Chapitre 1

Notions de base

1. Informatique

Le mot informatique est utilisé pour désigner des activités très différentes les unes des autres. Lorsque quelqu'un vous dit "je fais de l'informatique" vous n'êtes pas pour autant renseignés sur son activité ? Est-il secrétaire utilisateur de traitement de texte, est-il comptable utilisateur d'un logiciel de comptabilité, est-il physicien utilisateur des services d'un centre de calcul, est-il informaticien programmeur, analyste, administrateur de réseau, ingénieur système ?

Ce cours s'adresse à de futurs utilisateurs éclairés des systèmes informatiques qui, à défaut d'écrire eux même leurs programmes, auront besoin de communiquer de manière précise avec des informaticiens et donc de connaître la discipline de l'intérieur. Ce cours est donc conçu comme une initiation à **l'analyse** et à la **programmation**, il n'est pas l'apprentissage à l'utilisation de logiciel.

Avant d'écrire les programmes qui résoudront les problèmes, nous écrirons des algorithmes. On précise ce qu'est un **algorithme**.

2. Algorithme

Dans le sens général, un algorithme peut prendre de nombreuses formes :

- recette de cuisine
- notice de montage
- feuille de calcul d'impôt
- protocole expérimental

...

Dans tous les cas, il s'agit de la description, faite par un individu A, d'un enchaînement d'actions destinées à êtres exécutées par un individu B. Pour que B puisse exécuter sans erreur les tâches commandées par A, il faut que la description faite par A soit claire, complète, et sans ambiguïté. Cette description est une forme d'algorithme.

Pour nous, l'individu B, celui qui exécute, doit pouvoir être remplacé par un ordinateur. Il faut alors que les consignes soient données dans un langage interprétable par l'ordinateur. L'ensemble des ces consignes, écrite dans un langage de programmation, s'appelle un programme. Il est exécuté par l'ordinateur.

Comme il est difficile et peu souhaitable (pour des raisons qui seront développées plus tard) d'écrire directement un programme pour un ordinateur, on écrit des algorithmes, qui sont l'expression formalisée des solutions aux problèmes que l'on doit résoudre. Que veut dire formaliser ? Par exemple au lieu de poser la question "quel nombre faut-il ajouter à 7 pour obtenir 15, on peut formaliser la question sous la forme de l'équation $7 + x = 15$.

Voici l'exemple d'un algorithme qui ressemble à ce que les étudiants écriront dans quelques semaines.

3. Exemple

```
Algorithme Intérêts
/* calcule le nombre d'années à attendre pour obtenir une somme donnée avec un
dépôt annuel identique chaque année. Ces données sont saisies au clavier par
l'utilisateur. Les montants sont en euros */

/* Déclarations */
Constantes
/* identificateur       = valeur            rôle */
   TAUX                 = 4.5               /* taux annuel */
Variables
/* identificateur       : type              rôle */
   depot                : réel              /* montant du dépôt annuel */
   somme                : réel              /* somme courante */
   sommeFinale          : réel              /* somme souhaitée */
   nbAn                 : entier            /* nombre d'années */
   interet              : réel              /* montant de l'intérêt annuel */

/* Instructions */
Début
1    somme ← 0       /* on part d'une somme nulle */
2    nbAn ← 0        /* on part d'un nombre d'années nul */

     /* entrée des données de l'utilisateur */
3    écrire("montant du dépôt annuel : ")
4    lire(depot)
5    écrire("montant de la somme souhaitée : ")
6    lire(sommeFinale)

     /* traitement */
7    tantque (somme < sommeFinale) faire
8      somme ← somme + depot            /* premier jour de l'année */
9      nbAn ← nbAn + 1
10     interet ← (somme * TAUX) / 100
11     somme ← somme + interet          /* dernier jour de l'année */
12   fintantque

     /* affichage des résultats */
13   écrire("il faudra attendre : ", nbAn, " années et vous recevrez : ", somme,
       " euros")
Fin
```

On suppose que :

- le dépôt est identique chaque année
- le dépôt est effectué le premier jour de l'année
- l'intérêt est ajouté le dernier jour de l'année
- on s'intéresse à la somme possédée le dernier jour de l'année

Remarque : Normalement un algorithme ne comporte pas de numéros de lignes. Ceux qui ont été introduits dans cet exemple ne sont là que pour faciliter la simulation qui sera faite plus loin..

4. Description des éléments d'un algorithme

Un algorithme se compose de 2 parties :

- les **déclarations**
- les **instructions**

Dans chaque partie, des **commentaires** doivent faciliter la compréhension du lecteur.

4.1. Les déclarations

C'est la partie de déclaration des **constantes**, des **variables, ...**

On pourrait la comparer à la mise en place du dispositif expérimental d'une expérience de chimie : préparer la verrerie adéquate pour recevoir les solutions, ...

4.1.1. Variables

Une **variable** sert à mémoriser une valeur. C'est une **zone de mémoire**, physique dans un ordinateur, virtuelle dans un algorithme. Sa **valeur peut changer** au cours du temps. Pour manipuler une variable, on lui donne un nom : son **identificateur**. Lorsqu'on utilise l'identificateur d'une variable, c'est en fait pour manipuler la valeur de la variable. Un identificateur doit commencer par une lettre, il se compose de lettres, de chiffres et/ou du tiret souligné. On pourra adopter la règle suivante pour les identificateurs de variables : elles sont composées de minuscules, sauf aux ruptures de mots qui seront en majuscule (ex : sommeFinale).

Une variable est aussi caractérisée par son **type**, c'est à dire **une contrainte sur les valeurs** qu'elle peut prendre. Si on déclare une variable de type entier, on ne pourra pas y mettre un réel. Dans un algorithme il faut veiller à la compatibilité des types de variables et des valeurs qu'on met dans les variables. Dans une machine, le type d'une variable correspond à la place qui est réservée dans la mémoire ranger les futures valeurs. On conçoit qu'il faudra réserver une plus grande place pour un réel que pour un entier.

Le **rôle** (qui est un commentaire) est indispensable pour indiquer à quoi sert la variable à moins que celui-ci ne soit explicitement exprimé par l'identificateur.

4.1.2. Constantes

Une **constante** correspond à une valeur littérale. A chaque fois que la constante apparaît dans un algorithme, elle est remplacée par sa valeur. On pourra adopter la règle suivante pour les identificateurs de constantes : elles sont composées de majuscule avec un tiret bas pour les rupture de mots (ex : TAUX_ANNUEL).

4.1.3. Types

Au début on travaillera avec 4 types :

Type	**en algorithmique**	**en machine**
entier	élément de Z	élément d'un sous ensemble de Z
réel	élément de R	élément d'un sous ensemble de Q
caractère	signe, lettre, chiffre ...	Idem – délimiteur : '
chaîne	suite de caractères	Idem – délimiteur : "

Lorsqu'on veut manipuler directement des valeurs de chacun de ces types, il faut prendre certaines précautions :

- pour les entiers ou réels, pas de problème : 13, 15.78
- pour les caractères (chaînes) il peut y avoir ambiguïté : en effet si j'écris N, je désigne l'identificateur d'une variable et non le caractère donc je dois écrire 'N' pour désigner le caractère. **Les valeurs de type caractère ou chaîne doivent être entourées par des apostrophes pour les distinguer des identificateurs**.

4.2. Les instructions

Elles constituent la partie active de l'algorithme. Le chapitre suivant définira précisément ce que sont les instructions. Ici on ne présente qu'un exemple d'exécution de l'algorithme précédent. Les instructions de l'algorithme s'exécutent séquentiellement au cours du temps et modifient certaines valeurs des variables. On peut comparer une instruction à un événement qui agit et modifie l'état du système sur lequel il agit. L'état du système est constitué par l'ensemble des valeurs des variables.

Chaque numéro de ligne fait référence à la ligne de l'algorithme. Il est convenu que, si la valeur d'une variable est modifiée par l'instruction courante, la nouvelle valeur de la variable est inscrite dans la case correspondante.

L'exemple ci-dessous présente une exécution de l'algorithme pour laquelle l'utilisateur a donné

- 800 pour le dépôt
- 2600 pour la somme à obtenir

	depot	somme	sommeFinale	nbAn	interet	Ecran	Expr. bool.
1		0					
2				0			
3						Montant du dépôt annuel	
4	***800.00***					800	
5						Montant de la somme souhaitée	
6			***2600.00***			2600	
7							Vrai
8		800.00					
9				1			
10					36		
11		836.00					
12							
7							Vrai
8		1636.00					
9				2			
10					73.62		
11		1709.62					
12							
7							Vrai
8		2509.62					
9				3			
10					112.93		
11		2622.55					
12							
7							Faux
13						Il faudra attendre : 3 années et vous recevrez 2622.55 euros…	

Chapitre 2

Instructions élémentaires

Définition Algorithme :
Suite d'actions à exécuter. Dans notre langage ces actions s'appellent des instructions.

Définition *Instruction* :
Evénement qui prend les variables dans un certain état (leurs valeurs) et les met dans un autre état (certaines valeurs ont été changées). Certaines de ces instructions sont simples ou élémentaires (elles s'écrivent sur une ligne), d'autres sont structurées (elles renferment d'autres instructions). On se propose d'examiner d'abord les instructions élémentaires.

1. Affectation

Rôle : instruction de MODIFICATION de la valeur d'une variable à partir d'une expression calculée.

Syntaxe :

< identificateur de variable > ← < expression >

Exemples :
```
interet ← (somme * TAUX) / 100
somme ←  somme + interet
```

< **expression** > c'est :
- une constante
- une variable
- le résultat d'un calcul fait à partir de constantes et de variables

Mécanisme :
1. Evaluation de l'expression qui se trouve à droite de la flèche
2. Rangement de ce résultat dans la variable dont l'identificateur est à gauche de la flèche

Etat des variables avant l'exécution de l'affectation

somme	interet
1636	73.62

```
somme ← somme + interet
```

Etat des variables après l'exécution de l'affectation

somme	interet
1709.62	73.62

Conséquence : s'il y avait une valeur dans la variable, elle est perdue (écrasée par la nouvelle valeur)

Important : le type du résultat de l'expression doit être compatible avec le type de la variable.

Exemple (en grisé les cas d'incompatibilité)

```
unEntier  : entier
unReel : réel
unCar  : caractère
```

`unEntier ← 5`	`unReel ← 5`	~~`unCar ← 5`~~
~~`unEntier ← 5.5`~~	`unReel ← 5.5`	~~`unCar ← 5.5`~~
~~`unEntier ← 'a'`~~	~~`unReel ← 'a'`~~	`unCar ← 'a'`
~~`unEntier ← '5'`~~	~~`unReel ← '5'`~~	`unCar ← '5'`

Tableau de compatibilité :

Type de l'expression / Type de la variable	Entier	Réel	Caractère
entier	Oui	Non	Non
réel	Oui	Oui	Non
caractère	Non	Non	Oui

2. Lecture

Rôle : instruction d'ENTREE qui permet l'interactivité avec l'utilisateur. Utilise un périphérique d'entrée.

Syntaxe :

lire(< identificateur de variable1>, [<identificateur de variable2>, …])

Exemple :

```
lire(depot)
```

Mécanisme :

1. Interruption du déroulement de l'algorithme
2. Attente d'une entrée de l'utilisateur
3. Rangement de la valeur entrée dans la variable dont l'identificateur suit le mot réservé lire

Conséquence : s'il y avait une valeur dans la variable, elle est perdue (écrasée par la nouvelle valeur)

Important : le type de la valeur entrée doit être compatible avec le type de la variable.

3. Écriture

Rôle : instruction de SORTIE qui permet l'interactivité avec l'utilisateur. Utilise un périphérique de sortie.

Syntaxe :

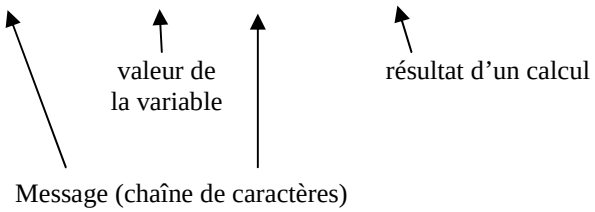
écrire(<expression_1>, [<expression_2>, …] >

Mécanisme :

Prend les expressions les unes après les autres et envoie leur valeur en sortie (écran).

Exemple :
soit x une variable qui contient la valeur entière 13. L'instruction

```
écrire("le successeur de ", x, " est ", x+1)
```



envoie à l'écran le résultat de 4 expressions ; on pourra lire : " le successeur de 13 est 14 ".

4. Retour sur l'exemple

On souhaite corriger un défaut : observation de la grille de simulation.

```
Tantque …..
   somme ← somme + depot          au 1er  Janvier
   …
   somme ← somme + interet        au 31 Décembre
```

Si on souhaite 2500 au lieu de 2600, on les a après le dépôt de 800 euros au début de la 3ème année. Ce n'est pas la peine d'attendre 3 ans. On modifie l'algorithme.

```
Début
…
     /* traitement */
7    tantque (somme < sommeFinale) faire
8       somme ← somme + depot            /* premier jour de l'année */
9       si (somme < sommeFinale) alors
10         nbAn ← nbAn + 1
11         interet ← (somme * TAUX) / 100
12         somme ← somme + interet       /* dernier jour */
13      finsi
14   fintantque
     /* affichage des résultats */
15   écrire("il faudra attendre : ", nbAn, " années et vous recevrez : ",
          somme, " euros")
Fin
```

	depot	somme	sommeFinale	nbAn	interet	Ecran	Expr bool
	…	…	…	…	…		
		1709.62	2500.00	2	73.72		
7							Vrai
8		2509.62					
9							
13							
14							
7							Faux
15						il faudra attendre 2 années et vous recevrez 2509.62 euros	

Chapitre 3

Expressions

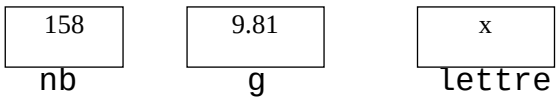
1. Définitions

Une expression simple est :

- une valeur (constante explicite)

13 21.73 ‘t’

- l’identificateur d’une variable (c'est à dire l'accès à sa valeur) : `nb, g` ou `lettre`



- la combinaison de valeurs et/ou identificateurs de variables et d'opérateurs (et d’appels de fonctions qui seront abordés ultérieurement :

`nb + 10` où `nb` et 10 sont les opérandes et + l’opérateur

Dans les expressions complexes, les opérandes peuvent être eux mêmes des expressions :

```
(nb + 10 ) * G
```

2. Type d’une expression

C’est le type de son résultat :

`nb + 10` est de type **entier**
`(nb + 10 ) * G` est de type **réel**
`‘lettre’` est de type **caractère**

Les expressions numériques ne posent pas de problèmes car les étudiants y sont habitués, mais l’extension de la notion de calcul à d’autres types que numériques est moins évidente.

3. Expressions numériques

Soit l’expression “ `a` **`op`** `b` ” où `a, b` sont les opérandes et **`op`** l’opérateur.

a	entier	réel
b		
entier	+ - * div mod → **entier** / → **réel**	+ - * / → **réel**
Réel	+ - * / → **réel**	+ - * / → **réel**

Les opérateurs *div* et *mod* sont ceux de la division euclidienne : `a = b * q + r` avec `r < b`

`q = a div b` **`r = a mod b`**

quotient entier dans la division entière de *a* par *b* reste dans la division entière de *a* par *b*

exemple : 32 div 5 vaut 6 et 32 mod 5 vaut 2

Priorité : les opérateurs {* / div mod,} sont prioritaires sur les opérateurs {+ -}

4. Expressions booléennes

- deux valeurs portées par les symboles *vrai* et *faux.*

- on peut déclarer des variables de type booléen.
  exemples :
  ```
  possible : booléen
  trouve : booléen
  possible ← vrai
  ```

Attention : *vrai* est un symbole et non pas un identificateur de variable

- Une expression booléenne peut être réduite à une variable booléenne
  exemple :
  ```
  possible ← trouve
  ```

- Une expression booléenne peut être construite à partir d'entiers et de réels (plus tard à partir de caractères et de chaînes) avec les opérateurs :
  `< <= > >= = <>`
  exemples :
  `46 > 100`
  est une expression booléenne dont la valeur est *faux*

  Si `x` est une variable réelle contenant la valeur 5.45,
  `x < 10`
  est une expression booléenne dont la valeur est *vrai*.

- Une expression booléenne peut aussi être construite à partir d'autres expressions booléennes et des opérateurs **:**
  `non , et, ou`

Tables d'opération de ces opérateurs (on dit aussi tables de vérités car ce sont des opérateurs booléens)

A	non A
vrai	*faux*
faux	*vrai*

A	B	A et B	A ou B
vrai	*vrai*	*vrai*	*vrai*
vrai	*faux*	*faux*	*vrai*
faux	*vrai*	*faux*	*vrai*
faux	*faux*	*faux*	*faux*

non (46 > 100) ➔ *vrai* 20 > 3 et 20 < 15 *vrai* et *faux* ➔ *faux* 20 > 3 ou 20 < 15 *vrai* ou *faux* ➔ *vrai*

Priorité : *non* est prioritaire sur *et* qui est prioritaire sur *ou*

Lois de De Morgan	non (A et B) ≡ non A ou non B	non (A ou B) ≡ non A et non B

On peut les démontrer en construisant des tables de vérité.

exemple : 3 notes sont rangées dans 3 variables *n1*, *n2*, *n3*. L'expression booléenne qui exprime l'admission à un module (moyenne arithmétique >= 10 et pas de note éliminatoire inférieure à 5)

```
(n1 + n2 + n3) / 3 >= 10 et non (n1 < 5 ou n2 < 5 ou n3 < 5)
```

est équivalente à :

```
(n1 + n2 + n3) / 3 >= 10 et (n1 >= 5 et n2 >= 5 et n3 >= 5)
```

- Les expressions booléennes peuvent être construites avec l'opérateur ∈ et un ensemble :

mois ∈ {1, 3, 5, 7, 8, 10, 12} exprime que l'entier rangé dans la variable *mois* représente un mois à 31 jours.

Remarques :

Les conditions qui contrôlent les instructions structurées sont des expressions booléennes. C'est leur valeur *vrai* ou *faux* qui détermine le fonctionnement de l'instruction.

Exemple d'utilisation dans un algorithme :

```
Algorithme : ExprBooleennesV1
Variables
   n1, n2, n3 : réel
Début
   lire(n1, n2, n3)
   si (n1 + n2 + n3) / 3 ≥ 10 et non (n1 < 5 ou n2 < 5 ou n3 < 5) alors
     écrire('admis')
   sinon
     écrire('refusé')
   finsi
Fin
```

ou encore

```
Algorithme : ExprBooleennesV2
Variables
   n1, n2, n3 : réel
   estAdmis : booléen
Début
   lire(n1, n2, n3)
   estAdmis ← (n1 + n2 + n3) / 3 ≥ 10 et non (n1 < 5 ou n2 < 5 ou n3 < 5)
   si estAdmis alors
     écrire('admis')
   sinon
     écrire('refusé')
   finsi
Fin
```

Dans le second exemple, l'expression booléenne qui suit le "si" est réduite à une variable booléenne. Au lieu de noter si estAdmis = vrai, on a utilisé si estAdmis, car les 2 expressions booléennes sont équivalentes.

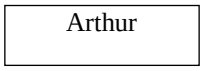
`a = vrai ≡ a` et `a = faux ≡ ! a`

Preuve :

a	a = vrai	! a	a = faux
vrai	vrai = vrai **vrai**	**faux**	vrai = faux **faux**
faux	faux = vrai **faux**	**vrai**	faux = faux **vrai**

5. Expressions caractères et chaînes de caractères

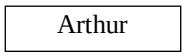
type **caractère** : un seul caractère
type **chaîne** : suite de caractères (éventuellement un seul)

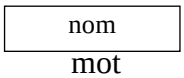


nom : chaîne de caractère
nom ← ‘Arthur
écrire(‘Son nom est : ‘, nom) ➔ Son nom est Arthur

Attention : Comme les identificateurs sont constitués de caractères, il y a ambiguïté entre un identificateur et une chaîne de caractères. Pour lever cette ambiguïté, toutes les valeurs de type caractère ou chaîne sont encadrées par des **séparateurs**.

`nom`	`‘nom’`
est l’identificateur d’une variable de type chaîne de caractères, représente une zone mémoire qu contient la valeur ‘Arthur’	est une valeur de type chaîne de caractères, valeur qui peut être rangée dans la variable *mot*.





Il existe un opérateur de construction : l’opérateur de **concaténation** qui met bout à bout deux caractères/chaînes

```
‘Bonjour’ + ‘Arthur’  ➔ ‘BonjourArthur’
‘Bonjour’ + ‘ Arthur’ ➔ ‘Bonjour Arthur’
‘51’+’200’            ➔ ‘51200’
51 + 200              ➔ 251
```

Toute expression résultat d’une concaténation entre caractères et/ou chaînes est de type chaîne.

Expressions booléennes construites à partir de caractères ou de chaînes
L’ordre est **alphabétique**. Il provient du fait que les caractères sont codés en respectant l’ordre alphabétique. Par contre, en machine, ce code ne respecte ni les accents, ni les majuscules.

`‘ELEPHANT’ < ‘SOURIS’` ➔ *vrai*
`‘100’ < ‘40’` ➔ *vrai* car ‘1’ est placé avant ‘4’
mais `100 < 40` ➔ *faux*

Chapitre 4

Instructions de choix

1. Instruction conditionnelle

Rôle : permet d'exécuter une séquence d'instructions plutôt qu'une autre en fonctions de données venant de l'utilisateur, ou de résultats calculés faits dans l'algorithme.

Syntaxe :

```
Si <condition> alors
    <séquence d'instructions>
finsi
```

<condition> est évaluée. Sa valeur peut être

- vrai ➔ la séquence est exécutée
- faux ➔ l'instruction est ignorée et on passe à l'instruction qui suit le fin si

Forme complète : instruction alternative :

```
Si <condition> alors
    <séquence d'instructions 1 >
sinon
    <séquence d'instructions 2 >
finsi
```

<condition> est évaluée. Sa valeur peut être

- vrai ➔ la séquence 1 est exécutée
- faux ➔ la séquence 2 est exécutée

Remarque : il faut que la condition soit **évaluable**, c'est à dire que les variables qui la composent éventuellement aient une valeur. Ci-dessous, une situation d'erreur

```
Début
  Si x > 5 alors
    …
  finsi
Fin
```

```
Algorithme maxdeux-1
/* demande deux nombres à l'utilisateur, calcule et affiche le plus grand des
deux. */
/* Les déclarations */
Variables
/* identificateur       : type              rôle */
   nb1, nb2             : entier            /* les deux nombres */
   maxi                 : entier            /* maximum des 2 nombres */

/* Les instructions */
Début
1 écrire('donner deux entiers ')
2 lire(nb1, nb2)
3 Si nb1 > nb2 alors
4    maxi ← nb1
  sinon
5    maxi ← nb2
6 finsi
  écrire('Le plus grand des deux
       nombres ', nb1, ' et ', nb2, 'est :
       ', maxi)
Fin
```

	nb1	nb2	maxi	Ecran
2	10	15		
3 : faux				
5			15	
6				
7 :				Le plus grand des deux nombres 10 et 15 est 15s

```
Algorithme maxdeux-2
/* demande deux nombre à l'utilisateur, calcule et affiche le plus grand des
deux. */
/* Les déclarations */
Variables
/* identificateur       : type              rôle */
   nb1, nb2             : entier            /* les deux nombres */
   maxi                 : entier            /* maximum des 2 nombres */

/* Les instructions */
Début
1 écrire('donner deux entiers ')
2 lire(nb1, nb2)
3 maxi ← nb1
4 Si nb2 > maxi alors
5    maxi ← nb2
6 finsi
7 écrire('Le plus grand des deux
       nombres ', nb1, ' et ', nb2,
       'est : ', maxi
Fin
```

	nb1	nb2	maxi	Ecran
2	10	15		
3			10	
4 : vrai				
5			15	
7 :				Le plus grand des deux nombres 10 et 15 est 15s

2. Instruction à choix multiple

Exemple : algorithme qui détermine le nombre de jours du mois entré par l'utilisateur.

```
Algorithme NombreDeJours

/* Les déclarations */
Variables
/* identificateur       : type            rôle */
   mois                 : entier          /* n° du mois dans l'année */
   an                   : entier          /* millésime de l'année */
   bis                  : booléen         /* indicateur d'année bissextile */
   nbJ                  : entier          /* nombre de jours par mois */

/* Les instructions */
Début
   lire(mois, an)
   bis ← (an mod 4 = 0 et an mod 100 <> 0) ou (an mod 400 = 0)
   Si mois ∈ {1, 3, 5, 7, 8, 10, 12} alors                        (*)
     nbJ ← 31
   sinon
     Si mois ∈ {4, 6, 9, 11} alors                                (*)
       nbJ ← 30
     sinon
       Si mois = 2 alors
         Si bis alors
           nbJ ← 29
         sinon
           nbJ ← 28
         finsi
       sinon
         nbJ ← 0
       finsi
     finsi
   finsi
   Si nbJ = 0 alors
     écrire('le nombre saisi pour le mois doit être compris entre 1 et 12')
   sinon
     écrire(nbJ, 'jours')
   finsi
Fin
```

(*) s'écrit encore :

```
mois = 1 ou mois = 3 ou mois = 5 ou mois = 7 ou mois = 8 ou mois = 10 ou mois = 12
```

On peut écrire un algorithme équivalent à la partie grisée avec l'instruction suivante :

Syntaxe :

selon <expression entière ou caractère> dans
<liste de valeurs n° 1 > : < séquence n°1 >
[<liste de valeurs n° 2 > : < séquence n°2 >
…
<liste de valeurs n° n > : < séquence n°n >
[autrement : <séquence autre>]]
finselon

la clause autrement est facultative

```
selon mois dans
  1, 3, 5, 7, 8, 10, 12   : nbJ ← 31
  4, 6, 9, 11             : nbJ ← 30
  2                       : Si bis alors
                              nbJ ← 29
                            sinon
                              nbJ ← 28
                            finsi
  autrement               : nbJ ← 0
finselon
```

Mécanisme :

- Evaluation de l'expression (le sélecteur), le plus souvent c'est une variable
- Recherche de cette valeur dans les listes, en parcourant celles-ci dans l'ordre où elles sont écrites
- Dès que la valeur est trouvée dans une liste
  - la séquence correspondante est exécutée
  - l'algorithme passe à l'instruction qui suit le fin selon
- Si la valeur du sélecteur n'est trouvée dans aucune liste, c'est la séquence qui correspond à *autrement* qui est exécutée

Remarques :

- les listes doivent être disjointes
- le sélecteur doit être d'un type énumérable, c'est à dire **entier** ou **caractère** (mais pas chaîne).

3. Instruction si … sinonSi … finsi

Dans l'exemple précédent, les différents cas possibles sont relatifs à un test d'égalité d'une variable par rapport à une valeur, ou à un test d'appartenance à un ensemble de valeurs. Examinons le cas d'un tarif dégressif suivant la quantité de produit commandée.

quantité	prix au kg
entre 1 et 5 kg (exclus)	100
entre 5 et 10 kg (exclus)	90
entre 10 et 30 kg (exclus)	80
entre 30 et 60 kg (exclus)	70
à partie de 60 kg	55

L'algorithme suivant répond à la question.

```
Algorithme Tarifs

/* Les déclarations */
Constantes
/* identificateur       = valeur          rôle */
   BORNE_1              = 5
   BORNE_2              = 10
   BORNE_3              = 30
   BORNE_4              = 60
   PRIX_1               = 100
   PRIX_2               = 90
   PRIX_3               = 80
   PRIX_4               = 70
   PRIX_5               = 55

Variables
/* identificateur       : type            rôle */
   quantite             : réel            /* quantité de produit en kg */
   prix                 : réel            /* prix au kg */

/* Les instructions */
Début
   lire(quantite)
   Si quantite > BORNE_4 alors
     prix ← PRIX_5
   sinon
     Si quantite > BORNE_3 alors
       prix ← PRIX_4
     sinon
       Si quantite > BORNE_2 alors
         prix ← PRIX_3
       sinon
         Si quantite > BORNE_1 alors
           prix ← PRIX_2
         sinon
           prix ← PRIX_1
         finsi
       finsi
     finsi
   finsi
   écrire(prix)
Fin
```

Il est visible que la mise en forme de ce type d'algorithme, où il y a beaucoup de si imbriqués et où les cas s'excluent mutuellement, peut être assez fastidieuse. On peut lui préférer la structure conditionnelle suivante :

```
/* Les instructions */
Début
   lire(quantite)
   Si quantite > BORNE_4 alors
     prix ← PRIX_5
   sinonSi quantite > BORNE_3 alors
     prix ← PRIX_4
   sinonSi quantite > BORNE_2 alors
     prix ← PRIX_3
   sinonSi quantite > BORNE_1 alors
     prix ← PRIX_2
   sinon
     prix ← PRIX_1
   finsi
   écrire(prix)
Fin
```

Syntaxe :

```
si <expression booléenne1> alors
    < séquence n°1 >
sinonSi <expression booléenne2> alors
    < séquence n°2 >
[sinonSi <expression booléenne3> alors
    < séquence n°3 >
 …
 sinonSi <expression booléenneN> alors
    < séquence n°N >]
 sinon
    < séquence n°N+1 >
finsi
```

Mécanisme :

- A la première expression booléenne vraie, la séquence correspondante est exécutée. et l'algorithme passe à l'instruction qui suit le fin selon
- Si aucune expression booléenne n'est vraie, la séquence correspondant à l'instruction sinon est exécutée.

Chapitre 5

Instructions répétitives

1. Instruction TANT QUE

Syntaxe :

```
Tantque <condition> faire
    < séquence d'instructions >
finTantque
```

Mécanisme
la première fois :

évaluation de <condition>

- vrai ➔ la séquence d'instructions est exécutée
- faux ➔ on n'entre pas dans la structure et c'est l'instruction qui suit *Fin tant que* qui est exécutée

les fois suivantes, on vient d'exécuter la séquence d'instructions. Il y a retour sur la condition pour la réévaluer

- vrai ➔ la séquence d'instructions est exécutée
- faux ➔ on n'entre pas dans la structure et c'est l'instruction qui suit *Fin tant que* qui est exécutée

Exemple : voir algorithme intérêts Chapître 1 page 4

Remarques :

- Il faut que la condition soit **évaluable** au moment où elle est évaluée
- La séquence peut ne jamais être exécutée si la condition est fausse
- Il faut que la condition soit **modifiée** par la séquence d'instructions, sinon soit on n'entre jamais, soit la boucle est éternelle

exemple :

```
Début
1 a ← 1
2 b ← a
3 tantque a < 100 faire
4    b ← b * 5
5    écrire(b)
6 fintantque
Fin
```

	a	b	Ecran
1	1		
2		1	
3			
4		5	
5			5
3			
4		25	
5			25…

a < 100 toujours vrai donc boucle éternelle

Résumé

Instructions simples : agissent directement sur les variables.

	affectation	modifient certaines variables
interactivité avec utilisateur	**écriture**	
	lecture	

Instructions structurées : contrôlent l'exécution des instructions simples en fonction de l'état des variables à chaque instant du déroulement de l'algorithme.

Si <condition> **alors** <séquence_1> **sinon** <séquence_2> **finsi**	instruction conditionnelle
Tantque <condition> **faire** <séquence> **fintantque**	instruction répétitive

D'autres instructions structurées seront présentées dans la suite.

2. Instruction Répéter

L'instruction *Tantque* était contrôlée par une expression booléenne d'entrée.
On dispose d'une autre répétitive contrôlée par une expression booléenne de sortie

`tantque somme < sommeFinale faire`
`  < séquence >`
`fintantque`
	`répéter`
	`  < séquence >`
	`jusqu'à somme >= sommeFinale`

Syntaxe :

répéter
< séquence >
jusqu'à <expression booléenne>

Mécanisme :

- éxécution de la séquence une première fois
- évaluation de <expression booléenne>
  - *vrai* ➔ sortie et exécution de la suite de l'algorithme
  - *faux* ➔ retour pour exécuter à nouveau la séquence et tester l'expression booléenne.

exemple :

```
Algorithme Exemple

/* Lexique */
Variables
   nbCoups, d1, d2 : entier
   …

Début
   …
   nbCoups ← 0
   répéter
      lire(d1)
      lire(d2)
      nbCoups ← nbCoups + 1
   jusqu'à d1 + d2 = 12
   …
Fin
```

Remarques :

- la séquence doit modifier l'expression booléenne ou condition de sortie, sinon la boucle est éternelle
- la séquence est exécutée au moins une fois

3. Instruction Pour

Ici la répétitive est contrôlée par le comptage du nombre de répétitions de la séquence.

Syntaxe :

❶
pour <variable> de <expr_début> à <expr_fin> faire
    <séquence>
    ❷
finpour

Mécanisme :

pour i **de** debut **à** fin **faire** <séquence> **finpour**	≡	i ← debut **tantque** i <= fin **faire** <séquence> i ← i + 1 **fintantque**

au repère ❶

- la variable de boucle prend la valeur de <expr_début>
- l'expression booléenne **<variable> ≤ <expr_fin>** est évaluée
  - *vrai* ➔ la séquence est exécutée
  - *faux* ➔ exécution de l'instruction qui suit le fin pour

au repère ❷

- la variable est incrémentée
- il y a retour pour évaluer l'expression booléenne **<variable> ≤ <expr_fin>**
  - *vrai* ➔ la séquence est exécutée
  - *faux* ➔ exécution de l'instruction qui suit le fin pour

Simulation d'un exemple

```
1    n ← 0
2    s ← 0
3    pour i de 1 à 3 faire
4       n ← n + 5
5       s ← s + n
6    finpour
7    écrire (s)
```

	n	n	i	Expr. bool.	Ecran
1	0				
2		0			
3			1	1 ≤ 3 : *vrai*	
4	5				
5		5			
6			2		
3				2 ≤ 3 : *vrai*	
4	10				
5		15			
6			3		
3				3 ≤ 3 : *vrai*	
4	15				
5		30			
6			4		
3				4 ≤ 3 : faux	
7					30

Remarques :

- La séquence ne doit modifier :
  - ni la variable de boucle
  - ni l'expression de fin
- la séquence peut ne jamais être exécutée si, avant d'aborder l'instruction pour, <expr_début> est déjà strictement supérieure à <expr_fin>
- Il existe une instruction pour par pas décroissant :

Simulation d'un exemple

```
1    n ← 0
2    s ← 0
3    pour i de 1 à 3 faire
4       n ← n + 5
5       s ← s + n
6    finpour
7    écrire (s)
```

	n	n	i	Expr. bool.	Ecran
1	0				
2		0			
3			1	1 ≤ 3 : *vrai*	
4	5				
5		5			
6			2		
3				2 ≤ 3 : *vrai*	
4	10				
5		15			
6			3		
3				3 ≤ 3 : *vrai*	
4	15				
5		30			
6			4		
3				4 ≤ 3 : faux	
7					30

Remarques :

- La séquence ne doit modifier :
  - ni la variable de boucle
  - ni l'expression de fin
- la séquence peut ne jamais être exécutée si, avant d'aborder l'instruction pour, <expr_début> est déjà strictement supérieure à <expr_fin>
- Il existe une instruction pour par pas décroissant :

pour i **de** debut **à** fin **par pas de –1 faire** <séquence> **fi pour**	≡	i ← debut **tant que** i >= fin **faire** <séquence> i ← i - 1 **fintantque**

4. Récapitulatif

- Instructions simples
  - affectation
  - lecture
  - écriture

- Instructions structurées

  - instructions de choix
    - si alors sinon finsi
    - selon finselon
    - si alors sinonsi alors sinon finsi

  - instructions itératives
    - tant que - fin tant que : contrôle à l'entrée
    - répéter - jusqu'à : contrôle à la sortie
    - pour - fin pour : contrôle sur le nombre de répétitions

Remarque 1 :

Comment choisir la structure itérative adaptée ?

Nombre d'itérations connu avant l'exécution de la 1ère itération et non modifiable.	Nombre d'itérations non connu avant l'exécution de la 1ère itération.	
	0 itération possible	au moins 1 itération
Pour	**tantque**	**répéter**

Remarque 2 :

Voici 2 types d'algorithmes itératifs qui traitent des listes et qui se distinguent suivant que l'on doive traiter le dernier élément de la liste ou non. Les exemples choisis pour illustrer ces algorithmes abordent des circonstances très concrètes.

Exemple 1 :

A la caisse d'un supermarché, la caissière doit traiter tous les clients dans la file d'attente.

Exemple 2 :

A la caisse d'un supermarché, la caissière doit traiter (comptabiliser) tous les articles d'un client qui se trouvent sur le tapis roulant, sauf le dernier élément qui se trouve être la barre de séparation avec le client suivant.

Pour peu que l'on dispose de fonctions permettant d'obtenir l'élément suivant et de tester si cet élément est le dernier de la liste, voici le principe de ces 2 algorithmes.

```
/* exemple 1 */
/* tous les éléments sont traités */

répéter
   element = elementSuivant()
   traiter(element)
jusqu'à dernier(element)
```

```
/* exemple 2 */
/* tous les éléments sont traités sauf le dernier */

element = elementSuivant()
tantque Non dernier(element)
   traiter(element)
   element = elementSuivant()
finTantque
```

Chapitre 7

Fonctions

1. Appel de fonctions prédéfinies

Les algorithmes font souvent appel à des fonctions prédéfinies :

- **fonctions mathématiques** : `racine`, fonctions trigonométriques, etc.
  exemple : pour calculer l'hypoténuse d'un triangle rectangle

```
lire(a, b)
c ← racine((a * a) + (b * b))
```

- **fonction sur les chaînes de caractères** : `longueur, sous_chaine`
  exemple : pour afficher les trois premiers caractères d'une chaîne

```
lire(chaine)
si nbcar >= 3 alors
  écrire ('Début : ', sous_chaine(chaine, 1, 3))
finsi
```

Dans un langage de programmation, les *fonctions prédéfinies* constituent une véritable boîte à outils, souvent organisée en *bibliothèques de fonctions* qui regroupent les fonctions par domaine (math, chaînes de caractères, système, etc.).

Si de nouvelles fonctions sont nécessaires, il est possible de les définir, c'est à dire de leur donner un identificateur et de programmer les calculs qui conduisent à ce qu'elles doivent renvoyer.

2. Rôle d'une fonction

Une fonction prend des données par l'intermédiaire de ses *paramètres*[1], et renvoie un résultat conforme à sa *spécification*, à condition que les conditions d'appels soient respectées. Par exemple un algorithme ne doit pas appeler une fonction `racine` avec un paramètre de valeur négative, si cela arrive, c'est l'algorithme qui est faux, pas la fonction.

3. Exemple

Les deux algorithmes `Maxquatre` calculent le maximum de quatre nombres entiers.

Dans le premier algorithme, la même instruction conditionnelle est répétée trois fois, aux noms des variables près

Dans le deuxième algorithme, les trois instructions conditionnelles analogues sont remplacées par des appels à une même fonction qui s'applique à des valeurs différentes. Ceci rend plus compréhensible ce que fait l'algorithme en évitant de dupliquer des instructions identiques.

[1] définis plus loin

```
Algorithme Maxquatre_1
/* demande deux nombre à l'utilisateur, calcule et affiche le plus
grand des deux. */

Variables
  a, b, c, d  : entier  /* nombres d'étude */
  mab, mcd    : entier  /* maxima intermédiaires */
  maxi        : entier  /* maximum des 4 nombres */
Début
  lire(a, b, c, d)
  si a > b alors
    mab ← a
  sinon
    mab ← b
  finsi
  si c > d alors
    mab ← c
  sinon
    mab ← d
  finsi
  si mab > mcd alors
    maxi ← mab
  sinon
    maxi ← mcdb
  finsi
  écrire(maxi)
Fin

Algorithme Maxquatre_2
/* idem avec appels de fonctions */
Variables
  idem ci-dessus
Début
  lire(a, b, c, d)
  mab ← maxdeux(a, b)
  mcd ← maxdeux(c, d)
  maxi ← maxdeux(mab, mcd)
  écrire(maxi)
Fin
```

Définition de la fonction maxdeux

```
fonction maxdeux(x, y : entier) : entier
/* renvoie le maximum des deux entiers passés en paramètres) */
Variables locales
  mxy   : entier    /* maximum de x et de y */
Début
  si x > y alors
    mxy ← x
  sinon
    mxy ← y
  finsi
  retourner(mxy)
Fin
```

4. Définition, syntaxe et mécanisme de passage de paramètres

Une fonction est un bloc d'instructions

- qui porte un nom, son *identificateur*
- qui prend des valeurs en entrée par l'intermédiaire de *paramètres*
- qui calcule et renvoie un résultat conforme à sa spécification et aux conditions de son appel

Le *profil* (ou *signature*) d'une fonction est constitué de la liste de types des valeurs entrées et du type de la valeur renvoyée en résultat

liste des types des valeurs d'entrée → type du résultat

exemples :

```
maxdeux : (entier, entier) → entier
sous-chaine : (chaine de caractères, entier, entier) → chaine de caractères
```

4.1. Définition de la fonction

Une fonction peut être définie n'importe où, à l'extérieur d'un algorithme. La définition de la fonction se compose de :

- son **en-tête** qui comprend :
- son identificateur
- la liste de ses **paramètres formels** et de leur type
- le type de la valeur qu'elle renvoie
- des déclarations locales (constantes ou variables)
- les instructions qui calculent son résultat
- au moins une instruction **retourner** qui renvoie la valeur résultat

```
fonction <ident-fonction>
                (<ident-paramètre> : <type> <role>
                     ...
                 <ident-paramètre> : <type> <role>) : <type>
/* spécification et conditions d'appels */
Variables locales
 <ident-variable> : <type>    <rôle>
 ....
 <ident-variable> : <type>    <rôle>

Début
 |     <instructions>
 |     retourner <expression>
Fin
```

NB : `<role>` correspond à du commentaire explicitant le rôle d'un paramètre ou d'une variable.

Attention : Il est indispensable de spécifier chaque fonction, et de préciser les conditions dans lesquelles elle doit être appelée. En effet si le programmeur qui intègre une fonction à un de ses programmes ne respecte pas les conditions d'appels, il n'a aucune garantie sur ce que fera la fonction dans son programme.

4.2. Appel de la fonction

La fonction est appelée depuis un algorithme principal, ou depuis une autre fonction (ou procédure[2]), dans l'expression où le résultat qu'elle renvoie doit être utilisé.

[2] Voir plus loin.

```
<ident-fonction>(<ident-paramètre>, ..., <ident-paramètre>)
```

4.3. Passage de paramètres

Dans la définition de la fonction : ***paramètres formels***

Dans l'appel de la fonction : ***paramètres effectifs*** (ou réel[3])

Il doit y avoir une correspondance **biunivoque** entre les *paramètres formels* et les *paramètres effectifs*, la correspondance étant régie par l'**ordre d'écriture**.

À l'appel de la fonction, le *premier paramètre effectif* passe sa valeur au *premier paramètre formel*, le *deuxième paramètre effectif* passe sa valeur au *deuxième paramètre formel*, et ainsi de suite. La fonction exécute son code puis renvoie son résultat.

Les paramètres qui se correspondent doivent avoir des types compatibles.

Appel	Définition

Début
...
  mab ← maxdeux (17, 10)
...
Fin
 	
17
	10
	17
	fonction maxdeux(x, y : entier) : entier
	variables
	  mx : entier
	Début
	  si x > y alors
	  mx ← x
	  sinon
	  mx ← y
	  finsi
	  retourner(mx)
	Fin

5. Intérêt des fonctions

- Ne pas dupliquer du code
- Offrir une meilleure lisibilité car le lecteur peut comprendre ce que fait une fonction, uniquement à la lecture de son nom, sans avoir à déchiffrer du code.
- Partager le développement entre différentes équipes qui se spécialisent.
- Construire des bibliothèques de fonction pour réutiliser ce qui a déjà été fait.

[3] Attention rien à voir avec le type des paramètres !

Chapitre 8

Procédures

1. Introduction

Comme les fonctions, les procédures sont des blocs d'instructions référencés par un nom et qui répondent à une spécification précise. Cependant elles ont des rôles différents :

- Une **fonction** renvoie une valeur qu'elle a calculé à partir des valeurs reçues en paramètres. L'appel d'une fonction se fait dans le même contexte qu'une **expression** au sein d'une instruction.
- Une **procédure** exécute des instructions qui utilisent des valeurs reçues en paramètres, mais qui peuvent aussi modifier l'environnement mémoire de l'unité qui l'appelle. L'appel d'une procédure se fait dans le même contexte qu'une **instruction**.
- Fonction et procédures sont désignées par le nom générique de **sous-programmes**.

2. Exemples

2.1. Procédure avec paramètres d'entrée

Supposons que l'on veuille afficher une suite de lignes comme le montre le schéma suivant :

```
*
++++
###
ooooooo
```

Au lieu de programmer chaque ligne l'une après l 'autre, on peut remarquer que chaque ligne est définie de manière non ambiguë dès que l'on connaît le caractère qui la compose et le nombre de caractères qu'elle comporte. En tenant compte de cette remarque, on peut définir une procédure dont le travail est d'afficher une ligne connaissant le caractère à afficher, ainsi que le nombre de caractères.

```
Algorithme dessin

Début
  ligne(1, '*')
  ligne(4, '+')
  ligne(3, '#')
  ligne(7, 'o')
Fin
```

```
procédure ligne
      (Entrée nb  : entier
       Entrée car : caractère)
Variables locales
  i : entier
Début
  Pour i de 1 à nb faire
    écrire(car)
  Finpour
  écrire(CRLF)    /* passage à la ligne */
Fin
```

2.2. Procédure avec paramètres d'entrée et paramètres de sortie

Cet algorithme demande deux angles exprimés en degrés et minutes d'angle, puis calcule la différence entre les deux. Avant de faire la différence, les deux angles sont convertis en minutes par la fonction `minute`. La fonction valAbs renvoie la valeur absolue du paramètre. Pour faire la conversion en sens inverse, on ne peut pas utiliser une fonction puisqu'il y a deux résultats (les degrés et les minutes). C'est pourquoi on utilise la procédure `convertir` qui a deux paramètres de sortie.

```
Algorithme rotation
Variables
  d1, m1, d2, m2  : entier
  mm1, mm2, dm    : entier
  deg, min : entier
Début
  lire(d1, m1)
  lire(d2, m2)
  mm1 ← minutes(d1, m1)
  mm2 ← minutes(d2, m2)
  dm ← valAbs(mm1 - mm2)
  convertir(dm, deg, min)
  écrire(deg, min)
Fin
```

```
fonction minutes(d, m : entier) : entier
Début
  retourner(60 * d + m)
Fin
```

```
fonction valAbs(x : entier) : entier
Variables locales
  abs : entier
Début
  si x > 0 alors
    abs ← x
  sinon
    abs ← -x
  finsi
  retourner(abs)
Fin
```

```
procédure convertir(Entree mi : entier
                    Sortie dd : entier
                    Sortie mm : entier)
Début
  dd ← mi div 60
  mm ← mi mod 60
Fin
```

2.3. Procédure avec paramètres d'entrée / sortie

Pour classer par ordre croissant trois lettres rangées dans trois variables c1, c2, et c3, on procède comme suit :

Variables :	**c1**	**c2**	**c3**
on compare le contenu de c1 et de c2 : dans l'ordre	T	Z	A
on compare et on échange le contenu de c2 et de c3	**T**	**Z**	A
on compare et on échange le contenu de c1 et de c2	T	**A**	**Z**
	A	**T**	Z

Les procédures classer et échanger sont susceptibles de modifier les valeurs des varaibles passées en paramètres (paramètres d'entrée/sortie).

```
Algorithme tri_lettres

Début
  lire(c1, c2, c3)
  classer(c1, c2)
  classer(c2, c3)
  classer(c1, c2)
  écrire(c1, c2, c3)
Fin
```

```
procédure classer(Entrée/Sortie a, b : caractère)
Début
  si a > b alors
    échanger(a, b)
  finsi
Fin
```

```
procédure échanger(Entrée/Sortie x, y : caractère)
Variables locales
  aux : caractère
Début
  aux ← x   ;     x ← y ;     y ← aux
Fin
```

3. Vocabulaire et syntaxe

Une procédure est un bloc d'instructions

- qui porte un nom, son *identificateur*
- qui communique avec l'unité appelante par l'intermédiaire des *paramètres*
- qui exécute ses instructions conformément à une spécification et aux conditions de son appel

Comme pour les fonctions il faut distinguer

- la définition de la procédure avec les *paramètres formels*,
- l'appel de la procédure avec les *paramètres effectifs*.

3.1. Définition de la procédure

Une procédure peut être définie n'importe où, à l'extérieur d'un algorithme. La définition de la procédure se compose de :

- son **en-tête** qui comprend :
- son identificateur
- la liste de ses **paramètres formels** et de leur type et de leur statut
- des déclarations locales (constantes ou variables) : les constantes ou variables locales ne sont connues que le temps de l'exécution de la procédure.
- les instructions qui calculent son résultat
- au moins une instruction **retourner** qui renvoie la valeur résultat

```
procédure <ident-procédure>
                (<statut> <ident-paramètre> : <type> <rôle>
                     ...
                <statut> <ident-paramètre> : <type> <rôle>)
/ spécification et conditions d'appels /
Variables locales
 <ident-variable> : <type>    <rôle>
 ....
 <ident-variable> : <type>    <rôle>

Début
      <instructions>
Fin
```

NB : `<role>` correspond à du commentaire explicitant le rôle d'un paramètre, ou d'une variable.

3.2. Appel de la procédure

La procédure est appelée depuis un algorithme principal, ou depuis une autre procédure, là où les instructions qu'elle renferme doivent être exécutées. Elle est appelée par son identificateur suivi de la liste des **paramètres effectifs** placés entre parenthèses..

```
<ident-procédure>(<ident-paramètre>, ..., <ident-paramètre>)
```

4. Paramètres

4.1. Règles générales

- Il doit y avoir une correspondance biunivoque entre les *paramètres effectifs* et les *paramètres formels.* La correspondance est assurée par l'**ordre d'écriture** dans les deux listes.
- Les types des paramètres qui se correspondent doivent être compatibles.

4.2. Trois statuts de paramètres

Paramètres d'entrée

Le paramètre effectif peut prendre la forme de n'importe quelle expression (constante, variable ou expression) dont la valeur est copiée dans le paramètre formel pour être **consultée** ou **utilisée** dans la procédure.

paramètre effectif		**paramètre formel**	**procédure action(info)**
une expression	►	`info`	`écrire (info)` `X ✱ info`

Paramètres de sortie

Le paramètre effectif est nécessairement un identificateur de variable qui reçoit une valeur produite par la procédure par l'intermédiaire du paramètre formel.

paramètre effectif		**paramètre formel**	**procédure action(info)**
un identificateur	◄	`info`	`lire (X)` `info ✱ X`

Paramètres d'entrée-sortie

Le paramètre effectif est un identificateur de variable qui contient une valeur. Celle-ci est envoyée à la procédure via le paramètre formel. La procédure modifie la valeur et la renvoie au paramètre effectif.

paramètre effectif		**paramètre formel**	**procédure action(info)**
un identificateur	► ◄	`info`	`info ✱ f(info)`

5. Intérêt des procédures et règle de bon usage

Les procédures sont les outils mêmes de la programmation modulaire. Elle permettent de fractionner les algorithmes (et par suite les programmes) en unités d'actions autonomes, plus intelligibles qu'un bloc d'instructions perdu au milieu d'un algorithme. Elles permettent aussi de partager le développement entre différentes personnes.

Toutes les informations qui circulent entre l'unité appelante et la procédure appelée doivent le faire par l'intermédiaire des paramètres.