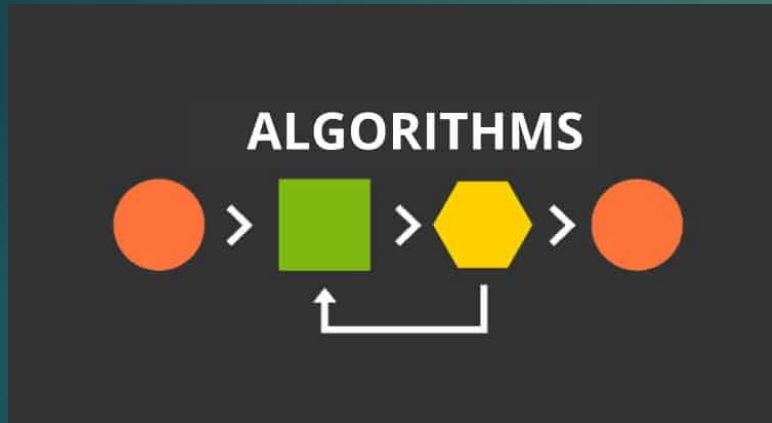


ALGORITHMIQUE



DE LA REFLEXION A LA PROGRAMMATION

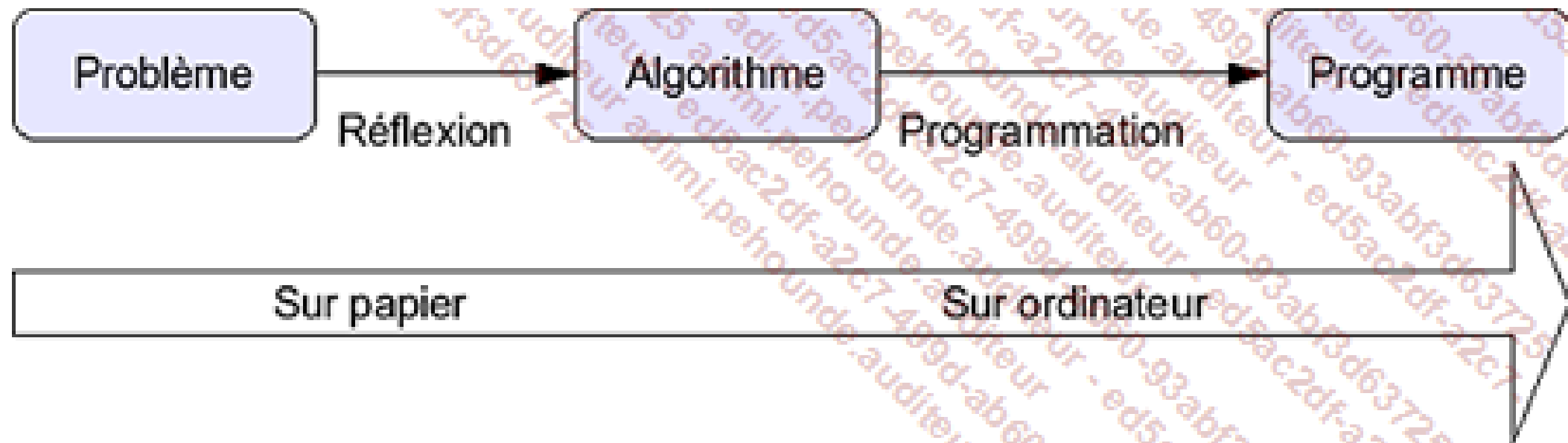
Introduction

L'algorithmique:

- ❖ c'est une suite d'instructions qui, quand elles sont exécutées correctement, aboutissent au résultat attendu.
- ❖ C'est un énoncé dans un langage clair, bien défini et ordonné qui permet de résoudre un problème, le plus souvent par calcul.

L'algorithme décrit formellement ce que doit faire l'ordinateur pour arriver à un but bien précis. Ce sont les instructions qu'on doit lui donner. Ces instructions sont souvent décrites dans un langage clair et compréhensible par l'être humain : faire ceci, faire cela si le résultat a telle valeur, et ainsi de suite.

Un algorithme bien établi et qui fonctionne (tout au moins en théorie) pourra être directement réécrit dans un langage de programmation évolué comme le C, Java, C# ou PHP. Malheureusement, en programmation c'est souvent à l'homme de se mettre au niveau de la machine.



De la réflexion à la programmation

Cas pratique

Avez-vous déjà eu l'occasion de faire la cuisine ?

Vous avez alors très probablement suivi une recette pour récupérer la liste et la quantité de chaque ingrédient et la liste des étapes de confection : faites fondre le chocolat et le beurre dans une casserole à feu doux, retirez

la casserole du feu, incorporez les jaunes d'œuf, puis le sucre et la farine, battez les œufs en neige puis incorporez doucement dans le mélange, etc. Dans tous ces cas, félicitations ! Vous avez déroulé votre premier algorithme !

Comment écrire l'algo?

Pour écrire un algo, on utilise généralement une série de conventions appelée « **pseudo-code** », qui ressemble à un langage de programmation authentique dont on aurait évacué la plupart des problèmes de syntaxe. Ce pseudo-code est susceptible de varier légèrement d'un livre (ou d'un enseignant) à un autre.

Syntaxe de base

\\ Pseudo-code tout bête!

DÉBUT

Instruction,

Serie d'instructions ...

FIN

Exemple

DÉBUT

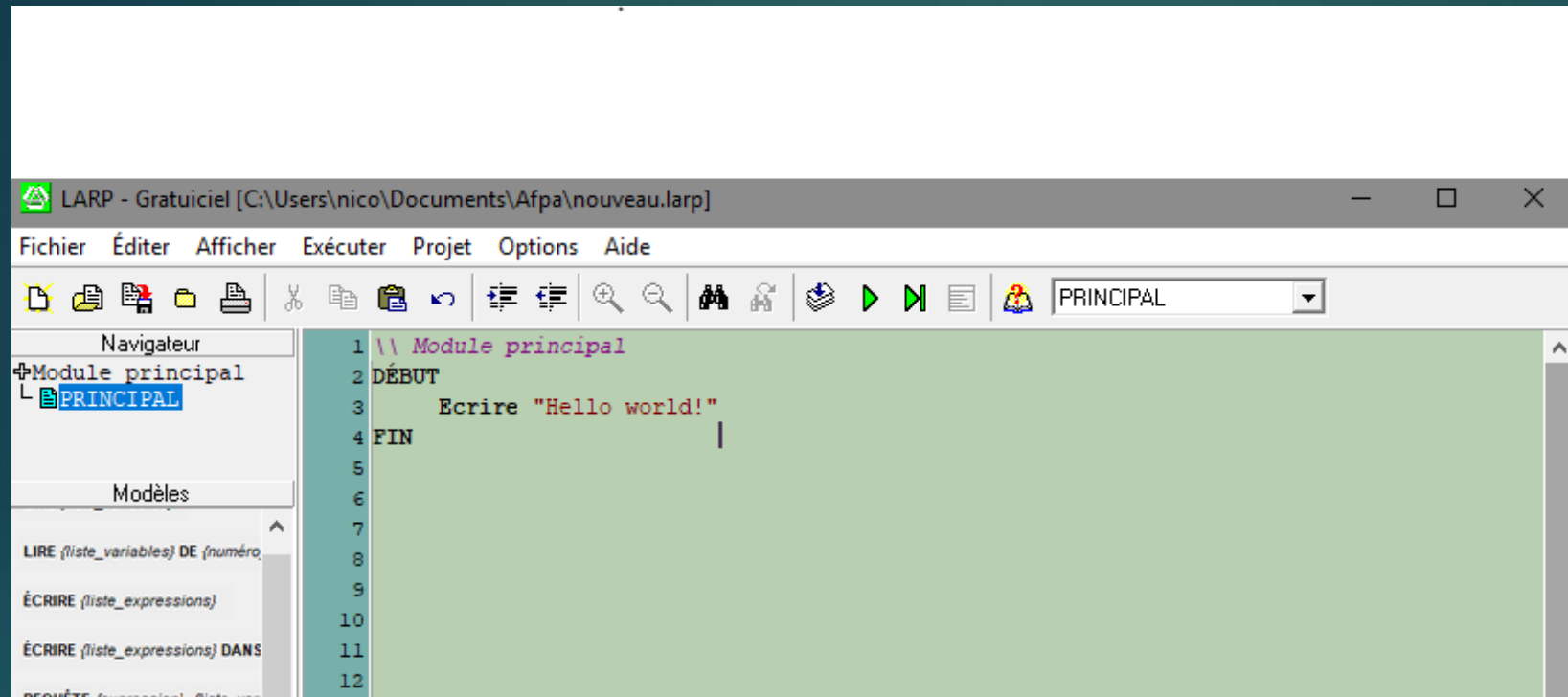
ÉCRIRE "Bonjour le monde!"

FIN

Pour tester nos algos nous utiliseron; LARP - Logiciel d'Algorithmes et de Résolution de Problèmes.

Il est téléchargeable à l'adresse suivante : <http://larp.marcolavoie.ca/fr/default.htm>

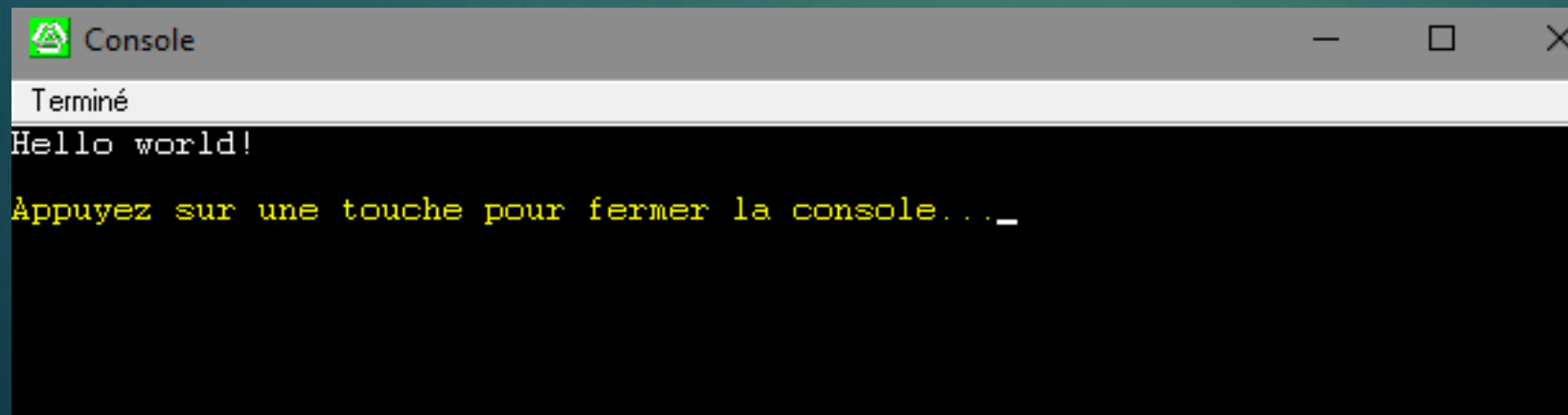
L e code



The screenshot shows the LARP - Gratuiciel application window. The title bar reads "LARP - Gratuiciel [C:\Users\nico\Documents\Afpa\nouveau.larp]". The menu bar includes "Fichier", "Éditer", "Afficher", "Exécuter", "Projet", "Options", and "Aide". The toolbar contains icons for file operations, editing, and execution. On the left, a "Navigateur" pane shows a tree structure with "Module principal" expanded, revealing a sub-item "PRINCIPAL". Below it, a "Modèles" pane lists several predefined code snippets. The main editor area displays the following code:

```
1  \\ Module principal
2  DÉBUT
3      Ecrire "Hello world!"
4  FIN
```

L'exécution



The screenshot shows the "Console" window of the LARP - Gratuiciel application. The title bar reads "Console". The text "Terminé" is displayed at the top. Below it, the output "Hello world!" is shown. At the bottom, a prompt "Appuyez sur une touche pour fermer la console..." is displayed, followed by a cursor.

Variables

Une variable est un nom ou une étiquette donné à une valeur (nombre, texte, etc.). Cette valeur peut varier au cours du temps : on affecte une nouvelle valeur au nom, d'où le nom de variable.

Une variable est caractérisée par : son identificateur (son nom) ; son type (par exemple numérique) ; son contenu (valeur prise par la variable à un niveau donné de l'algorithme). L'identificateur est le nom de la case réservée en mémoire, le type est la catégorie d'information qu'elle peut contenir, son contenu est l'information que l'on a mise dans la case.

Les variables permettent de stocker temporairement une information d'un algorithme pour la réutiliser plus tard. Les variables peuvent être de différents types mais nous retiendrons 3 types principaux:

Le type numériques

Le type chaîne de caractère

Le type booléen

Type numérique

D'une manière générale, deux types numériques sont utilisés en algorithmique :

Les **entiers** : nombres sans virgule, négatifs ou positifs ;

Les **réels** : nombres à virgule, positifs ou négatifs.

Exemple

- Entiers: 1, 227, 14 -25, -175, -5....
- Réels: 12.7, 4.26 ...

Type alphanumérique

Une variable peut aussi contenir des caractères. Suivant les livres et sites Internet, vous trouverez les types "Alphanumérique", "Caractère", "Chaîne", "String". Ainsi une variable peut stocker votre nom, une ligne complète de texte, ou tout ce que vous voulez qui nécessite une représentation alphanumérique. On appelle d'ailleurs une suite de caractères alphanumériques, une chaîne de caractères..

Exemples:

``coucou `` , ``Bonjour Mr Dupont`` .

Type booléen

Booléen est un type soit vrai ou soit faux.

Pour déterminer si une affirmation est vraie ou fausse, l'ordinateur doit se baser sur le résultat de cette affirmation. En informatique, on emploie plus volontiers la notion d'expression et d'évaluation de cette expression. Une expression peut être décrite comme étant tout ce qui peut fournir une valeur que l'ordinateur peut déterminer, stocker, évaluer. Par exemple, l'affirmation " $a > b$ " selon laquelle a est supérieur à b . Si a vaut 3 et b vaut 2, l'affirmation est vraie. Si maintenant a vaut 1 et b vaut 2, l'affirmation est fausse. Dans les deux cas " $a > b$ " est une expression qui vaut soit vrai, soit faux. C'est le cas le plus simple, mais aussi le plus courant et le plus pratique comme vous le constaterez lors des tests et des conditions.

Exemple:

true ou false

Opérateurs et calculs

1. Les affectations

Le symbole d'affectation " \leftarrow " ou " $=$ " fait partie d'une grande famille, celle des opérateurs. Comme son nom l'indique, un opérateur est utilisé pour et dans des opérations. Le symbole " \leftarrow " ou " $=$ " est un opérateur d'affectation. Il existe plusieurs opérateurs qui servent aux calculs, affectations, comparaisons, rotations (de bits), groupages, etc.

2. Les opérateurs arithmétiques

Pour que les algorithmes puissent effectuer des calculs, il faut pouvoir au moins faire des opérations simples. Vous utiliserez pour cela les symboles suivants :

+ : addition

- : soustraction

***** ou **x** : multiplication (il est plus facile d'écrire un x pour « multiplié par » qu'une étoile sur du papier, mais plus simple de mettre * sur un clavier)

/ : division

% ou **mod** : modulo

DIV : La division entière

Rappel : un modulo est le reste d'une division entière. Par exemple $15/2$ vaut 7, mais il reste 1. On dit que 15 modulo 2 vaut 1.

Ces opérateurs sont dites binaires car ils s'utilisent avec deux valeurs : une avant le symbole et une après. Les valeurs avant et après peuvent être des données (de même type que la variable qui reçoit les résultats) ou des variables. Voici un exemple d'opérations dans un simple algorithme qui calcule la surface et le périmètre d'un cercle.

Programme Cercle

```
Var
R, PI, surface, perimetre: réels
DEBUT
    PI = 3.1415927
    r = 5.2
    surface = PI * r * r
    perimetre = 2 * PI * r
    ECRIRE surface
    ECRIRE perimetre
FIN
```

Ici il n'y a que des multiplications. Vous pouvez déjà remarquer que vous avez parfaitement le droit de chaîner vos calculs et de mélanger les données et les variables.

3. Les opérateurs booléens

Ici il n'y a que des multiplications. Vous pouvez déjà remarquer que vous avez parfaitement le droit de chaîner vos calculs et de mélanger les données et les variables. Simulez les deux calculs :

Programme Cercle

Var

R, PI, surface, perimetre: réels

DEBUT

PI = 3.1415927

r = 5.2

surface = PI * r * r

perimetre = 2 * PI * r

ECRIRE surface

ECRIRE perimetre

FIN

Ici il n'y a que des multiplications. Vous pouvez déjà remarquer que vous avez parfaitement le droit de chaîner vos calculs et de mélanger les données et les variables.

3. Les opérateurs booléens

Les opérateurs logiques et, ou, non:

- L'opérateur **ET** indique que les deux expressions situées avant et après doivent être toutes les deux vraies pour que l'ensemble le soit aussi.
- L'opérateur **OU** indique que seule l'une des deux expressions, que ce soit celle située avant ou celle située après, doit être vraie pour que l'expression complète soit vraie aussi.
- Le **NON** est la négation. Si l'expression était vraie elle devient fausse, et vice versa.

Les trois opérateurs logiques ET, OU et NON peuvent être simplement compris à l'aide de petits tableaux appelés "**tables de vérité**". Exp1 et Exp2 sont des expressions booléennes vraies ou fausses. Par exemple l'expression $a=1$ est vraie si a vaut vraiment 1.

ET

<div>Exp2</div> <div>Exp1</div> <div>ET</div>		Vrai (1)	Faux (0)
		Vrai (1)	Faux (0)
Vrai (1)	Faux (0)	Vrai (1)	Faux (0)
Faux (0)	Faux (0)	Faux (0)	Faux (0)

Table de vérité ET

Ce tableau est à comprendre ainsi : si Exp1 est vraie et Exp2 est vraie, alors l'ensemble est vrai. On dit plus généralement que Exp1 ET Exp2 est vrai. Dans le cas du ET, l'ensemble est vrai uniquement si les deux expressions sont vraies : l'une ET l'autre. Dans les autres cas, comme au moins l'une des deux est fausse, alors le résultat total est faux.

OU

Exp1 \ Exp2	Vrai (1)	Faux (0)
OU	Vrai (1)	Faux (0)
Vrai (1)	Vrai (1)	Vrai (1)
Faux (0)	Vrai (1)	Faux (0)

Dans le cas du OU, au moins l'une des deux expressions doit être vraie pour que l'ensemble soit vrai. L'assertion est donc fausse dans le cas où les deux expressions sont toutes deux fausses.

Comparaison Inférieur, supérieur

Quatre opérateurs permettent de comparer des valeurs inférieures et supérieures, avec ou sans égalité :

$<$: inférieur

\leq ou $<=$: inférieur ou égal

$>$: supérieur

\geq ou $>=$: supérieur ou égal

La compréhension de ces quatre opérateurs ne doit pas poser de problème. Le résultat est vrai si la valeur de gauche est inférieure, inférieure ou égale, supérieure, supérieure ou égale à la valeur de droite.

LECTURE ÉCRITURE DE DONNÉES

- LIRE<nom de donnée, ...>
- ECRIRE<nom de donnée, ...>
- Fonction : Instructions permettant
 - de placer en mémoire les informations fournies par l'utilisateur.
 - De visualiser des données placées en mémoire
- Exemples:

DEBUT

 LIRE nom

 LIRE prenom

 ECRIRE (" le nom est " , nom, " et le prénom est " , prenom)

FIN

STRUCTURE ALTERNATIVE « SI ... ALORS ... SINON ... FSI » (1)

Une condition est donc une affirmation : l'algorithme, et le programme ensuite, détermineront si celle-ci est vraie, ou fausse.

Une condition retournant VRAI ou FAUX a comme résultat un **booléen**.

Une condition est souvent une comparaison. Pour rappel, une comparaison est une expression composée de trois éléments :

une première valeur : variable ou scalaire.

un opérateur de comparaison.

une seconde valeur : variable ou scalaire.

- Exemple :

Algorithme SimpleOuDouble {Cet algorithme saisit une valeur entière et affiche son double si cette donnée est inférieure à un seuil donné.)

constante (SEUIL : entier) = 10

Variable val : entier

début

ECRIRE("Donnez-moi un entier : ") { saisie de la valeur entière}

LIRE(val)

si val < SEUIL { comparaison avec le seuil}

alors ECRIRE ("Voici son double : " , val ×2)

sinon ECRIRE ("Voici la valeur inchangée : " , val)

STRUCTURE ALTERNATIVE « SI ... ALORS ... SINON ... FSI » (2)

- Ou instruction conditionnelle
 si <expression logique>
 alors instructions
 [sinon instructions]
 fsi
- Si l'expression logique (la condition) prend la valeur vrai, le premier bloc d'instructions est exécuté;
- si elle prend la valeur faux, le second bloc est exécuté (s'il est présent, sinon, rien).

STRUCTURE ALTERNATIVE « SI ... ALORS ... SINON ... FSI » (3)

- Autre écriture de l'exemple :

Algorithme SimpleOuDouble {Cet algorithme saisit une valeur entière et affiche son double si cette donnée est inférieure à un seuil donné.)

constante (SEUIL : entier) =10

Variable val : entier

Début

 ECRIRE("Donnez-moi un entier : ") { saisie de la valeur entière}

 LIRE(val)

 si val < SEUIL { comparaison avec le seuil} alors

 val = val ×2 Finsi

 ECRIRE ("Voici la valeur val : " , val)

fin

STRUCTURES ALTERNATIVES IMBRIQUÉES

- Problème: afficher :
- "Reçu avec mention Assez Bien " si une note est supérieure ou égale à 12,
- " Reçu mention Passable" si elle est supérieure à 10 et inférieure à 12, et
- "Insuffisant" dans tous les autres cas.

```
si note ≥ 12
    alors ECRIRE( "Reçu avec mention AB" )
    sinon si note ≥ 10 alors
        ECRIRE("Reçu mention Passable" )
        sinon afficher("Insuffisant" )
    finsi
finsi
```

Les Boucles

Une boucle permet de répéter une instruction (ou une liste d'instructions) plusieurs fois. Il y a principalement deux types de boucles

- Les boucles pour répéter une instruction un certain nombre de fois, il s'agit de la boucle Pour .
- Les boucles pour répéter une instruction jusqu'à une condition d'arrêt, il s'agit des boucles Tant que.

Le passage dans une boucle est appelé itération .

➤ La structure de contrôle Pour

Une première manière de répéter plusieurs fois un ensemble d'instructions est de les positionner dans une boucle Pour. Cette boucle répète un certain nombre de fois les instructions qu'elle contient. Vous pouvez par exemple décider de répéter dix fois les mêmes instructions. Il est nécessaire que ce nombre de répétitions soit connu avant le début de la boucle. Si ce n'est pas le cas, il faudra utiliser une des autres boucles présentées dans ce chapitre.

Syntaxe :

```
Pour variable = valeur_initiale à valeur_finale  
    instructions
```

FPour

Cette boucle est délimitée entre les mots-clé Pour et FPour (Fin de Pour). Elle utilise une variable qui lui permet de connaître le nombre d'itérations, c'est-à-dire de répétitions, déjà effectuées. Cette variable est initialisée à l'entrée dans la boucle avec une valeur initiale et elle est incrémentée à chaque fin d'itération. Lorsque la variable atteint la valeur finale, la boucle est terminée et le traitement se poursuit avec les instructions positionnées après la boucle.

Exemple :

Algo TableDeMultiplication

affiche la table de multiplication de 7

```
1  \\ Module principal
2  DÉBUT
3
4      Table = 7
5      POUR n = 0 JUSQU'A 10    FAIRE
6          Ecrire n,"x",Table,"=", n*Table
7      FINPOUR
8
9  FIN
```

```
Terminé
0 x 7 = 0
1 x 7 = 7
2 x 7 = 14
3 x 7 = 21
4 x 7 = 28
5 x 7 = 35
6 x 7 = 42
7 x 7 = 49
8 x 7 = 56
9 x 7 = 63
10 x 7 = 70

Appuyez sur une touche pour fermer la console..._
```


Cet algorithme affiche tout d'abord : Table de 7. Ensuite, la boucle commence. La variable m est initialisée à la valeur zéro. Une première itération est réalisée. L'unique instruction de la boucle affiche donc la valeur de la variable m, c'est-à-dire zéro, puis la chaîne de caractères " × " suivie de la valeur de la constante TABLE, soit sept, et enfin, après la chaîne de caractères " = ", est affiché le résultat du calcul $m \times \text{TABLE}$, donc zéro.

Comme cette itération est terminée, la variable m est incrémentée, elle vaut donc un. Comme la valeur finale n'a pas été atteinte, une nouvelle itération est réalisée avec cette nouvelle valeur. Cela est donc répété jusqu'à ce que la variable m atteigne la valeur finale. Une dernière itération est alors réalisée avec cette valeur, puis la boucle se termine. Et dans cet exemple, puisqu'il n'y a pas d'instructions après la boucle, le programme se termine. Voici donc ce qui est affiché par cet algorithme :

Table de 7

$$0 \times 7 = 0$$

$$1 \times 7 = 7$$

$$2 \times 7 = 14$$

$$3 \times 7 = 21$$

$$4 \times 7 = 28$$

$$6 \times 7 = 42$$

$$7 \times 7 = 49$$

$$8 \times 7 = 56$$

$$9 \times 7 = 63$$

$$10 \times 7 = 70$$

Parfois, entre chaque itération, il n'est pas souhaitable que la variable soit incrémentée d'une seule unité. Il est possible de changer ce pas en le précisant après le mot-clé par :

Pour variable <- valeur_initiale à valeur_finale par pas

instructions


FPour

La structure de contrôle TantQue

Une deuxième manière de répéter des instructions est d'utiliser une boucle TantQue. Contrairement à la boucle Pour, il n'est pas nécessaire de connaître avant de commencer la boucle le nombre d'itérations qui seront effectuées. Le principe de cette boucle est d'être répétée tant que la condition indiquée par celle-ci est vraie. Dans la vie de tous les jours, cela peut se traduire, par exemple, par la situation suivante : "Tant que la balance affiche un point inférieur à 400 g, je verse de la farine".

Syntaxe :

```
TANTQUE condition
    Instructions
FINTANTQUE
```



Cette boucle est délimitée par les mots-clés TantQue et FTq (Fin de TantQue). Comme le mot-clé TantQue le suggère, la condition indiquée à sa suite correspond à ce qui doit être vrai pour commencer une première itération ou pour recommencer avec une nouvelle itération. Autrement dit, tant que cette condition est vraie, on reste dans la boucle, et dès lors que cette condition devient fausse, la boucle se termine et l'exécution se poursuit avec les instructions situées après cette boucle.

La conséquence de cela est qu'il est possible de rester à tout jamais dans une boucle TantQue ; c'est ce qui se nomme une boucle infinie.

Exemple d'un code incorrect pouvant provoquer une boucle infinie :

Algo PeutNeJamaisTerminer

exemple de boucle infinie si la valeur saisie

n'est pas la bonne réponse

Variable saisie : texte

```
1  \\ Module principal
2  DEBUT
3  ECRIRE  "Quelle est la capitale de la France ?"
4  LIRE saisie
5  TANTQUE saisie <>| "Paris" FAIRE
6  ECRIRE "Vous vous êtes trompé !"
7  FINTANTQUE
8  FIN
9
10
```

Si la valeur saisie est différente de Paris, alors la condition `saisie ≠ "Paris"` est vraie et on rentre alors dans la boucle. Le message d'erreur s'affiche une première fois. Le problème c'est qu'il n'y a, au sein de la boucle, aucune instruction modifiant la valeur de la variable `saisie`. La condition qui était vraie initialement reste vraie et le restera pour toujours ! Aussi, le message s'affichera indéfiniment.

Il est donc nécessaire de rajouter au sein de la boucle une instruction modifiant cette variable pour qu'à un moment, la condition puisse valoir faux et ainsi permettre la sortie de la boucle. Cette instruction s'appelle la *relance*.

Avant la boucle, il est nécessaire que la variable sur laquelle porte la condition ait été initialisée. Cette instruction s'appelle l'*amorçage*.

amorçage

Voici donc la structure complète d'une boucle TantQue :

TantQue condition
instructions
relance

Attention à ne pas oublier d'écrire une amorce et une relance lorsque vous écrivez une boucle TantQue !

Voici l'exemple précédent corrigé pour éviter de réaliser une boucle infinie :

Algo QuestionGeographie

fait saisir l'utilisateur jusqu'à

ce qu'il saisisse la bonne réponse

```
1  \\ Module principal
2  DEBUT
3  ECRIRE "Quelle est la capitale de la France ?"
4  LIRE saisie
5  TANTQUE saisie <> "Paris" FAIRE
6  ECRIRE "Vous vous êtes trompé !"
7  LIRE saisie
8  FINTANTQUE
9  FIN
10
11
12
```

Voici un exemple d'exécution possible du code précédent :

Quelle est la capitale de la France ?

Lyon

Vous vous êtes trompé !

Réessayez... Capitale de la France ?


Nantes

Vous vous êtes trompé !

Réessayez... Capitale de la France ?

Paris

Bravo !



Dans cet exemple, l'utilisateur donne une mauvaise réponse lors de la saisie initiale. La condition de la boucle est donc vraie et le traitement à l'intérieur de la boucle est donc effectué. La relance permet de réinitialiser la valeur de la variable saisie. Malheureusement, l'utilisateur n'a toujours pas donné la bonne réponse. La condition de la boucle est donc encore vraie, donc une nouvelle itération de la boucle se produit. À nouveau, la relance réinitialise la variable saisie, cette fois-ci avec la valeur Paris. La condition de la boucle est testée une troisième fois, mais contrairement aux deux premières fois, elle est maintenant fausse. La boucle se termine donc et l'exécution se poursuit avec la suite du programme.

Remarquez que si l'utilisateur donne la bonne réponse du premier coup, on ne rentre pas dans la boucle et les instructions contenues dans celle-ci ne sont jamais exécutées. C'est une des caractéristiques de la boucle TantQue.

La structure de contrôle Répéter

Cette troisième boucle est assez similaire à la boucle TantQue. L'unique différence (mais qui fait toute la différence !) est que la condition de cette boucle n'est pas testée initialement, mais seulement après chaque itération. De ce fait, les instructions contenues dans la boucle sont exécutées au moins une fois. De même que pour la boucle TantQue, tant que cette condition est vraie, une nouvelle itération sera effectuée. Dans la vie de tous les jours, cela pourrait par exemple être utilisé pour saisir son code de carte bleue : il faut au moins le saisir une fois, mais cette opération peut être répétée si le code saisi est erroné. La condition de la boucle n'est donc pas testée initialement, mais seulement après avoir effectué une première itération.

Syntaxe :

Répéter

- réaffectation de la variable de la condition

- instructions

TantQue condition FRépéter

La boucle est délimitée par les mots-clés Répéter et FRépéter. Tout comme avec une boucle TantQue, il est possible de faire une boucle infinie si la variable sur laquelle porte la condition n'est jamais modifiée au sein de la boucle. À la fin de la boucle, le mot-clé TantQue est utilisé pour indiquer que la boucle sera répétée tant que la condition qui suit est vraie.

Il faut prendre garde à ne pas confondre la boucle Répéter avec une boucle TantQue même si, dans une boucle Répéter, le mot-clé TantQue est présent.

Exemple :

Algo ConvertisseurEuroBitcoin

Variable txConversion : réel

Variable valeurEuro : réel

Variable encore : texte

Début

txConversion <- saisir("Combien d'euros vaut 1 bitcoin ?")

Répéter

valeurEuro <- saisir("Quelle somme en euros voulez-vous " &
"convertir en bitcoins ?")

écrire("Cela représente une somme de " &
valeurEuro / txConversion & " bitcoin(s)")

encore <- saisir("Avez-vous encore une conversion à effectuer ?" &
"(oui/non)")

TantQue encore = "oui"

FRépéter

Fin

Cet algorithme réalise des conversions d'euros en bitcoins tant que l'utilisateur souhaite en effectuer.

Voici un exemple d'exécution de cet algorithme :

```
1 // Module principal
2 DEBUT
3 RÉPÉTER
4   ÉCRIRE "Combien d'euros vaut 1 bitcoin"
5   LIRE txConversion
6
7   ÉCRIRE ("Quelle somme en euros voulez-vous convertir en bitcoins ?")
8   LIRE valeurEuro
9   ÉCRIRE "Cela représente une somme de ", valeurEuro / txConversion , " bitcoin(s)"
10  ÉCRIRE "Avez-vous encore une conversion à effectuer ?" , "(oui/non)"
11  LIRE encore
12  JUSQU'A encore = "non"
13
14
15 FIN
16
```

Cet algorithme réalise des conversions d'euros en bitcoins tant que l'utilisateur souhaite en effectuer.

Voici un exemple d'exécution de cet algorithme :

Combien d'euros vaut 1 bitcoin ?

3085,53

Quelle somme en euros voulez-vous convertir en bitcoins ?

6000

Cela représente une somme de 1,94 bitcoin(s)

Avez-vous encore une conversion à effectuer ? (oui/non)

oui

Quelle somme en euros voulez-vous convertir en bitcoins ?

453,89

Cela représente une somme de 0,15 bitcoin(s)

Avez-vous encore une conversion à effectuer ? (oui/non)

non

Tout d'abord, le taux de conversion euros/bitcoins est demandé à l'utilisateur (il serait envisageable de récupérer automatiquement cette valeur sur Internet, mais cela est un peu trop avancé pour le moment...). Cette instruction est en dehors de la boucle, car elle n'est exécutée qu'une seule fois. Ensuite, le mot-clé Répéter indique le début de la boucle, et comme il n'y a pas de condition à l'entrée dans la boucle, les instructions présentes dans la boucle sont exécutées une première fois. L'algorithme effectue donc sa conversion, puis il initialise la variable encore sur laquelle porte la condition de la boucle. La première itération de la boucle se termine et c'est seulement à ce moment-là que la condition de la boucle est testée. Si elle vaut vrai, alors une nouvelle itération de la boucle est effectuée, sinon la boucle se termine.

Dans cet exemple, l'utilisateur qui exécute le programme veut effectuer au moins une conversion. C'est pourquoi la boucle Répéter est bien adaptée puisqu'elle effectue forcément une fois le traitement présent dans la boucle avant de tester la condition pour savoir s'il est nécessaire de la recommencer.

Dans la littérature, vous trouverez une boucle Répéter... Jusqu'à. Il s'agit de la même boucle, la seule différence est qu'il faut indiquer la condition pour sortir de la boucle à la place de la condition pour rester dans la boucle. Il suffit de rajouter l'opérateur Non et un couple de parenthèses autour de la condition pour réaliser la même chose avec une boucle Répéter... TantQue. La plupart des langages de programmation (sauf VB.Net, Pascal...) n'ont pas d'équivalent à cette boucle Répéter... Jusqu'à, c'est pourquoi elle n'est pas abordée dans cet ouvrage.

Le choix de la boucle la plus adaptée

Vous avez maintenant trois boucles à votre disposition pour écrire vos algorithmes. Une question légitime à se poser est de savoir dans quelles situations préférer l'une plutôt que l'autre. Pour répondre à cette question, je vous invite à vous poser deux questions :

Avant de commencer la boucle, le nombre d'itérations est-il connu ?

Si oui, alors la boucle la plus adaptée est la boucle Pour.

Si non, alors il est nécessaire de se poser une nouvelle question : est-il possible de ne réaliser aucune itération de cette boucle ?

Si oui, alors la boucle la plus adaptée est la boucle TantQue

Si non, alors il faut préférer une boucle Répéter

Ces règles constituent des bonnes pratiques. Il est tout à fait possible de ne pas les respecter : par exemple, il est tout à fait possible de réaliser une boucle dont le nombre d'itérations est connu avant son début à l'aide d'une boucle TantQue ou d'une boucle Répéter ! Néanmoins, votre code sera moins facile à comprendre, donc plus difficile à maintenir et à optimiser.

En Java, il existe des mots-clés permettant de sortir de manière anticipée d'une boucle (break) ou de ne pas terminer l'itération en cours (continue). Il est toujours possible de se passer de ces instructions en choisissant la bonne boucle et en écrivant la bonne condition pour celle-ci. Il est préférable de ne pas avoir recours à ces mots-clés, qui cassent le déroulement normal de la boucle et rendent également le code plus difficile à lire et à optimiser.

Les boucles imbriquées

Dans le chapitre précédent, nous avons vu que les structures de contrôle peuvent être imbriquées les unes dans les autres. Ainsi, il est possible de mettre un Si dans une boucle. Il est également possible de nicher une boucle au sein d'une autre boucle.

Exemple d'algorithme avec une boucle imbriquée :

Algo TableDeMultiplication2

Variable i, a, b, rep : entier

Constante NB_QUESTIONS : entier <- 10

Début

Pour i <- 1 à NB_QUESTIONS

 a <- aléa(0, 10)

 b <- aléa(0, 10)

 rep <- saisir(a & " × " & b & " = ")

 TantQue rep ≠ a*b

 rep <- saisir("C'est faux ! Réessayez...")

 FTq

FPour

Fin

```
1  \\ Module principal
2  DEBUT
3  NB_QUESTIONS = 10
4
5  POUR i = 1 JUSQU'À NB_QUESTIONS FAIRE
6      a = aléa(0, 10)
7      b = aléa(0, 10)
8      ECRIRE a , " × " , b , " = ?| "
9      LIRE rep
10     TANTQUE rep <> a*b FAIRE
11         ECRIRE "C'est faux ! Réessayez..."
12         LIRE rep
13     FINTANTQUE
14
15
16 FINPOUR
17
18 FIN
19
```

Dans cet exemple, il y a une boucle Pour dans laquelle est nichée une boucle TantQue.

Dans un premier temps, oublions la boucle Pour et regardons le code qui est à l'intérieur. Une première interrogation sur une multiplication est posée à l'utilisateur. La boucle TantQue sert à vérifier la réponse de l'utilisateur. Si la réponse est bonne du premier coup, cette boucle sera sautée puisque la condition est fausse. Si par contre la réponse est erronée, et tant que celle-ci n'est pas la bonne, un message d'erreur est affiché et l'utilisateur doit à nouveau saisir sa réponse.

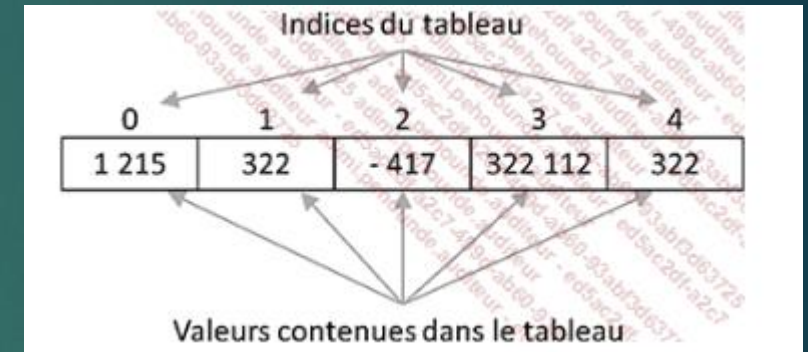
La boucle Pour permet de répéter dix fois l'ensemble de ces opérations. Ainsi cet algorithme permet de tester la connaissance de l'utilisateur de ses tables de multiplication.

LES TABLEAUX

Un tableau est donc une sorte particulière de variable capable de stocker un ensemble de valeurs, toutes de même type. Ces valeurs sont positionnées dans ce qui se nomme les cases du tableau. Afin de pouvoir s'y retrouver dans toutes ces cases, celles-ci sont numérotées par un nombre entier appelé indice.

Voici un exemple de tableau à 5 cases contenant chacune une valeur entière :

Les indices sont numérotés à partir de zéro. Ainsi la première case du tableau est à l'indice zéro, la seconde case a pour indice un. Un tableau de N cases a donc ses cases numérotées avec des indices allant de zéro à N-1.




La déclaration d'un tableau

Les constantes de type conteneur sont représentées à l'aide des crochets ([et]) à l'intérieur desquelles les éléments (i.e. les valeurs contenues dans le conteneur) sont énumérées, séparées par la virgule (,). L'exemple qui suit crée des conteneurs et les affecte à des variables :

```
Jours = ["Lu","Ma","Me","Je","Ve","Sa","Di"] \\ Conteneur de chaînes
```

```
Notes = [45, 78, 56, 96, 35] \\ Conteneur de valeurs \\ entières
```



Les éléments d'un conteneur peuvent être de différents types. Par exemple, le conteneur ci-dessous contient différentes données à propos d'un employé (son nom, son matricule, son salaire, l'année de son embauche et le montant de ses quatre dernières paies). Un conteneur peut même contenir d'autres conteneurs (les quatre paies sont regroupées dans un conteneur) :

```
Dossier = ["Gustave Labrie", 2013345, 56320.00, 1996, $  
[1401.98, 1456.02, 1399.57, 1423.41]]
```

Le parcours d'un tableau

Il est assez courant d'avoir à répéter une opération sur l'ensemble des cases du tableau. Pour cela, une boucle Pour est utilisée et la valeur de la variable de cette boucle est utilisée comme valeur d'indice pour accéder à chaque case du tableau.

Exemple

```
1  \\ Module principal
2  DEBUT
3
4  tab = [17,51,43,32]
5  POUR i = 1 JUSQU'À 3 FAIRE
6      ECRIRE tab[i]
7  FINPOUR
8
9  FIN
10
```

Lors de la première itération de la boucle, la variable *i* vaut zéro donc *tab[i]* permet d'accéder à la case d'indice zéro du tableau, c'est-à-dire la première case du tableau. Celle-ci est initialisée avec la valeur un. Lors de la deuxième itération, la variable *i* vaut un donc *tab[i]* correspond à *tab[1]* et cette deuxième case est initialisée à la valeur deux. Ainsi de suite, jusqu'à ce que la dernière case du tableau, la case d'indice neuf, soit initialisée avec la valeur dix. Ainsi, les cases du tableau ont été initialisées de la manière suivante :

0	1	2	3	4	5	6	7	8	9
1	2	3	4	5	6	7	8	9	10

Les tableaux multidimensionnels

Il est possible de déclarer un tableau à plusieurs dimensions. Voici par exemple un tableau à deux

	0	1	2	3	4	5	6	7	8	9
0	47	32	51	21	7	19	22	34	19	84
1	57	65	53	22	47	22	14	17	54	39
2	74	0	2	15	36	24	17	10	35	20
3	50	24	67	52	24	94	8	4	12	53

Afin de créer un tableau multidimensionnel, il faut déclarer un tableau de la même manière que précédemment, mais il faut en plus rajouter la taille pour les dimensions supplémentaires.


Syntaxe :

Variable nom_du_tableau :

type[dimension1][dimension2]...[dimensionN]

Une question se pose : pour créer un tableau d'entiers avec quatre lignes et dix colonnes, faut-il déclarer Variable tab2d : entier[4][10] ou Variable tab2d : entier[10][4] ? En fait, les deux façons de faire sont possibles : dans la mémoire, les valeurs ne sont pas organisées de cette manière-là. Par contre, une fois que ce choix a été effectué, il faudra être cohérent : si la première dimension correspond aux lignes et la seconde aux colonnes, il faut que par la suite ce soit toujours le cas. Dans cet ouvrage, c'est ce choix qui est fait. Ainsi, pour déclarer ce tableau, il faut indiquer :

Variable tab2d : entier[4][10]




Pour accéder à une case d'un tableau multidimensionnel, il ne faut plus préciser qu'un seul indice, mais autant d'indices qu'il y a de dimensions. Ainsi une et une seule case est désignée. Par exemple pour accéder à la case de la troisième ligne et de la septième colonne du tableau précédent, il faut indiquer `tab2d[2][6]`. Une fois précisée une case déterminée du tableau, il est possible de la manipuler comme si nous avions affaire à une simple variable.

Les tableaux multidimensionnels ne sont pas très différents des tableaux à une seule dimension : ils contiennent tous un ensemble de cases, toutes de même type. La principale différence est la manière d'indiquer la case sur laquelle nous souhaitons réaliser une opération : avec un tableau à une dimension, il faut indiquer un unique indice, alors qu'avec un tableau multidimensionnel, il faut indiquer autant d'indices qu'il y a de dimensions. En fait, les tableaux multidimensionnels sont surtout très utiles pour représenter des choses qui ont intrinsèquement plusieurs dimensions : par exemple un plateau de jeu de dames (deux dimensions), une IRM cérébrale (trois dimensions)... Les indices correspondent donc aux coordonnées de la case à cibler.

Voici un exemple d'algorithme utilisant un tableau à deux dimensions :

```
1  \\ Module principal
2  DEBUT
3      NB_LIGNES = 2
4      NB_COLONNES = 3
5
6  tab2d[1][1] = "D"
7  tab2d[1][2] = "M"
8  tab2d[1][3] = "W"
9  tab2d[2][1] = "d"
10 tab2d[2][2] = "e"
11 tab2d[2][3] = "v"
12 POUR j = 1 JUSQU'A NB_LIGNES      FAIRE
13     POUR i = 1 JUSQU'A NB_COLONNES  FAIRE
14         ECRIRE tab2d[j][i]
15     FINPOUR
16     ECRIRE ""
17 FINPOUR
18
19 FIN
20
```



Cet algorithme crée un tableau de caractères à deux dimensions ayant deux lignes et trois colonnes. Dans la case de coordonnées (1; 1) est stocké le caractère D. Dans la case de coordonnées (1; 2), c'est-à-dire la deuxième case de la première ligne, est positionné le caractère M. Ainsi de suite, jusqu'à stocker dans la case de coordonnées (2; 3), c'est-à-dire la dernière case de la dernière ligne, le caractère W.

Ensuite, deux boucles Pour sont imbriquées l'une dans l'autre. La première répète le traitement qu'elle contient pour toutes les lignes. La seconde boucle est donc répétée pour chaque ligne et celle-ci effectue pour chaque colonne le traitement qu'elle contient, c'est-à-dire afficher la valeur contenue dans la case du tableau, mais sans retourner à la ligne. Une fois ce traitement effectué pour toutes les colonnes de la ligne du tableau, l'instruction ECRIRE " " n'écrit rien du tout sur la console, mais surtout retourne à la ligne. Ainsi les caractères présents dans la ligne suivante du tableau sont écrits dans la ligne suivante de la console.

LES FONCTIONS PREDEFINIES

Tout langage de programmation propose un certain nombre de fonctions ; certaines sont indispensables, car elles permettent d'effectuer des traitements qui seraient sans elles impossibles. D'autres servent à soulager le programmeur, en lui épargnant de longs – et pénibles – algorithmes.

Les fonctions permettent d'automatiser certaines actions.

```
E 1 DEBUT
2   prix = [12,6,25,2,4]
3   m = Max(prix)
4   ECRIRE m
5
6 FIN
7
8
```

```
1 DEBUT
2   prix = [12,6,25,2,4]
3   mi = Min(prix)
4   ECRIRE mi
5 FIN
6
```

```
E 1 DEBUT
2   notes = [15, 20, 13]
3   POUR i = 1 jusqu'à COMPTER(notes) FAIRE
4     ECRIRE "note : ", notes[i]
5   FINPOUR
6
7   ECRIRE COMPTER(notes)
8 FIN
9
```

LES PROCEDURES ET LES FONCTIONS

Les procédures et les fonctions sont des sous-programmes qui permettent une réutilisation du code plus facile. En effet, si par exemple on code l'algorithme de calcul de T.V.A. et qu'on l'utilise à divers endroits du programme, plutôt que de recopier le code à chaque fois, il est préférable de créer un sous-programme de calcul. La recopie a toutefois les inconvénients suivants :

La recopie du code s'accompagne souvent d'un changement des noms de variables, voire de la valeur de certains paramètres,
Chaque copie du code augmente la taille des programmes source et compilé,
La maintenance est plus difficile : s'il faut modifier l'algorithme, il faut modifier toutes les copies sans en oublier une seule.

Les procédures

Une procédure est un bloc d'instructions nommé et déclaré et appelé dans son corps à chaque fois que le programmeur en a besoin.

E

```
1  \\ Module auxiliaire salutation
2  ENTRER
3      ECRIRE "Bonjour tout le monde!"
4  RETOURNER
5
6
7
```

```
1  DEBUT
2      salutation
3  FIN
4
5
```

```

1  \\ Module auxiliaire salutation
2  ENTRER
3      ECRIRE "Quel est votre nom?"
4      LIRE nom
5      ECRIRE "Vous vous appelez| " , nom
6  RETOURNER

```

```

1  DEBUT
2      salutation
3  FIN
4

```

Les Fonctions

Une fonction est un bloc d'instructions qui retourne obligatoirement une et une seule valeur résultat à l'algorithme appelant. Une fonction n'affiche jamais la réponse à l'écran car elle la renvoie simplement à l'algorithme appelant.

```

1  \\ Module auxiliaire somme
2  ENTRER
3      s = 5+2|
4  RETOURNER s
5
6

```

```

1  DEBUT
2      addition = somme
3      ECRIRE "Le résultat est:",| addition
4  FIN
5

```

```

1  \\ Module auxiliaire somme
2  ENTRER
3      ECRIRE "Entrer le premier nombre"
4      LIRE a
5      ECRIRE "Entrer le deuxième nombre"
6      LIRE b|
7      s = a+b
8  RETOURNER s
9
10

```

```

1  DEBUT
2      addition = somme
3      ECRIRE "Le résultat est:",| addition
4  FIN
5

```


Passages de paramètres

Un sous-programme avec paramètres est très utile parce qu'il permet de répéter une série d'opérations complexes pour des valeurs qu'on ne connaît pas à l'avance.

Ex:

```
1  \\ Module auxiliaire somme
2  ENTRER a,b|
3    s = a+b
4  RETOURNER s
5
6
```

```
1  DEBUT
2    addition = somme(5,8)|
3    ECRIRE "Le résultat est:", addition
4  FIN
5
6
```

```
1  DEBUT
2    ECRIRE "Entrer nombre1"
3    LIRE n1
4    ECRIRE "Entrer nombre2"
5    LIRE n2
6    addition = somme(n1,n2)|
7    ECRIRE "Le résultat est:", addition
8  FIN
```

```
1  \\ Module auxiliaire somme
2  ENTRER a,b|
3    s = a+b
4  RETOURNER s
5
```