

# Chapitre 1 : Pseudo-code

# Définition

- Le pseudo-code permet de décrire facilement un algorithme avec un vocabulaire simple et sans connaissance du langage de programmation utilisé pour son implémentation.
- En ayant comme connaissances quelques principes de programmation, vous pouvez échanger en pseudo-code avec une autre personne qui utilise un langage de programmation que vous ne connaissez pas.

# Comment l'écrire?

- L'écriture du pseudo-code peut se faire sur un ordinateur, un bout de papier, etc.
- Il n'y a pas de standard pour l'écriture d'un algorithme en pseudo-code mais seulement quelques conventions partagées par un grand nombre de personnes.
- Nous utiliserons les mots anglophones au courant de ce cours.
- Cependant, chaque mot-clé aura son équivalent en français.

# Création d'un algorithme

- La structure générale d'un algorithme se fait de cette manière:

```
ALGORITHM  
    ...  
BEGIN  
    ...  
END
```

# Règles d'écriture d'un algorithme

- Pour tout langage, il faut définir l'alphabet (les caractères utilisés) et les mots qui définissent le langage.
- Il y aura deux types de mots dans un algorithme:
  - Les **mots clés**, prédéfinis dans le langage
  - Les *identifiants*, mots construits pour 'nommer' des variables, des procédures, etc.
- Pour les *identifiants*, il existe quelques règles de construction à suivre.

# Les identifiants

- Un *identifiant* est un nom déclaré et valide pour:
  - Une constante
  - Une variable
  - Une procédure
  - Une fonction
  - L'algorithme principal
- Les noms d'identifiants ne peuvent contenir que des caractères compris dans les intervalles suivants:
  - 'a' ... 'z'
  - 'A' ... 'Z'
  - '0' ... '9'
  - ' \_

# Les identifiants

- Pour construire un *identifiant*, il faudra respecter les règles suivantes:
  - Il ne peut en aucun cas commencer par un chiffre
  - Tout *identifiant* doit avoir été déclaré avant d'être utilisé
  - Un *identifiant* doit bien entendu être différent d'un mot clé, ceci pour éviter toute ambiguïté
  - Pour faciliter l'écriture et la lecture des algorithmes, il est très fortement conseillé d'utiliser des identifiants explicites (exemple: **largeur\_image** et non **xi**)

# Les mots clés

- Les **mots clés** seront utilisés pour construire les algorithmes, les déclarations et les instructions.
- Ceux-ci sont prédéfinis dans le langage
- Comme pour les *identifiants*, aucune distinction n'est faite entre les majuscules et les minuscules. Cependant, agissons comme si les mots clés DOIVENT être en majuscule.



# Les mots clés

- Voici une liste de mots clés principaux du langage:
  - **ALGORITHM**
  - **CONSTANTS**
  - **VARIABLES**
  - **BEGIN**
  - **END**
  - **IF**
  - **THEN**
  - **ELSE**
  - **FOR**
  - **WHILE**
  - **INTEGER**
  - ...

# Les séparateurs et symboles spéciaux

- La structure de l'algorithme (déclaration + instructions) est faite de telle manière qu'il n'y a pas besoin de séparateurs particuliers.
- Simplement la virgule sera utilisée comme séparateurs pour les listes de paramètres dans les appels de routines.
- Un certain nombre de caractères et de combinaisons de caractères ont une signification spéciale pour le langage algorithmique:

←	^	.	,	:	{	}	[	]	<=	>=	<>	=	+	-	*	/	(	)
---	---	---	---	---	---	---	---	---	----	----	----	---	---	---	---	---	---	---

# Les commentaires

- Il est vivement conseillé d'ajouter des commentaires dans son algorithme.
- Les commentaires ne sont pas obligatoires, mais **fortement suggérés**.
- Les blocs de commentaires sont délimités par les caractères //

# La partie déclaration

- La partie déclaration est l'endroit où nous définissons ce dont nous allons avoir besoin pour l'algorithme:
  - Constants
  - Variables
  - Fonctions

```
ALGORITHM algo1
    <déclaration des constantes>
    <declaration des variables>
    <déclaration des routines>
BEGIN
    <instructions>
END
```

- **L'ordre des déclarations est important, et ne peut être changé.**

# La partie déclaration

- Les **constantes** sont des variables qui ne peuvent être modifiées dans l'algorithme.
- Les variables, elles, contiennent des données manipulées par l'algorithme.
- Lors de la déclaration, on doit leur attribuer un type. Celui-ci doit être prédéfini dans le langage.
- Il existe plusieurs types différents en programmation, qui ont chacun leur propre rôle dans un fonctionnement d'algorithme.

# La partie déclaration

- Les constantes sont initialisées dès leur début. Cependant, les variables, quant à elle, doivent être initialisées au début de l'application.
- D'un côté plus technique, les constantes sont initialiser à la compilation, tandis que les variables au runtime.

# Les variables

- Les différents types prédéfinis en langage algorithmique que nous utiliserons sont:

- INTEGER	nombres entiers signés	42
- FLOAT	nombres flottants signés	0.154
- BOOL	énumération définissant les données <i>vrai</i> et <i>faux</i>	vrai
- CHAR	caractère ANSI sur un octet	'a'
- STRING	chaîne de caractères	"lapin"

# La déclaration des variables

- Variables simples :

```
VARIABLES
```

```
ident_type : ident_variable1, ident_variable2, ...
```

- Variables indicées ou tableaux

- Les tableaux permettent d'associer dans une même variable plusieurs données de même type avec un indice allant de l'indice minium (entier1) et l'indice maximum (entier2).

```
VARIABLES
```

```
ident_type : ident_tableau[<entier1>,<entier2>] , ...
```



# La déclaration des variables

- Un tableau peut comporter plusieurs dimensions délimitées par un point virgule:

VARIABLES

```
ident_type : ident_tableau[<entier11>,<entier12>;<entier21>,<entier22>] , ...
```

- Dans cet exemple, il s'agit d'un tableau à 2 dimensions avec un indice pour les lignes (compris entre entier11 et entier12) et un deuxième indice pour les colonnes (compris entre entier21 et entier22).
- L'accès à un élément d'un tableau se fait en indiquant la liste des indices correspondant à chaque dimension.

```
ident_var[<indice1>;<indice2>; ...]
```

# Les instructions - Expressions

- Une expression représente une succession de calculs
  - Elle peut faire intervenir des constantes, des variables, des fonctions et des opérateurs.
- Les expressions sont utilisées dans tout l'algorithme
  - Dans les affectations
  - En paramètre de routines
  - Dans les structures de contrôles
  - Etc.

# Les instructions - Expressions

- Une expression peut être:

- une <b>valeur</b>	42
- une <b>variable</b>	x
- une <b>constante</b>	C
- un appel à une <b>fonction</b>	<i>cos</i> (x)
- <expression> <b>OpérateurBinaire</b> <expression>	32 + x
- <b>OpérateurUnaire</b> <expression>	non A

# Les instructions – Opérateurs arithmétiques

- Il existe aussi une série d'opérateurs qui permettront de faire des calculs sur les différentes variables. Les opérandes suivants peuvent être soit des entiers (integer), soit des réels (float):
  - Integer (operator) integer = integer
  - Integer (operator) float = float
  - Float (operator) float = float

- (unaire)	Changement de signe
+	Addition
-	Soustraction
*	Multiplication
/	Division flottante

# Les instructions – Opérateurs arithmétiques

- Il existe des opérateurs de division entière qui sont utilisés soit:
  - Garder la partie entière d'une division

<b>DIV</b>	Division entière
------------	------------------

- Garder la partie restante de la division

<b>MOD</b>	Modulo (reste de la division entière)
------------	---------------------------------------

- **Ces opérateurs ne fonctionnent que sur des entiers (integer)!**

# Les appels aux fonctions et procédures

- L'appel d'une procédure ou d'une fonction (**routine**) se fait par son nom suivi, s'il y a lieu, de la liste des arguments placés entre parenthèses.
- Il faut respecter l'ordre de déclaration des paramètres.

# Les appels aux fonctions et procédures

- **APPEL DE PROCÉDURE = UNE INSTRUCTION**

- L'appel de procédure est une instruction à part entière :

```
FUNCTION_NAME (param1, param2, ...)
```

- **Exemples de procédures : les entrées-sorties**

- Les procédures d'affichage : **WRITE**

```
WRITE "Level: " + level
```

- affiche sur l'écran (ou écrit dans un fichier) la chaîne Level : ', suivi du contenu de la variable level.

# Les appels aux fonctions et procédures

- La procédure de lecture : **READ**



```
READ level
```

demande à l'utilisateur d'entrer un integer, float, char ou une string et affecte la variable level



# Les appels aux fonctions et procédures

- **APPEL DE FONCTION**

- Une fonction peut retourner une valeur.
- L'appel de fonction sera donc utilisable comme n'importe quelle autre valeur (dans une expression, en paramètre d'une fonction, ...).
- Par exemple dans une affectation :

```
variable = FunctionName(param1, param2)
```

- **Note : un appel de fonction seul n'est pas une instruction !**

# Les appels aux fonctions et procédures

- APPEL DE FONCTION

```
// Return the average of three numbers
FUNCTION CalculateAverage(n1, n2, n3) : INTEGER
    VARIABLES
        INTEGER : sum
        FLOAT : average
    BEGIN
        sum = n1 + n2 + n3
        average = sum / 3
    RETURN average
END

average = CalculateAverage(60, 80, 70)
```

# Les instructions – Opérateurs logiques

- Les opérandes sont booléens, on a alors une opération logique.
- Le résultat est un booléen.
- Les expressions booléennes sont utilisées comme conditions dans les structures de contrôles.

<b>! or NOT</b>	négation logique
<b>AND</b>	et logique
<b>OR</b>	ou logique

# Les instructions – Opérateurs logiques

- Voici un petit tableau qui permet de mettre en lumière les résultats des différents tests de conditions entre deux booléens:

a AND b	n'est vrai que si a est vrai et b est vrai	est faux dès qu'un des deux est faux
a OR b	n'est faux que si a est faux et b est faux	est vrai dès qu'un des deux est vrai

- **IMPORTANT:** Les opérateurs ET et OU sont séquentiels. Si l'évaluation du premier opérande suffit à donner le résultat, la deuxième n'est pas évaluée. Ainsi, si a est faux, a ET b sera faux, sans que b n'ait été évalué.

# Les instructions – Opérateurs relationnels

- Les opérateurs relationnels sont utilisés pour comparer deux variables ensemble.
- Par exemple, déterminer si la variable a et équivalente à la variable b.
- Le résultat de ce type d'opérateur est toujours un booléen, c'est-à-dire vrai ou faux.

==	égal
!=	différent
<	inférieur à
>	supérieur
<=	inférieur ou égal
>=	supérieur ou égal

# Règles d'évaluation des expressions

- Priorité des opérateurs, par ordre décroissant :

opérateurs unaires	- ; !
opérateurs multiplicatifs	* ; / ; div ; mod ; et
opérateurs additifs	+ ; - ; ou
opérateurs relationnels	= ; < ; <= ; > ; >= ; <> (ou !=)

# Règles d'évaluation des expressions

- Les expressions entre parenthèses sont entièrement évaluées avant d'intervenir dans la suite des calculs.
- **Concordance de type** : un opérateur *binnaire ne peut porter que sur deux valeurs du même type. Une exception a lieu lorsqu'une valeur est réelle et l'autre entière. Dans ce cas la valeur entière est convertie en une valeur réelle. Cette règle s'applique pour les opérateurs arithmétiques (+, -, \*, /) et ceux de comparaisons.*

# L'affectation

- Cette instruction permet d'affecter une valeur à une variable. La valeur peut être n'importe quelle expression de type compatible avec la variable.

```
variable = <value>
```

- Une valeur (une expression) ne peut en aucun cas figurer à gauche d'une affectation.
- Une variable figurant à droite d'une affectation (et plus généralement dans toute expression) doit obligatoirement contenir une valeur.



# Les structures de choix

- **L'ALTERNATIVE : IF - THEN - ELSE**
- Remarque : la partie **ELSE** <instruction> est facultative.
- **Attention** : Le **ENDIF** est obligatoire ! Il en sera de même pour toutes les instructions structurées
- Cette marque de fin doit être présente même s'il n'y a qu'une seule instruction.

```
IF <bool is true> THEN
    ...
ELSE // if the bool is false
    ...
ENDIF
```

# Structures de répétition

- LA RÉPÉTITIVE : WHILE

```
WHILE <bool is true>  
    ...  
ENDWHILE
```

# Structures de répétition

- **Fonctionnement**

- Les instructions sont répétées tant que la condition est vérifiée.
- Comme le test est au début, les instructions peuvent donc ne jamais être exécutées.

- **ATTENTION!**

- Il est impératif que la condition devienne fausse à un moment.
- Pour cela il faut que l'expression booléenne contienne au moins une variable qui sera modifiée dans la boucle.

# Structures de répétition

- LA RÉPÉTITIVE : DO...WHILE

DO

...

WHILE <bool is true>

# Structures de répétition

- **Fonctionnement**

- La condition est placée après les instructions, elles sont exécutées donc au moins une fois puis tant que la condition reste satisfaite.
- **ATTENTION!**
- Il est impératif que la condition devienne fausse à un moment.
- Pour cela il faut que l'expression booléenne contienne au moins une variable qui sera modifiée dans la boucle.

# Structures de répétition

- L'ITÉRATIVE : FOR

```
// FOR with increment of 1  
FOR i FROM 0 TO 10  
    ...  
ENDFOR
```

```
// FOR with increment of 2  
FOR i FROM 0 TO 10 INCREMENT OF 2  
    ...  
ENDFOR
```

# Structures de répétition

- **Fonctionnement**

- La variable est nécessairement de type scalaire : entier, caractère ou énumération.
- Les expressions de début et de fin doivent être compatibles avec elle.
- Elle prend successivement toutes les valeurs comprises entre les deux bornes, dans l'ordre croissant ou décroissant (si l'incrément est négatif).
- La déclaration de l'incrément lorsqu'il est unitaire peut être omise

# Structures de répétition

- **Fonctionnement**

- La boucle POUR est un cas particulier de la boucle TANT QUE.
- Si on connaît à l'avance le nombre de répétitions à effectuer, la boucle POUR est toute indiquée.
- A l'inverse, si la décision d'arrêter la boucle ne peut s'exprimer que par un test, c'est la boucle TANT QUE qu'il faut choisir.



# Les fonctions

- **LES FONCTIONS**

- Une fonction est un sous-algorithme effectuant un traitement et qui retourne une valeur.
- La fonction retourne une valeur au moyen de la procédure système **RETURN**.
- Celle-ci doit donc obligatoirement figurer dans les instructions de la fonction.
- La procédure système **RETURN** est débranchante : son exécution termine la fonction. Toute instruction placée après ne sera donc pas prise en compte.

# Les fonctions

- **LES FONCTIONS : SYNTAXE**
  - FUNCTION – BEGIN – RETURN – END

```
FUNCTION FunctionName(param1, param2, ...) : return_type
    VARIABLES
        TYPE : var1, var2, ...
        ...
    BEGIN
        ...
    RETURN <result>
END
```

# Les fonctions

- **PORTÉE DES IDENTIFIANTS**

- La portée d'un identifiant est la partie de l'algorithme dans laquelle cet identifiant est reconnu conformément à sa déclaration, c'est-à-dire l'ensemble des lignes de codes dans lesquelles l'utilisation de cet identifiant fera référence à la donnée qu'il définit.
- Un identifiant sera "*visible*" dans l'algorithme où il a été déclaré et dans tout sous algorithme appelé, mais jamais à un niveau plus haut.
- Il est possible d'avoir des "*conflits*" de portée, c'est-à-dire qu'un niveau d'imbrication de routine déclare un identifiant portant le nom d'un autre identifiant existant à un niveau supérieur
- La règle alors est la suivante : la version la plus proche (la plus profondément imbriquée) de l'identifiant a la priorité.