certain nombre de résultats par l'intermédiaire d'un périphérique. Ils utilisent des instructions d'affichage.

L'instruction d'affichage permet de fournir des résultats sous forme directement compréhensible pour l'utilisateur à travers l'écran.

Syntaxe : **AFFICHER expression1 [expression2, …]** ou **ECRIRE expression1 [expression2, …]**

Exemples :

AFFICHER toto => Affichage de la valeur de la variable toto à l'écran
AFFICHER "Bonjour !" => Affichage de la chaîne de caractères *Bonjour!* à l'écran
AFFICHER a b

Quand on veut afficher deux objets à la suite, on les sépare par un espace
Si a vaut 5 et b vaut 10, on obtient alors à l'écran: 5 10

On peut mélanger l'affichage de valeurs littérales et de variables.
Cela est particulièrement utile si on veut voir apparaître un libellé (texte accompagnant la saisie des données ou l'édition des résultats, permettant de guider l'utilisateur).

Exemple : AFFICHER "Voici les résultats : x = " x " et y = " y

5 Structures de contrôle conditionnelles

.5.1 Introduction

L'ordre des instructions est primordial en programmation.
Le processeur exécute les instructions dans l'ordre dans lequel elles apparaissent dans le programme : l'exécution est séquentielle.
Une fois que le programme a fini une instruction, il passe à la suivante.
Tant qu'une instruction n'est pas terminée, il attend avant de continuer (par exemple, une instruction de saisie attend que l'utilisateur saisisse une valeur au clavier avant de continuer).

Parfois, il est nécessaire qu'un programme n'exécute pas toutes les instructions, ou encore qu'il recommence plusieurs fois les mêmes instructions.
Pour cela, il faudra casser la séquence : c'est le rôle des structures de contrôle.

Les structures de contrôle conditionnelles permettent de n'exécuter certaines instructions que sous certaines conditions.
Les structures conditionnelles permettent d'exécuter des instructions différentes en fonction de conditions.

Une condition (ou expression conditionnelle ou expression logique) est évaluée: elle est jugée vrai ou fausse :
- Si elle est vraie, un traitement (une ou plusieurs instructions) est réalisé.
- Si la condition est fausse, une autre traitement peut éventuellement être exécuté.

Ensuite le programme continue normalement.

Il existe 2 types principaux de structures conditionnelles :
- les structures conditionnelles au sens strict (SI…ALORS)
- les structures alternatives (SI…ALORS…SINON)

.5.2 La structure conditionnelle SI…ALORS

Cette structure est utilisée si on veut exécuter une instruction seulement si une condition est vraie et ne rien faire si la condition est fausse.

```
Syntaxe :   SI <condition>
            ALORS
                    instruction
                    ...
            FINSI
```

.5.3 La structure alternative SI…ALORS... SINON

Dans le déroulement d'un algorithme, on doit souvent choisir entre 2 actions, suivant une condition. On utilise la structure alternative.

```
Syntaxe :   SI <condition>
            ALORS
                    instruction
                    …
            SINON
                    instruction
                    …
            FINSI
```

Exemple : écrire un message précisant si la valeur d'une variable n saisie par l'utilisateur est positive ou négative.

```
DEBUT
        AFFICHER "Entrez un nombre"
        SAISIR n
        SI n > 0
        ALORS    //Cas où l'expression n>0 est vraie
                AFFICHER "La valeur ",  n  , " est positive"
        SINON    //Cas où l'expression n>0 est fausse
                AFFICHER "La valeur ",  n  , " négative ou nulle"
        FINSI
FIN
```

.5.4 Expression des conditions

Une expression conditionnelle (ou expression logique, ou expression booléenne) est une expression dont la valeur est soit VRAI soit FAUX, c'est à dire finalement un booléen.

.5.5 Conditions simples

Une condition simple est une **comparaison de deux expressions de même type**.

Exemples :
a < 0 : type entier ou réel ;
op = 's' : type caractère

Une condition est une comparaison. Elle est composée de trois éléments :
- une valeur
- un opérateur de comparaison
- une autre valeur

Syntaxe : valeur1 *symbole_de_comparaison* valeur2
(Symboles de comparaison en algorithmique : **< , > , = , ≤ , ≥ , <>)**

L'ensemble des 3 éléments constituant la condition constitue une affirmation, qui a un moment donné est VRAIE ou FAUSSE.

Pour les comparaisons de caractères, l'ordre de la table de caractères ASCII permet de comparer. Une lettre placée avant une autre dans l'ordre alphabétique sera inférieure à l'autre.

Exemple :
's' est supérieur à 'm'.
'A' est inférieur à 'a'

Remarque : une condition simple ne signifie pas condition courte.
Exemple : (a + b - 2) * 2*c ≤ (5 * y –2) / 3

.5.6 Conditions composées

Les conditions (ou expressions conditionnelles) peuvent aussi être complexes, c'est à formées de plusieurs conditions simples ou variables booléennes reliées entre elles par les opérateurs logiques ET, OU, OU EXCLUSIF et NON.

Exemple 1 :

SI a < 0 ET b > 0
  ALORS…

Exemple 2 :

SI (a +3 = b ET c < 0) OU (a = c *2 ET b <> c)
  ALORS …

1. Opérateur logique ET

Le ET a le même sens en informatique que dans le langage courant.
Condition1 ET Condition2 est VRAI, si Condition1 est VRAI et Condition2 est VRAI.
Dans tous les autres cas, "Condition 1 ET Condition2" sera faux.

2. Opérateur logique OU

"Condition1 OU Condition2" est VRAI, si Condition1 est VRAI ou si Condition2 est VRAI.

Point important : si Condition1 est VRAIE et Condition2 est VRAIE aussi, Condition1 OU Condition2 reste VRAIE.
Le OU informatique ne signifie pas « ou bien »

3. Opérateur logique OU exclusif : XOR

Condition1 XOR Condition2 est VRAI si soit Condition1 est VRAI, soit Condition2 est VRAI.

4. Opérateur logique NON

Le NON inverse une condition :
NON(Condition1)est VRAI si Condition1 est FAUX, et il sera FAUX si Condition1 est VRAI.

.5.7 Tests imbriqués

Il est souvent nécessaire d'effectuer des tests qui donnent lieux à plus de 2 cas. Dans ce cas, on peut imbriquer les structures conditionnelles.

Syntaxe :

```
SI <condition>
ALORS
        instruction
        …
SINON
        SI <condition>   //test imbriqué
        ALORS
                instruction
                …
        SINON
                instruction
                …
        FINSI
FINSI
```

Cette imbrication limite le nombre de tests effectués par rapport à une suite de tests non imbriqués : le code sera au final plus rapide.

.5.8 Tests imbriqués de type SI ALORS SINON SI

L'imbrication des tests peut devenir compliquée. On peut alors utiliser une forme simplifiée

```
Syntaxe :   SI <condition>
            ALORS
                    instruction
                    …
            SINON SI <condition>
            ALORS
                    instruction
                    …
            SINON
                    instruction
                    …
            FINSI
```

Dans cette forme plus compacte, l'indentation est réduite mais le code reste bien lisible.
Une seule instruction FINSI permet de fermer l'ensemble des tests.

.5.9 Structure choix multiples SELON (ou SUIVANT) CAS... FAIRE

La structure SELON permet de choisir le traitement à effectuer en fonction de la valeur d'une variable ou d'une expression. Cette structure permet de remplacer avantageusement une succession de structures SI…ALORS.

```
Syntaxe :   SELON CAS expression FAIRE
                    valeur1_de_l_expression : traitement 1
                    valeur2_de_l_expression : traitement 2
                    valeur3_de_l_expression : traitement 3
                    …
                    DEFAUT : traitement par défaut   //gestion éventuelle des autres cas
            FINSELON
```

L'expression peut être :

- une variable

Exemple :

```
SELON CAS num FAIRE
        1 : écrire "Insuffisant"
        2 : écrire "Faible"
        3 : écrire "moyen"
        DEFAUT : écrire "Convenable"
FINSELON
```

- une expression évaluée

Exemple :

```
SELON CAS (i*3)-1 FAIRE
        1 : écrire "Insuffisant"…
```

6 Structures de contrôle itératives (boucles)

.6.1 Introduction

Les structures répétitives ou structures itératives boucles permettent de répéter un traitement (c'est à dire une instruction simple ou composée) autant de fois que nécessaire :

- soit un nombre déterminé de fois,
- soit tant qu'une condition est vraie.

3 grands types principaux de structures répétitives:

- la structure TANT QUE … FAIRE : permet d'effectuer une instruction tant qu'une condition est satisfaite ;
- La structure POUR : permet de répéter une instruction un certain nombre de fois ;
- La structure REPETER…JUSQU'A : permet de répéter une instruction jusqu'à ce qu'une condition soit satisfaite.

Seule la boucle TANT QUE est fondamentale : avec cette boucle, on peut réaliser toutes les autres boucles. L'inverse n'est pas vrai.

La boucle POUR est aussi très utilisée :
elle permet de simplifier la boucle TANT QUE lorsque le nombre de répétitions de boucle est connu d'avance.

La boucle REPETER est beaucoup moins utilisée.

.6.2 La structure TANT QUE … FAIRE

La boucle TANT QUE … FAIRE permet de répéter un traitement tant qu'une expression conditionnelle est vraie. Si d'emblée, la condition n'est pas vraie, le traitement ne sera pas exécuté.

Remarque :
La boucle TANT QUE a un point commun avec la structure conditionnelle : si la condition n'est pas vraie, le traitement n'est pas exécuté. Ne pas les confondre cependant !

```
Syntaxe :   TANT QUE condition d'exécution (critère de continuation) FAIRE
                traitement      // instruction simple ou bloc d'instructions
            FINTANTQUE
```

Exemple :

```
PROGRAMME Somme
VAR
    somme, i : entiers
DEBUT
    somme ← 0
    i ← 0
    TANT QUE i <= 100
        somme ← somme + i
        i ← i+1
    FINTANTQUE
FIN
```

.6.3 La structure POUR

La boucle POUR permet de répéter une instruction un nombre connu de fois.
Elle permet de faire la même chose que la boucle TANT QUE mais de manière plus rapide, lorsque le nombre de répétitions est connu.

```
Syntaxe :   POUR compteur DE valeur initiale A valeur finale [PAR PAS DE incrément] FAIRE
                traitement
            FINPOUR
```

La variable compteur est de type entier. Elle est initialisée à la valeur initiale.
Le compteur augmente (implicitement) de l'incrément à chaque répétition du traitement.
Lorsque la variable compteur vaut la valeur finale, le traitement est exécuté une dernière fois puis le programme sort de la boucle.

Exemple :

```
PROGRAMME MultiplicationPar7
VAR
  nb : Entier
DEBUT
  POUR nb de 1 à 10 PAR PAS DE 1 FAIRE
        AFFICHER nb, " * 7 = ", nb * 7
  FINPOUR
FIN
```

Par défaut, l'incrément est de 1. Mais l'incrément peut valoir 2,...
Il peut être négatif. Dans ce cas, la *valeur finale* doit être inférieure à la *valeur initiale*.

Exemple :

```
PROGRAMME compte_a_rebours
VAR
  nb : Entier
DEBUT
  POUR nb de 5 à 0 PAR PAS DE -1 FAIRE
        AFFICHER nb, " secondes"
  FINPOUR
FIN
```

.6.4 La structure REPETER … JUSQU'A

Cette boucle sert à répéter une instruction jusqu'à ce qu'une condition soit vraie.

Utilisation

```
Syntaxe :   REPETER
                traitement              // une instruction simple ou un bloc d'instructions
            JUSQU'A condition d'arrêt
```

Le traitement est exécuté, **puis la condition est vérifiée**.

Si elle n'est pas vraie, on retourne au début de la boucle et le traitement est répété.
Si la condition est vraie, on sort de la boucle et le programme continue séquentiellement.
A chaque fois que le traitement est exécuté, la condition d'arrêt est de nouveau vérifiée à la fin.

Exemple :

```
PROGRAMME Aire
VAR
        rayon : réel
        réponse : caractère
DEBUT
        AFFICHER "Calcul de l'aire d'un disque"
        REPETER
                AFFICHER  "Entrez le rayon d'un cercle en cm"
                SAISIR  rayon
                AFFICHER "L'aire du disque est ", rayon^2 *3.14, "cm²"
                AFFICHER "Voulez-vous l'aire d'un autre disque? o/n)"
                SAISIR réponse
        JUSQU'A réponse <> "o"   // pour toute autre réponse que "o", stop
        AFFICHER "Le programme est terminé! "
FIN
```

Différences avec la structure TANT QUE

La différence entre les deux structures est que dans la boucle REPETER … JUSQU'A, le traitement est effectué au moins une fois.
Dans la structure TANT QUE, si la condition n'est pas vraie, le traitement n'est jamais exécuté.

La structure REPETER … JUSQU'A n'est pas implémentée dans tous les langages et peut être remplacée par la structure TANT QUE … FAIRE

Exemple précédent avec TANT QUE :

```
        ...
        TANT QUE reponse = "o"
                AFFICHER  "Entrez le rayon d'un cercle en cm"
                SAISIR rayon
                AFFICHER "L'aire du disque est ", rayon^2 *3.14, "cm²"
                AFFICHER "Voulez-vous l'aire d'un autre disque? (o/n)"
                SAISIR réponse
        FINTANTQUE
        AFFICHER  "Le programme est terminé! "
FIN
```

7 Les sous-programmes

.7.1 Présentation

Utilité

Lorsqu'un programme est long, il est irréaliste d'écrire son code d'un seul tenant. On décompose le programme en plusieurs parties plus petites qu'on assemble pour former l'application finale.

Les sous-programmes permettent donc :

- De découper un gros programme en éléments plus petits et donc plus simples à écrire et à comprendre ;
- D'éviter de répéter plusieurs morceaux de code identiques ;
- De travailler à plusieurs sur un même programme.

Types de sous-programmes

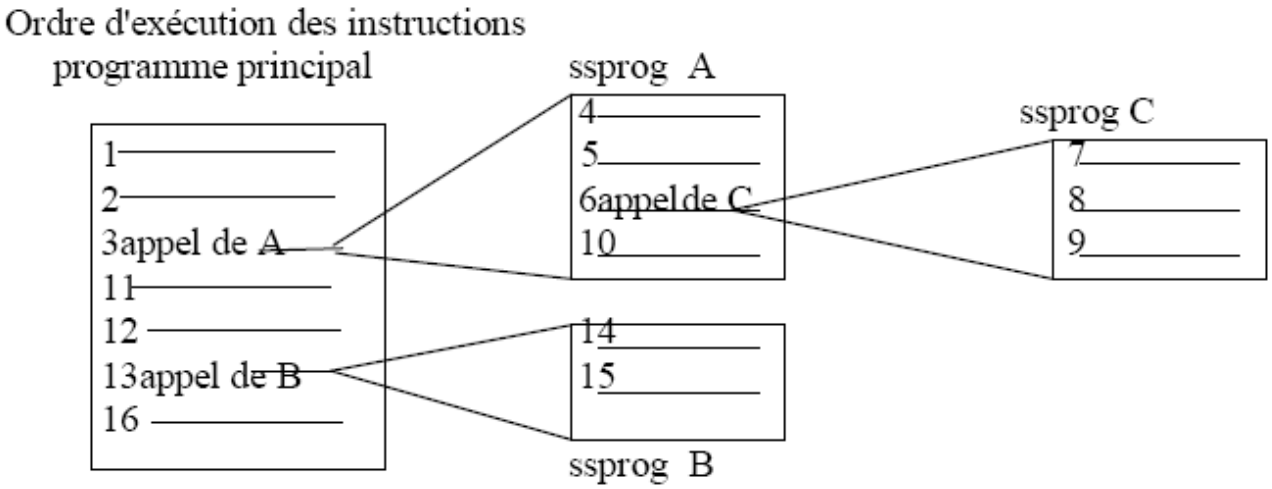
Deux types existent :

- Les procédures. Elles ne représentent qu'une suite d'instructions ;
- Les fonctions. Elles renvoient en plus une valeur résultat, tout comme une fonction en maths.

Appel d'un sous-programme

Contrairement à un programme, un sous-programme ne peut pas s'exécuter indépendamment d'un autre programme.
Seul le programme principal s'exécute au lancement de l'application, puis il peut ensuite demander l'exécution d'un ou plusieurs sous-programmes grâce à une instruction spécifique : **l'appel**.
Un sous-programme peut être appelé :

- Par le programme principal ;
- Par un autre sous-programme ;
- Par lui-même (appel récursif).



Lorsque l'exécution d'un sous-programme est terminée, il y a un RETOUR à l'instruction qui suit l'appel dans le programme appelant.

Attention : si un sous-programme écrit n'est jamais appelé, il ne sera jamais exécuté.

Un sous-programme est appelé simplement en invoquant son nom depuis le programme appelant.

.7.2 Les procédures

Une procédure est un sous-programme qui remplace une série d'instructions.

Tout comme un programme, une procédure doit être définie et possède :

- Un nom ;
- Des variables ;
- Des instructions ;
- Un début et une fin.

```
Syntaxe :
PROCEDURE nomProcedure(...)
VAR locales
        ...
DEBUT
        // Bloc d'instructions
FIN
```

Exemple :

```
Procédure DessinerLigne()
VAR i :entier
DEBUT
        POUR i DE 1 A 10 FAIRE
                ECRIRE "*"
        FINPOUR
FIN

//Programme appelant la procédure
PROGRAMME principal
VAR n : entier
DEBUT
        ECRIRE "Voici une belle figure géométrique"
        POUR n DE 1 A 5 FAIRE
                DessinerLigne()
        FINPOUR
FIN
```

.7.3 Les fonctions

Les fonctions réalisent aussi une suite d'instructions mais en plus elles renvoient une valeur résultat au programme appelant grâce à l'instruction RETOURNER.

Le type de la valeur retournée doit être indiqué dans la définition de la fonction.

Syntaxe :
```
FONCTION nomFonction ( … ) : typeValeur de retour
DEBUT
        //Bloc d'instructions
        RETOURNER …
FIN
```

La valeur résultat de l'appel d'une fonction **doit être utilisée** par le programme appelant.
En effet, l'appel d'une fonction est remplacé dans le programme appelant par la valeur renvoyée.
On ne peut pas trouver l'appel d'une fonction tout seul, sur une ligne à part.
Il se trouve forcément dans un calcul, une affectation, un affichage, un test, …

Exemple d'affectation de variable à partir du résultat d'une fonction ;
total ← calculeTTC( )

.7.4 Communication entre programme appelant et sous-programme

Les données et les résultats d'un sous-programme proviennent d'une communication entre le programme appelant et le sous-programme.
3 modes de communication sont possibles :

- L'utilisation de variables communes à l'appelant et à l'appelé : ce sont les variables globales (méthode la moins conseillée) ;
- Le passage de paramètres, pour les procédures et les fonctions ;
- La valeur de retour pour les fonctions.

Utilisation de variables globales

En algorithmique, une variable globale est déclarée **à l'extérieur du programme principal et des sous-programmes** : elle est commune à l'ensemble des sous-programmes et du programme principal.

Syntaxe :
```
VAR globales
…
PROCEDURE …
…
FONCTION …
…
PROGRAMME PRINCIPAL
VAR
…
```

Une telle variable est accessible par tous les sous-programmes ainsi que par le programme principal. On peut ainsi passer des paramètres dans les deux sens entre un programme et un sous-programme.

C'est là également l'inconvénient : un sous-programme qui comporte une erreur peut modifier la valeur d'une variable qui sera ensuite utilisée à d'autres endroits de l'application, provoquant une corruption des données.

Variable locale

Une variable est locale si elle est déclarée à l'intérieur d'un sous-programme. Elle n'est accessible que par le sous-programme dans lequel elle a été déclarée.

Le même nom de variable peut donc désigner plusieurs variables locales différentes déclarées dans différents sous-programmes.

> C'est fréquemment le cas des variables compteur, en général nommées **i**.

Passage de paramètres par valeur

Avec cette technique, les paramètres sont utilisés seulement en entrée du sous-programme.
On déclare des variables locales au sous-programme dans les parenthèses situées à côté du nom du sous-programme.

```
Syntaxe (pour une procédure) :
PROCEDURE nomProcedure(E param1:type, E param2:type,...) param1, param2 sont des paramètres formels
VAR locales
        ...
DEBUT
        // Bloc d'instructions
FIN
```

Ces paramètres sont des paramètres auxquels on va passer des valeurs lorsque l'on va appeler le sous-programme.
Le sous-programme travaillera alors avec des copies des variables de l'appel.

Exemple d'appel :
nomProcedure(v1, v2) **v1 et v2 sont les paramètres effectifs**

On appelle la procédure en passant les valeurs de v1 et v2 qui vont être affectées aux variables locales param1 et param2 de la procédure.

> Attention : il est nécessaire que les valeurs passées au sous-programme correspondent aux types déclarés lors de la définition du sous-programme.

Passage de paramètres par adresse

Avec cette technique, les paramètres sont utilisés en entrée et en sortie du sous-programme, ils sont donc modifiables par le sous-programme.
On déclare des variables locales au sous-programme dans les parenthèses situées à côté du nom du sous-programme.

```
Syntaxe (pour une procédure) :
PROCEDURE nomProcedure(E/S param1:type, E/S param2:type,...)
VAR locales
        ...
DEBUT
        // Bloc d'instructions
FIN
```

Ces paramètres sont des paramètres dont on va passer l'adresse mémoire lorsque l'on va appeler le sous-programme. Le sous-programme travaillera alors avec la ou les **variable(s) originale(s)**.

Exemple d'appel :
nomProcedure(@v1,@v2)
@v1 et @v2 sont les adresses mémoire des variables v1 et v2

8 Les tableaux

.8.1 Tableaux à une dimension

A. Exemple introductif

Saisir la liste des 12 notes sur 30 *16 23 8 19 28 20 18 14 10 9 15 24* Voici la liste de ces notes sur 20 10.67 15.33 5.33 12.67 18.67 13.33 12 9.33 6.67 6 10 16

Dans l'état actuel de vos connaissances, pour écrire l'algorithme qui donne la sortie d'écran suivante, vous êtes obligé de déclarer 12 variables différentes. On ne peut pas utiliser une seule variable qui prend successivement chaque valeur. Passe encore avec 12 notes, mais si l'on voulait réaliser ce traitement avec 30 ou 40 notes, cela deviendrait fastidieux.
En outre, le même traitement est effectué 12 fois sur des variables différentes. Comme les variables ont des noms différents, on ne peut pas utiliser de boucle, ce qui allonge considérablement le code et le rend très répétitif.

Pour résoudre ces problèmes, il est tentant d'utiliser un nom commun pour toutes les variables et de les repérer par un numéro. Ainsi, on pourrait déclarer toutes les variables d'un seul coup et utiliser une boucle pour effectuer le traitement en faisant varier le numéro des variables. Cela est possible grâce à l'utilisation d'un tableau.

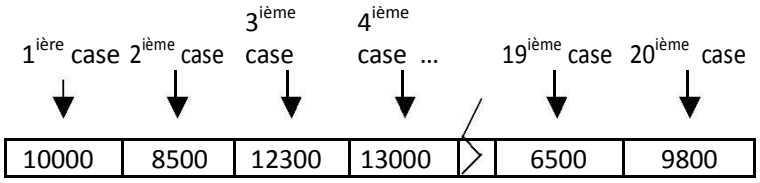
```
Programme conv_note
Var
    note: tableau[1..12] de réels
    i: entier

Début
    Afficher "Saisir la liste des 12 notes sur 30"
    Pour i de 1 à 12 Faire
        Saisir note[i]
    FinPour
    Afficher "Voici la liste de ces notes sur 20"
    Pour i de 1 à 12 Faire
        Afficher note[i]*2/3
    FinPour
Fin
```

B. Caractéristiques d'un tableau à une dimension

Un tableau à une dimension (ou vecteur) peut être vu comme une liste d'éléments. On le représente souvent comme une suite de **cases** contenant chacune une valeur.

Ex : Tableau salaire



- Un tableau possède un **nom** (ici salaire) et un nombre d'**éléments** (de cases) qui représente sa **taille** (ici 20).
- **Tous les éléments d'un tableau ont le même type** (c'est normal car ils représentent des valeurs logiquement équivalentes).

- Pour désigner un élément, on indique le nom du tableau suivi son **indice** (son numéro) entre crochets : **salaire [3]** représente le 3ième élément du tableau salaire et vaut 12300.

Remarque:

Dans un programme, chaque élément d'un tableau est repéré par un indice. Dans la vie courante, nous utilisons souvent d'autres façons de repérer une valeur. Par exemple, au lieu de parler de salaire [1], salaire[2], salaire [3], nous préférons parler des salaires de Mr Dupond, de Mme Giraud et de Mr Fournier. Le tableau ne permet pas de repérer ses valeurs autrement que par un numéro d'indice. Donc si cet indice n'a pas de signification, un tableau ne permet pas de savoir à quoi correspondent les différentes valeurs.

Les variables d'un tableau

La notion de « case contenant une valeur » doit faire penser à celle de variable. Et, en effet, **les cases du tableau**, encore appelées éléments du tableau, sont des **variables**, qualifiées d'**indicées**.

Différence entre variables classiques et variables indicées:

- les **variables classiques** sont déclarées individuellement et ont un nom distinct ;
- les **variables indicées** (constituant le tableau) sont **implicitement déclarées** lors de la déclaration du tableau. Pour bien montrer que ces variables n'ont pas de signification propre, mais sont juste « une des variables du tableau », elles ne possèdent pas de nom propre, mais juste un numéro appelé indice (de 1 à *n* si le tableau possède *n* cases). Chaque case (variable) est donc totalement identifiée par son indice et le nom du tableau.

Le tableau lui-même constitue aussi une variable. C'est une **variable complexe** (par opposition aux variables simples) car il est constitué d'autres variables (les éléments du tableau).

C. Déclaration et manipulation des tableaux à 1 dimension

1. Déclaration

1.1 Tableau variable

La syntaxe de la déclaration d'une variable tableau est la suivante:

*identificateur***: tableau [***valeur_indice_minimum* **..** *valeur_indice_maximum***] de** *type_des_éléments*

ex: sal : tableau[1.. 20] de réels
notes : tableau [1..10] de réels
nom_clients: tableau [1..20] de chaînes

Remarques:
Lorsqu'on déclare un tableau, on déclare aussi de façon implicite toutes les variables indicées qui le constituent.

En règle générale, l'indice minimum vaut 1. Mais on peut aussi utiliser un autre indice minimum, comme 0. Dans ce cas, l'indice maximum sera égal au nombre d'éléments – 1.

Lorsque l'indice minimum est 1, la **taille** du tableau (nombre d'éléments) est égale à la valeur de l'**indice maximum**

Certains langages acceptent que les valeurs minimales et maximales d'indice soient des variables. Mais comme ce n'est pas le cas de tous les langages, nous nous limiterons à des **valeurs d'indice constantes**.

1.2. Tableau constant

Il est parfois utile de déclarer un tableau constant, c'est-à-dire un tableau dont les valeurs ne peuvent pas changer. Dans ce cas, comme pour toutes les autres constantes, il faut initialiser les valeurs du tableau dès la déclaration. Pour cela, on utilise une liste d'initialisation entre accolades :

Exemple : Voici un tableau constant qui mémorise le libellé des mois :

libmois[1..12] ={"janvier", "février", "mars", "avril", "mai", "juin", "juillet", "août", "septembre", "octobre", "novembre", "décembre"}

2. Manipulation

Les tableaux peuvent se manipuler de deux manières:

- soit à travers leurs éléments le nom du tableau est alors suivi d'un indice entre crochets c'est la méthode que nous utiliserons
- soit dans sa globalité le nom du tableau n'est pas suivi d'indice nous n'étudierons pas cette possibilité de manipulation car elle fait appel à des concepts mathématiques qui n'ont pas été vus.

Les éléments d'un tableau sont des variables indicées qui **s'utilisent exactement comme n'importe quelles autres variables classiques**. Autrement dit, elles peuvent faire l'objet d'une affectation, elles peuvent figurer dans une expression arithmétique, dans une comparaison, elles peuvent être affichées et saisies…

L'**indice** d'un élément peut être:
- directement une valeur ex: salaire [10]
- une variable ex: salaire [i]
- une expression entière ex: salaire [k+1] avec k de type entier

Quelque soit sa forme, **la valeur de l'indice** doit être :

- **entière**
- **comprise entre les valeurs minimales et maximales** déterminées à la déclaration du tableau.

Par exemple, avec le tableau tab [1..20], il est impossible d'écrire tab[0] et tab[21]. Ces expressions font référence à des éléments qui n'existent pas.

Le fait que les éléments constituant un tableau soient indicés permet de **parcourir tous les éléments avec une boucle**. Le ***Pour*** est généralement la boucle la plus adaptée, puisque l'on connaît le nombre de fois qu'on doit effectuer le traitement (une fois par élément). On utilise une variable qui sert d'indice et s'incrémente à chaque tour de boucle.

Remarques :

Il est très grave de confondre l'indice d'un élément et la valeur de cet élément. En d'autres termes, il ne faut pas confondre i avec tab[i]

De même qu'une variable possède toujours une valeur (qui peut être indéterminée), un tableau peut très bien posséder des éléments dont la valeur est indéterminée. Donc avant d'utiliser un tableau, il est donc nécessaire de l'initialiser

3. Synthèse

- **Un tableau est une structure de donnée permettant de mémoriser des valeurs de même type et logiquement équivalentes**.
- Un **vecteur** est un tableau à une dimension (un seul indice permet d'identifier toutes les valeurs)
- Chaque **élément** d'un tableau est repéré par un **indice** indiquant sa position.
- La **taille** d'un tableau est le nombre de ses éléments. La taille d'un tableau est **fixe**.
- Les indices doivent obligatoirement être entiers et varier entre une **valeur minimale** (en général 1) et une **valeur maximale constante**.
- Chaque élément du tableau est une **variable indicée**, identifiée par le **nom du tableau suivi de l'indice entre crochets**. Les variables indicées s'utilisent **comme des variables classiques**.
- **Déclaration**
  Un tableau nommé toto de 10 éléments entiers se déclare ainsi : **toto : tableau[1..10] d'entiers**

.8.2 Tableaux à deux dimensions et plus

La syntaxe de la déclaration d'une variable tableau à plusieurs dimensions est la suivante:

identificateur: **tableau [***valeur_indice_minimum* **..** *valeur_indice_maximum, valeur_indice_minimum* **..** *valeur_indice_maximum***, [***valeur_indice_minimum* **..** *valeur_indice_maximum***]] de** *type_des_éléments*

ex : Notes : tableau[1.. 20, 1..5] de réels

Tableau Notes contient les notes de 20 étudiants à 5 devoirs

	1	2	3	4	5
1	10	12	15	08	7.5
2	8	9.5	10	14	10
3	7.5	12	13	12.5	0
…	…	…	…	…	…
…	…	…	…	…	…
20	18	14	10	12	10.5

Deux indices sont nécessaires pour parcourir ce tableau : **Notes [i, j]** pour accéder à la note du ième étudiant au jème devoir.

Si le tableau est constant, on peut le « remplir » lors de sa déclaration, ligne par ligne :
VAR Notes[1..20, 1..5] ={ {10, 12, 15, 08, 7.5}, {8, 9.5, 10, 14, 10}, {…. }, { ……}, {18, 14, 10, 12, 10.5}}

autre ex: Notes : tableau[1.. 20, 1..5, 1..2] de réels

Tableau Notes contient les notes de 20 étudiants à 5 devoirs pour 2 classes.
Trois indices sont nécessaires pour parcourir ce tableau : **Notes [i, j, k]** pour accéder à la note du ième étudiant au jème devoir pour la kème classe.