Algorithmique et Programmation

par

Dr Ousmane THIARE

UFR DE SCIENCES APPLIQUEES ET DE TECHNOLOGIE
UNIVERSITÉ GASTON BERGER

Saint-Louis, Sénégal, 11 novembre 2008

TABLE DES MATIÈRES

LISTE DES TABLEAUX

LISTE DES FIGURES

CHAPITRE 1

Notions d'algorithme et de programme

1.1 Introduction à l'Algorithmique

L'algorithmique est une science très ancienne. Son nom vient d'un mathématicien arabe du IXème siècle EL KHAWRISMI. Des mathématiciens grecs comme Euclide ou Archimède en ont été les précurseurs (calcul du PGCD de 2 nombres, calcul du nombre π).

1.1.1 Qu'est-ce qu'un algorithme ?

Il existe plusieurs définitions possibles de l'Algorithme :

Définition 1 *Spécification d'un schéma de calcul, sous forme d'une suite finie d'opérations élémentaires obéissant à un enchaînement déterminé*

Définition 2 *Ensemble de règles opératoires dont l'application permet de résoudre un problème donné au moyen d'un nombre fini d'opérations*

Définition 3 *Etant donné un traitement à effectuer, un algorithme du traitement est l'énoncé d'une séquence d'actions primitives réalisant ce traitement*

1.1.2 Propriétés d'un algorithme

Tout algorithme décrit un traitement sur un nombre fini de données est la composition d'un nombre fini d'étapes, chaque étape étant formée d'un nombre fini d'opérations dont chacune est définie de façon rigoureuse et non ambiguë, effective, c'est-à-dire pouvant être effectivement réalisée par une machine Quelque soit la donnée sur laquelle il travaille, un algorithme doit toujours se terminer et fournir un résultat. Un algorithme est déterministe : étant donné un algorithme, toute exécution de celui-ci sur les mêmes données donne lieu à la même suite d'opérations et aboutit au même résultat. Il existe une relation étroite entre la notion de programme informatique et celle d'algorithme. Un programme informatique est écrit dans un langage de programmation et s'exécute sur un ordinateur (processeur, mémoire et organes d'Entrées-Sorties). En résumé, un algorithme doit être

$\Longrightarrow$ PRECIS : Il doit indiquer :
- l'ordre des étapes qui le constituent
- à quel moment il faut cesser une action
- à quel moment il faut en commencer une autre
- comment choisir entre différentes possibilités

$\Longrightarrow$ DETERMINISTE
- Une suite d'exécutions à partir des mêmes données doit produire des résultats identiques.

$\Longrightarrow$ FINI DANS LE TEMPS
- c'est-à-dire s'arrêter au bout d'un temps fini.

Exemple 4 *Résolution de l'équation du premier degré $A\ X + B = 0$*

```
Lire les coefficients A et B
Si A est non nul Alors
                    affecter à X la valeur - B / A
                    afficher à l'écran la valeur de X
                 Sinon
                    Si B est nul   Alors
                                       écrire "tout réel est solution"
                                   Sinon
                                       écrire "pas de solution"
```

1.1.3 Place de l'algorithme dans la résolution d'un problème informatique

Un algorithme doit être exprimé dans un langage de programmation pour être compris et exécuté par un ordinateur. Le programme constitue le codage d'un algorithme dans un langage de programmation donné, et qui peut être traité par une machine donnée. L'écriture d'un programme n'est qu'une étape dans le processus de programmation, comme le montre le schéma suivant :

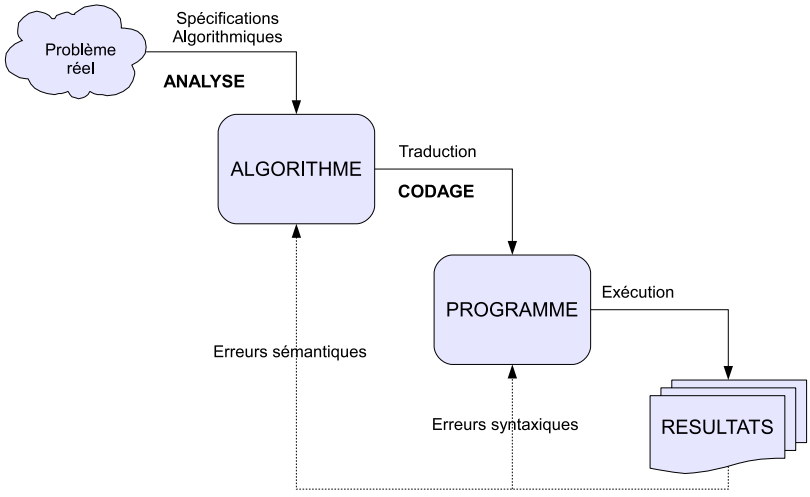


FIGURE 1.1 – Les différentes étapes du processus de programmation.

L'analyse du problème consiste à définir les différentes étapes de sa résolution. Elle permet de définir le contenu d'un programme en terme de données et d'actions. Le problème initial est décomposé en sous problèmes plus simples à résoudre auxquels on associe une spécification formelle ayant des conditions d'entées et le(s) résultat(s) que l'on souhaiterait obtenir. L'ensemble de ces spécifications représente l'algorithme. La phase suivante consiste à traduire l'algorithme dans un langage de programmation donné tout en respectant strictement la syntaxe du langage. Lors de l'étape d'exécution, soit des erreurs syntaxiques sont signalées, ce qui entraîne des corrections en général simples à effectuer, soit des erreurs sémantiques plus difficiles à déceler. Dans le cas d'erreurs

syntaxiques les retours vers le programme peuvent être fréquents. Dans le cas d'erreurs sémantiques, le programme produit des résultats qui ne correspondent pas à ceux escomptés : les retours vers l'analyse (l'algorithme) sont alors inévitables.

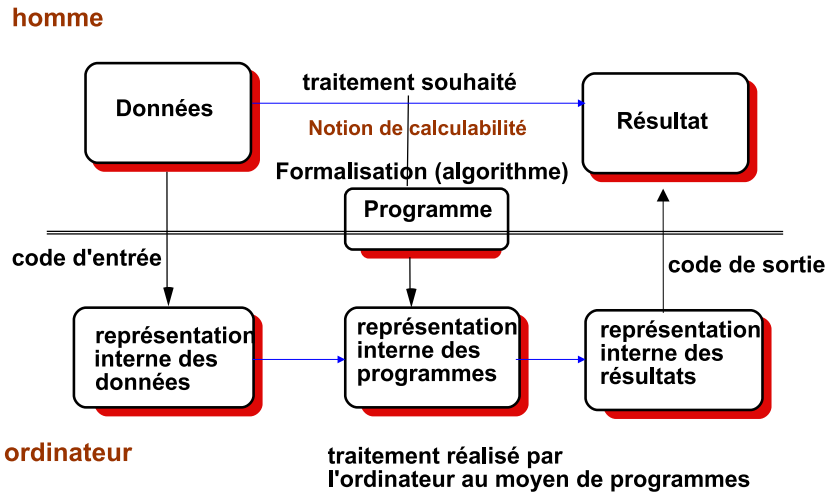


FIGURE 1.2 – Traitement des données.

1.2 Elaboration d'un algorithme

On peut distinguer quatre phase principales dans l'élaboration d'un algorithme.

- Analyse du problème
- Expression d'une solution en langage courant
- Expression d'une solution en pseudo-langage
- Tests et Vérification de l'adéquation de la solution

Analyse du problème :L'analyse consiste à bien comprendre l'énoncé du problème : Il est inutile et dangereux de passer à la phase suivante si vous n'avez pas bien discerné le problème.

Expression du raisonnement : Bien souvent, quelques lignes écrites en langage courant suffisent pour décrire succinctement l'essentiel du problème. L'intérêt de cette étape est qu'elle permet de vérifier rapidement que l'on se trouve sur la bonne voie. De plus, ces quelques lignes seront un support efficace lors de l'écriture de l'algorithme.

Expression d'une solution en pseudo-langage : Il peut arriver que plusieurs solutions répondent à un problème donné. Il faudra choisir la solution la plus judicieuse et rester cohérent jusqu'au bout.

Tests et Vérification de l'adéquation de la solution : Vérifier l'exactitude du comportement de l'algorithme, son bon déroulement. Si l'algorithme ne répond pas parfaitement à toutes les requêtes exprimées dans l'énoncé du problème, retournez à la phase n°1.

1.3 Programmation

Il existe différents logiciels réalisés par d'autres programmeurs et prêts à l'emploi :
- jeux,
- Bureautique (traitements de textes, tableurs...),
- de gestion (fichiers, bases de données...),
- retouche de son ou d'images...

Cependant, il arrive fréquemment que l'on doive concevoir soi-même un logiciel, pour une tâche spécifique, inédite. Il est alors indispensable de programmer. On utilise dans ce cas un langage de programmation.

De nos jours, il existe de nombreux langages informatiques, dédiés à la programmation. Les plus connus sont FORTRAN, COBOL, BASIC, C, ADA, C++, LISP, PROLOG, JAVA... Dans le cadre de ce cours, nous utiliserons le langage PASCAL, très pédagogique, et bien adapté pour les débutants, car il donne de bonnes bases en programmation structurée.

1.4 Objets simples, types et actions élémentaires

Un algorithme manipule des objets au sens informatique. Ces objets pourront être des données qui seront fournies en entrée, des résultats produits par l'algorithme ou des outils nécessaires au bon déroulement de l'algorithme.

1.4.1 Type d'un objet

En mathématiques, lorsque l'on utilise des objets, on précise leur type.

Exemple 5 $x \in \mathbb{R}, \quad i \in \mathbb{Z}$

x et i appartiennent à des types dont on connaît les propriétés

En informatique, les objets manipulés par un algorithme doivent appartenir à un type connu au moment de leur utilisation. Tout objet peut être caractérisé par un type qui indique :

- les ensembles de valeurs que peut prendre l'objet.
- les actions autorisées sur cet objet.

Les objets simples sont :

- des nombres (par exemple 3 , 7 , 3.141592 , 1996).
- des caractères ou des chaînes de caractères (par exemple : 'A' , '45' , 'BONJOUR').
- des valeurs booléennes (VRAI , FAUX)

On distingue généralement :

- les types scalaires qui sont par définition totalement ordonnés
- les types structurés qui regroupent sous un même nom une collection d'objets élémentaires qui peuvent être de même type (type homogène) ou de type différents (type hétérogène). Ces types structurés seront vus ultérieurement.

Les différents types simples sont les suivants :

- **Type entier** : prend ses valeurs dans un sous-ensemble des entiers relatifs. C'est un ensemble fini dans lequel chaque élément possède un successeur et un prédécesseur.
- **Type réel** : prend ses valeurs dans un sous-ensemble de réels décimaux signés. Dans la plupart des langages, cet ensemble n'est pas un ensemble fini. On ne peut trouver de successeur ou de prédécesseur à un réel donné.
- **Type caractère** : prend ses valeurs dans l'ensemble des caractères de la table ASCII.
- **Type chaîne de caractère** : se compose d'une suite de symboles de type caractère
- **Type booléen** : type logique qui peut prendre les valeurs VRAI ou FAUX.

En résumé,
objet $\Longrightarrow$ donnée ou algorithme
algorithme $\Longrightarrow$ procédure ou fonction
donnée $\Longrightarrow$ constante ou variable
constante ou variable $\Longrightarrow$ type scalaire ou type structuré
type structuré $\Longrightarrow$ type homogène ou hétérogène

1.4.2 Les objets

Pour désigner ces différents objets, on utilisera des chaînes de caractères qui seront appelées les identificateurs des objets. Pour différentier un identificateur d'un nombre, un identificateur commence par une lettre et ne comporte pas d'espace. De plus, on essaiera toujours de choisir des noms explicites afin de faciliter la relecture et l'éventuelle maintenance de l'algorithme par un tiers.

Un objet peut être :

– une constante ou une variable s'il s'agit d'une donnée

– au sens large, une fonction (ou une procédure) s'il s'agit d'un algorithme

1.4.3 Actions élémentaires

Les actions élémentaires sur une donnée dépendent évidemment du type de cette donnée et de sa catégorie (variable ou constante).

1.4.3.1 Opérateurs sur les types simples

Opérateur	**Notation**	**Type des opérandes**	**Type du résultat**
+ et - unaires	+ -	entier ou réel	celui de l'opérande
négation logique	NON	booléen	booléen
Puissance	↑	entier ou réel	entier ou réel
Multiplication	*	entier ou réel	entier ou réel
Division entière	DIV	entier	entier
Division	/	entier ou réel	réel
Reste(modulo)	MOD	entier	entier
Addition	+	entier ou réel	entier ou réel
Soustraction	-	entier ou réel	entier ou réel
Comparaison	$<$, $\leq$, $>$, $\geq$, $<>$, $=$	tout type	booléen
ET logique	ET	booléen	booléen
OU logique	OU	booléen	booléen

Tableau 1.1 – Opérateurs sur les types simples

Dans le tableau 1.1, les opérateurs sont classés par ordre de priorité décroissante mais attention dans l'utilisation des priorités car il existe souvent des différences d'un langage de programmation à l'autre. En absence de parenthèses, l'évaluation se fait de gauche à droite.

1.4.3.2 Affectation

L'affectation a pour rôle d'attribuer une valeur, résultat d'une évaluation, à un objet. La valeur doit être compatible avec le type de la valeur à gauche de l'affectation. Le symbole utilisé pour l'affectation est ← ou :=

Exemple 6

```
X <-- 1       ( X prend la valeur 1 )
X <-- 2*3+5   ( X prend la valeur du résultat de l'opération 2*3+5)
D <-- D+1     ( D augmente de 1)
prix_total <- nb_kg * prix_du_kg  (Si nb_kg est de type entier et
                                   prix_du_kg est de type  réel alors
                                   prix_total doit être de type réel)
```

1.4.3.3 Lecture et écriture

Ces actions permettent d'assurer l'interface entre l'environnement externe (l'utilisateur) et l'algorithme.

```
Lire(Valeur1, Valeur2 ...);
Ecrire(Valeur3, Valeur4 ...);
Ecrire ('Le résultat du calcul est : ',prix_total);
```

Exemple 7

```
ALGORITHME Epicier;
VAR prix_total , prix_du_kg : REEL;
    nb_kg : ENTIER;
DEBUT
    Ecrire('Entrez le prix d'un kilogramme de choux : ');
    Lire(prix_du_kg);
    Ecrire ('Entrez le nombre de kilogramme de choux : ');
    Lire (nb_kg);
    prix_total <- prix_du_kg * nb_kg;
    Ecrire ('Le prix total de l''achat est :',prix_total);
FIN .
```

1.4.3.4 Commentaires

Afin d'améliorer la lisibilité d'un algorithme, on peut utiliser des commentaires. Un commentaire est une suite de caractères quelconques encadrée par les symboles (* et *). Cette suite de caractère ne sera pas exécutée. Elle sert seulement à documenter le programme pour le rendre lisible par un tiers.

Exemple 8 *(* ceci est un commentaire *)*

CHAPITRE 2

Le langage Pascal

2.1 Historique

Le langage de programmation Pascal a été conçu au début des années 70 par N. Wirth. Depuis l'utilisation de ce langage s'est développé dans les universités et la communauté scientifique. Son succès toujours croissant a montré qu'il s'agit du langage qui durant les années 80, a détrôné les langages tels que FORTRAN, les dérivés de ALGOL. Le Pascal est facile à enseigner et à apprendre. Il permet d'écrire des programmes très lisibles et structurés. Il dispose entre autres de facilités de manipulation de données.

2.2 Programmer en Pascal

- **Programme**

  Un programme est une suite d'instructions destinées à l'ordinateur. Or le langage le langage de l'ordinateur est un langage machine qui n'utilise que deux symboles 0 et 1. On utilise donc un lange de programmation, ici le langage Pascal permettant de produire des programmes lisibles et facilement modifiables. Ces programmes sont traduits en langage machine par un compilateur

- **Code source**

  Un programme pascal (prog par exemple) peut être écrit avec un simple éditeur de texte. Le programme ainsi réalisé est stocké sous forme de fichier avec l'extension

.pas (prog.pas).

- **Compilation et édition de liens**

  La version en langage machine d'un programme s'appelle aussi le code objet. L'éditeur de liens est un programme qui intègre, après la compilation du fichier source, le code machine de toutes les fonctions utilisées dans le programme et non définies à l'intérieur. A partir du code objet du programme l'éditeur de liens génère un fichier exécutable d'extension .exe(prog.exe). Le fichier exécutable renferme alors le programme qui peut être chargée dans la mémoire pour être exécuté.

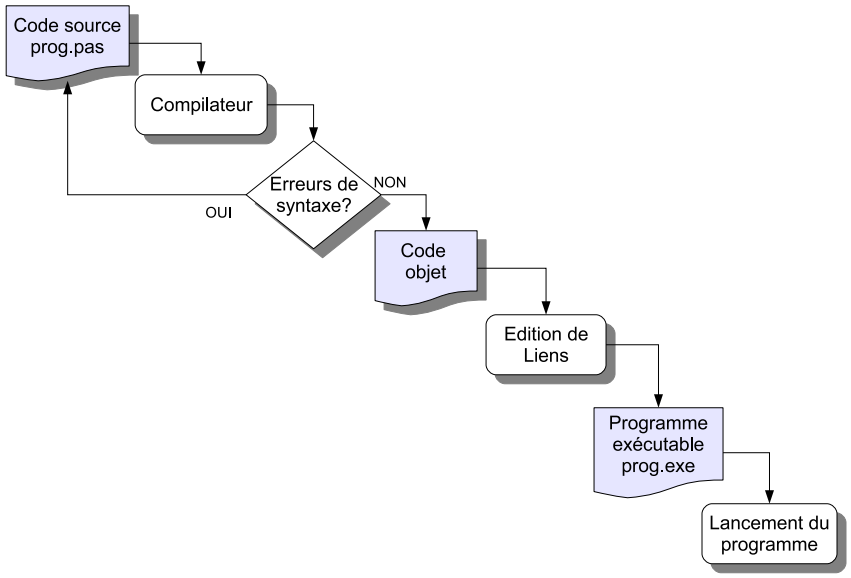


FIGURE 2.1 – Cycle de la programmation en Pascal.

2.3 Les constituants élémentaires du Pascal

2.3.1 L'Alphabet

L'alphabet Pascal est constitué des éléments suivants :

- Les majuscules : **A, B,..., Z** (26 caractères)
- Les minuscules : **a, b,..., z** (26 caractères)
- Le caractère « blanc »
- Les chiffres : **0, 1,..., 9**
- Les symboles spéciaux :

- Les opérateurs :
  - arithmétiques : + - * /
  - relationnels : <;<>; =; <=; >=; <>
- Les séparateurs : ( ); { }; [ ];(* *)
- Le signe« pointeur » : ^
- Les signes de ponctuation : . , ; : ' !?

Remarque **:**
Le Pascal n'est pas sensible à la casse. Il ne fait pas de distinction entre majuscule et minuscule. L'écriture begin est correcte ainsi que les écritures suivantes BEGIN ou Begin

2.3.2 Les mots du langage

Un mot est une suite de caractères encadrés par des espaces ou des caractères spéciaux.

2.3.2.1 Les mots réservés

- Ne peuvent être redéfinis par l'utilisateur
- Ont une signification précise
- Aident à la construction syntaxique des instructions.

Exemples de mots réservés :

```
AND     BEGIN   END    CASE    PROCEDURE
REPEAT  WHILE   VAR    TYPE    FUNCTION
CONST   FOR     GOTO   IF      UNTIL
WITH    TYPE    ARRAY  DO      ELSE    ... ... ...
```

2.3.2.2 Les identificateurs

Un identificateur est un nom donné à un élément du programme introduit par l'utilisateur (constante, variable, fonction, procédure, programme).

Remarque **:**
- Un identificateur est une suite alphanumérique commençant nécessairement par une lettre de l'alphabet et ne comportant pas d'espaces.
- Possibilité de lier plusieurs mots à l'aide de "_".
- Existence d'une limite au nombre de caractères. (dépend du compilateur)

• ***Exemples d'identificateurs légaux***

`TERME  terme  TRES_LONG_TERME  X23  Z1Z2`

- ***Exemples d'identificateurs non légaux*** :
  `3MOT  U.T.C.  MOT-BIS  A!8  $PROG  AUTRE MOT`

2.3.2.3 Les identificateurs standards

Exemples d'identificateurs standards :

– Fonctions :
`COS  SIN  EXP  SQR  SQRT  SUCC  PRED`

– Constantes :
`MAXINT  TRUE  FALSE`

– Types :
`INTEGER  REAL  BOOLEAN  CHAR`

– Procédures :
`READ  WRITE  NEW  RESET  REWRITE`

– Fichiers d'entrée-sortie :
`INPUT  OUTPUT`

2.4 Le Langage PASCAL

2.4.1 Caractéristiques globales

- **C'est un Langage Typé**
  - Toutes les variables doivent être pré-déclarées, ce qui évite les erreurs
  - Leur type doit être explicitement défini, ce qui enlève toute ambiguïté
  - Il s'agit d'un langage très 'pédagogique' et bien adapté pour les débutants
- **C'est un Langage Structuré**
  - Organisation du programme en "blocs d'instructions" emboîtés,
  - Utilisation d'identificateurs pour spécifier les blocs,
  - Utilisation d'indentations pour visualiser l'architecture du programme.
- **C'est un Langage Récursif**
  - Programmation concise et efficace

2.4.2 Structure globale d'un programme PASCAL

- EN-TETE
- DECLARATIONS
  1. UNITE
  2. ETIQUETTE
  3. CONSTANTES
  4. TYPES
  5. VARIABLES
  6. FONCTIONS / PROCEDURES
- BLOC D'INSTRUCTIONS EXECUTABLES

2.4.2.1 En-tête

Il s'agit de la première ligne d'un programme PASCAL. L'en-tête commence par le mot réservé PROGRAM suivi d'un identificateur, éventuellement suivi d'une liste de paramètres situés entre parenthèses. La ligne est terminée par un point-virgule.
SYNTAXE : `PROGRAM identificateur (id1,id2, ..., idn) ;`
Exemples d'en-tête :

```
PROGRAM  second_degre ;
PROGRAM  second_degre (input, output);
```

2.4.2.2 Partie déclarative

La zone de déclaration comprend :
- Déclarations d'unité avec `USES`
- Déclarations d'étiquette avec `LABEL`
- Déclarations de constante avec `CONST`
- Déclarations de type avec `TYPE`
- Déclarations de variables avec `VAR`

– Déclaration des fonctions et/ou de procédures respectivement avec `FUNCTION` et `PROCEDURE`

L'ordre indiqué doit être impérativement respecté. Les clauses indiquées sont optionnelles et dépendent des besoins du programmeur. Seuls l'entête et le corps du programme sont obligatoire.

2.4.2.3 Bloc d'instructions

Une instruction est une phrase du langage représentant un ordre ou un ensemble d'ordres qui doivent être exécutés par l'ordinateur. On distingue :

- Les instructions simples :
  - affectation, appels, branchement
- Les instructions structurées
  - instructions composées
  - instructions itératives
  - instructions conditionnelles

  Le bloc d'instructions principal commence par 'BEGIN' et se termine par 'END.'

Exemple 9 *Nous présentons ci-dessous un programme qui donne la moyenne de N nombres entrés au clavier par l'utilisateur, qui précisera le nombre de données qu'il va taper. A ce stade du cours, il n'est pas nécessaire de comprendre le contenu de ce programme. Il suffit simplement de reconnaître l'architecture globale décrite précédemment (déclarations de variables, blocs, indentations, begin...end).*

```
program MOYENNE ;                                    {En-tête}
USES crt;
var     DONNEE, SOMME, MOYENNE : real;
              I, N : integer ;                       {partie déclarative}
begin                                                {début du bloc d'instructions}
    clrsrc;
    writeln('entrer le nombre de données');
    readln(N);                                       {instruction}
    if N > 0 then
                begin
                    SOMME := 0;
                    for I := 1 to  N  do
```

```
                    begin
                       read(DONNEE);
                       SOMME := SOMME + DONNEE;
                    end;
                MOYENNE := SOMME / N;
                writeln('moyenne =',MOYENNE);
            end
        else
            writeln('pas de donnees');
end.                                          {Fin du bloc d'instructions}
```

2.4.3 Déclarations de constantes

En Pascal, tout symbole utilisé dans un programme doit être explicitement déclaré.

Une constante est désignée par un identificateur et une valeur, qui sont fixées en début de programme, entre les mots clés `CONST` et `VAR`. La valeur **ne peut pas être** modifiée, et **ne peut pas** être une expression.

Syntaxe

```
identificateur = valeur_constante;
```

ou

```
identificateur : type = valeur_constante;
```

Dans la première forme, le type est sous-entendu (si il y a un point, c'est un réel, sinon un entier ; si il y a des quotes, c'est un caractère (un seul) ou une chaîne de caractères (plusieurs).

L'utilisation de constantes en programmation est vivement conseillée. Elles permettent :

- une notation plus simple
  Exemple : `PI` à la place de `3,141592653` `CONST pourcent : real = 33.3;`
- La possibilité de modifier simplement la valeur spécifiée dans la déclaration au lieu d'en rechercher les occurrences, puis de modifier dans tout le programme.

Exemple 10 *Déclaration de constantes*

- *Déclarations de type numérique*

```
const    DEUX = 2;
         PI = 3.14;
```

- *Déclarations de type booléen*

```
VRAI = true;
FAUX = false;
```

- *Déclarations de type caractère*

```
CARA  = 'A';
```

- *Déclarations de type chaîne de caractères*

```
PHRASE = 'Vaut mieux une fin effroyable qu''un effroi sans fin';
```

2.4.4 Déclaration de types

Un type est un ensemble de valeurs que peut prendre une donnée.

2.4.4.1 Les types standards

Un type standard est un type qui est normalement connu de tout langage Pascal et qui n'a donc pas été déclaré par l'utilisateur. Les différents types standards :

Le type entier : `integer`

- Sur$n$ bits, il sera possible d'utiliser des entiers compris entre -2^{n-1} et $(2^{n-1}-1)$
- Donc sur 16 bits, les entiers seront compris entre -32768 et 32767 (-2^{15} et $+2^{15}-1$).
- Donc sur 32 bits, les entiers seront compris entre -2147483648 et $+2147483647$ (-2^{31} et $+2^{31}-1$).

- Opérateurs sur les entiers :

`abs(x)`	valeur absolue de x
`pred(x)`	$x - 1$
`succ(x)`	$x + 1$
`odd(x)`	`true` si x est impair, `false` sinon.
`sqr(x)`	le carrée de x.
`+ x`	identité.
`- x`	signe opposé
`x + y`	addition
`x - y`	soustraction.
`x * y`	multiplication.
`x / y`	division, fournissant un résultat de type réel.
`x div y`	dividende de la division entière de x par y.
`x mod y`	reste de la division entière, avec y non nul.

  Remarques
  - Attention, les opérateurs `/`, `div` et `mod`, produisent une erreur à l'exécution si y est nul.
  - Lorsqu'une valeur (ou un résultat intermédiaire) dépasse les bornes au cours de l'exécution, on a une erreur appelée débordement arithmétique.

Le type réel : `real`

- Utilisable pour représenter les rééls et les entiers élevés. Le domaine de définition dépend de la machine et du compilateur utilisés.
- 0.0 ; $-21.4E3$ $(= -21,4 \times 10^3 = -21400)$; $1.234E-2$ $(= 1,234 \times 10^{-2})$
- Opérateurs sur un argument x réel : `abs(x)`, `sqr(x)`, `+x`, `-x`. Si l'un au moins des 2 arguments est réel, le résultat est réel pour : `x - y`, `x + y`, `x * y`.
- Résultat réel que l'argument soit entier ou réel : `x / y` (y doit être non nul) ; fonctions `sin(x)`, `cos(x)`, `exp(x)`, `ln(x)`, `sqrt(x)` (*square root*, racine carrée).
- Fonctions prenant un argument réel et fournissant un résultat entier : `trunc(x)` (partie entière), `round(x)` (entier le plus proche). Si le résultat n'est pas représentable sur un `integer`, il y a débordement.

Le type booléen : `boolean`

- Définit les valeurs logiques
- Constantes : `true` (vrai) et `false` (faux)
- Ce type permet la manipulation avec des opérateurs logiques

- Opérateurs booléens : `not` (négation), `and` (et), `or` (ou).

x	y	not x	x and y	x or y
true	true	false	true	true
true	false	false	false	true
false	true	true	false	true
false	false	true	false	false

- Opérateurs de comparaison (entre 2 entiers, 2 réels, 1 entier et 1 réel, 2 chars, 2 booléens) : `<, >, <=, >=, =` (égalité, à ne pas confondre avec l'affectation `:=`), `<>` (différent).
  Le resultat d'une comparaison est un booléen. On peut comparer 2 booléens entre eux, avec la relation d'ordre `false < true`.
- En mémoire, les booléens sont codés sur 1 bit, avec 0 pour false et 1 pour true. De là les relations d'ordre. Les opérateurs booléens `not`, `and`, `or` s'apparentent approximativement à $(1-x)$, $\times$, $+$.

Le type caractère : `char`

- C'est un unique caractère entouré d'apostrophes
- Ensemble des valeurs de ce type :
  - alphanumériques : `'a' . . 'z'  'A' . . 'Z'  '0' . . '9'`
  - caractère blanc : ' '
  - Le choix et l'ordre des 256 caractères possible dépend de la machine et de la langue. Sur PC, on utilise le code ASCII, où `'A'` est codé par 65, `'B'` par 66, `'a'` par 97, ' ' par 32, `'{'` par 123, etc.
  - Les opérateurs sur les chars sont :

    `ord(c)` numéro d'ordre dans le codage ; ici ¡ code ascii ¿.

    `chr(a)` le résultat est le caractère dont le code ascii est a.

    `succ(c)` caractère suivant `c` dans l'ordre ascii , `chr(ord(c)+1)`

    `prec(c)` caractère précédent `c` dans l'ordre ascii.

  - On peut remplacer `chr(32)` par `#32`, mais pas `chr(i)` par `#i`.

Le type chaîne de caractère : `string` ou `varying`

- Ce type n'est pas disponible sur tous les compilateurs
- Possibilité d'accéder à un caractère particulier de la chaîne, en indiçant la variable qui y fait référence

Exemple 11

```
phrase  <---  ' il fait beau'
phrase[5] <--- 'a'
```

2.4.4.2 Les types scalaires et non standards

Principe : on fabrique les types dont on a besoin par l'intermédiaire du mot réservé `TYPE`. Les types **scalaires standard** sont les **entiers** et les **caractères**.

LE TYPE ÉNUMÉRÉ

Un type énuméré est une séquence ordonnée d'identificateurs.

SYNTAXE : `TYPE identificateur = (id1, id2,..., idn) ;`

Exemple 12
```
Type COULEUR  =  (jaune, vert, rouge, bleu, marron);
     SEMAINE=(lundi, mardi, mercredi, jeudi, vendredi, samedi, dimanche);
     SEXE =(masculin, féminin);
     VOYELLE = (A, E, I, O, U);
```

Le mot réservé `TYPE` ne doit être écrit qu'une seule fois.

Avec un type énuméré, on peut donc donner la liste des valeurs possibles les unes après les autres. Les autres types énumérés déjà définis sont les `boolean`, les `char`, mais *aussi* les `integer`. Tous ces types se comportent comme des entiers, et la déclaration qu'on a faite dans l'exemple précédent est traduite sous la forme `lundi = 0`, `mardi = 1` et ainsi de suite. L'avantage est qu'il existe des fonctions définies sur tous les types énumérés :

- la fonction `ORD`, qui donne le numéro dans la liste de déclaration (`ord(mardi)=1`, on commence à `0`) ;
- les fonctions `SUCC` et `PRED`, qui donnent le successeur et le prédecesseur dans la liste : `succ(mardi)= mercredi`, `pred(samedi)=vendredi`. Attention, `pred(lundi)` et `succ(dimanche)` ne sont pas définis. On ne sait pas ce qu'ils valent ;
- les opérateurs de comparaison
- les boucles for ;

Il faut faire attention, les fonctions `readln` et `writeln` ne marchent pas avec ces types.

D'autre part, deux types énumérés différents ne peuvent contenir le même identificateur. $\Longrightarrow$ Ensembles énumérés disjoints

LE TYPE INTERVALLE

Un type intervalle est un sous-type d'un type scalaire déjà défini.

SYNTAXE : `TYPE identificateur = [borne inf].. [borne sup] ;` Points importants :

- Valeurs autorisées : toutes celles de l'intervalle
- Deux sortes de type intervalle :
  - les types issus d'un type énuméré standard
  - les types issus d'un type énuméré déclaré.

Exemple 13 *Nous présentons ci-dessous des exemples issus d'un type standard.*

- *Intervalle d'entiers :*

```
Type DECIMAL =  0 .. 9   ;
     OCTAL   =  0 .. 7   ;
     AGE     =  0 .. 150 ;
```

- *Intervalle de caractères :*

```
Type ABC = 'A' .. 'C' ;
     MAJ = 'A' .. 'Z' ;
```

- *A présent, voici quelques exemples issus d'un type non-standard*

```
Type OUVRABLE = lundi  .. vendredi ;
     WEEK-END  = samedi .. dimanche ;
     LETTRES   = 'A' .. 'Z' ;
```

Ordre ascendant requis : borne-inf doit être placé avant borne-sup dans le type énuméré source.

Exemple 14 *Déclarations de type intervalle incorrectes :*

```
Type OCTAL    = 7 .. 0 ;
     OUVRABLE = vendredi .. lundi ;
```

Pas de type intervalle issu du type réel (non scalaire) !

2.4.5 Déclaration de variables

Une variable est une donnée manipulée par un programme et pouvant être modifiée. Elle peut être :

– une donnée d'entrée ;
– le résultat final d'un calcul ;
– un résultat intermédiaire de calcul.

Déclarer une variable, c'est définir l'ensemble des valeurs qu'elle peut prendre. Toutes les variables utilisées dans un programme doivent être déclarées. Une variable représente un objet d'un certain type ; cet objet est désignée par un identificateur. Toutes les variables doivent être déclarées après le `VAR`.

Deux façons pour déclarer une variable :
– à l'aide d'un type standard ou d'un type pré-déclaré
– par une déclaration explicite et spécifique à cette variable de l'ensemble des valeurs qu'elle peut prendre.

  SYNTAXE : `VAR identificateur : type ;`
– `VAR` ne peut apparaître qu'une seule fois
– Possibilité de grouper plusieurs variables pour le même type
– Séparation des variables par une virgule

Exemple 15 *Déclarations de variables*

– *avec référence à un type existant*

```
var    JOUR    :  semaine ;
       A, B, C :  real    ;
       I, J, K :  integer ;
       CONGE   :  week-end ;
       VIVANT  :  boolean ;
```

– *avec déclaration locale explicite :*

```
var  LETTRE :  'A'  . .  'Z' ;
     FEUX   :  (vert, orange, rouge) ;
```

Exemple 16 *Déclaration de constante, type et variable*

```
const  JOUR_MAX =  31 ;
       AN_MIN   =  1901 ;
       AN_MAX   =  2000 ;
type   SIECLE   =  AN_MIN  . .  AN_MAX ;
```

```
    SEMAINE  = (LUNDI, MARDI, MERCREDI,JEUDI,VENDREDI,SAMEDI, DIMANCHE) ;
    ANNEE    = (JANVIER, FEVRIER, MARS, AVRIL, MAI,JUIN, JUILLET,  AOUT,
                SEPTEMBLE, OCTOBRE , NOVEMBRE, DECEMBRE);
var  MOIS       : année ;
     JOUR       : semaine ;
     N_JOUR     : 1 .. jour_max ;
     AN         : siecle ;
     OUVRABLE   : lundi .. vendredi ;
     I, J       : integer ;
     N_ETUDIANT : 1 ..  maxint ;
```

2.4.6 Instructions composées

2.4.6.1 Définition

Les instructions composées permettent de regrouper, dans un même bloc, un ensemble d'instructions qui seront exécutées au même niveau.

Syntaxe : Séquence de deux ou plusieurs instructions comprises entre BEGIN et END et séparées par des points virgules

2.4.6.2 Instruction d'affectation

`<VARIABLE>:=<expression>`

– Evaluation de l'expression (calcul)

– Puis affectation (rangement) dans la variable (identificateur)

Nécessité d'avoir des types compatibles (les mélanges de types sont interdits) Ne pas confondre " :=", l'opérateur d'affectation et "=", l'opérateur de test.

Exemple 17 *Soit* `X` *une variable de type integer et on lui donne comme valeur* `10` `X := 10` *signifie que l'on affecte la valeur* `10` *à la variable* `X` *donc* `X` *vaut* `10`.
On peut tester si `X` *est égal à une certaine valeur avant d'effectuer un calcul :*

```
Si X = 3 alors X := X / 2
```

Ici `X` *vaut toujours 10 car le test* `X = 3` *n'est pas vérifié (puisque la valeur* `10` *a été placée dans* `X`*)*

A la déclaration, les variables ont une valeur indéterminée. On initialise les variables juste après le `BEGIN` pricncipal (on ne peut pas le faire dans la déclaration). Utiliser la

valeur d'une variable non initialisée est une erreur grave !

Exemple 18

```
VAR
a, b, c : integer;
BEGIN
    { Partie initialisation }
    b := 5;
    { Partie principale }
    a := b + c; { ERREUR, c n'a pas été initialisée' }
END.
```

L'opération `identificateur := expression;` est une affectation. On n'a pas le droit d'écrire `id1 := id2 := expr` , ni `expr := id` ni `expr1 := expr2` .

2.4.7 Opérateurs et Fonctions arithmétiques

2.4.7.1 Opérateurs disponibles

Opérateur	Description
+	somme
-	soustraction
*	multiplication
/	division
DIV	division entière (5 div 3 = 1)
MOD	modulo (5 mod 3 = 2)

TABLEAU 2.1 – Opérateurs disponibles

Exemple 19

```
var A, B, C, D : real;
    I, J, K : integer;
begin
   A := 7.4 ; B := 8.3 ;
   C := A + B ;
   D := A / B + C ;
   I  :=  42 ; J := 9 ;
   K := I  mod  J ;     { K vaut 6 }
end.
```

2.4.7.2 Expressions

Une expression désigne une valeur, exprimée par composition d'opérateurs appliquées à des opérandes, qui sont : des valeurs, des constantes, des variables, des appels de fonction ou des sous-expressions.

Exemple 20 *Etant donnée une variable* `x`*, une constante* `max` *et une fonction cos(), chaque ligne contient une expression :*

```
5
x + 3.14
2 * cos(x)
(x < max) or (cos(x-1) > 2 * (x+1))
2.08E3 * x
(x>2) OR (x<8)
```

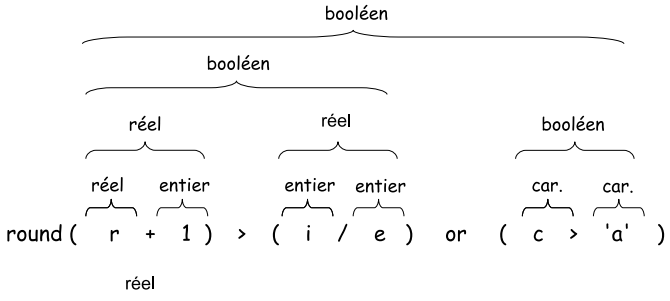
Type des expressions bien formées

Une expression doit être bien formée pour que l'on puisse trouver sa va- leur. Par exemple, `3 * 'a' - true` n'est pas bien formée, et la compilation Pascal échouera. L'expression bien formée a un type, qui dépend des règles d'évaluation de l'ex- pression.

Soit r un réel, i un entier, e une constante entière, `c` un caractère. L'expression

`(round(r+1) > (i/e)) or (c < 'a')`

est bien formée, et sont type est booléen comme on le montre ici :



Remarque Le fait qu'une expression soit bien formée n'implique pas que son évaluation est sans erreur, ce qui peut être le cas ici si `e` est nul.

Règles d'évaluation

L'expression `a + b * c` est évaluée `a + (b * c)` et non pas `(a + b) * c` : ceci parce que le `*` est prioritaire par rapport à `+`. On classe les différents opérateurs par ordre de priorité, les opérateurs de plus forte priorité étant réalisés avant ceux de plus faible priorité. Lorsque deux opérateurs sont de priorité égale, on évalue de gauche é droite. Par exemple `a + b - c` est évaluée `(a + b) - c`, et non pas `a + (b - c)`. Voici la table des priorités classées par ordre décroissant, les opérateurs sur une même ligne ayant une priorité égale.

`() fonction()`	primaire
`+ - not`	unaire
`* / div mod and`	multiplicatif
`+ - or`	additif
`= <> < <= >= >`	relation

Remarque Est-ce que l'expression `a < b or c <= d` est bien formée ? Quel est son type ?

Réponse : non ! Ecrire une telle expression booléenne sans parenthèses est une erreur classique. En effet dans la table de priorités, l'opérateur or a une priorité plus élevée que les opérateurs `<` et `<=`, et donc l'expression sera évaluée `a < (b or c) <= d` , ce qui est faux. L'expression bien formée est ici `(a < b) or (c <= d)` .

2.4.7.3 Fonctions arithmétiques

2.4.7.4 Fonctions logiques

2.5 Entrées / Sorties

Ce sont des échanges d'informations entre la mémoire (variables et constantes) et les périphériques (clavier, écran ...). Types autorisés : réel, booléen, caractères et chaînes de caractères.

Le clavier et l'écran sont gérées comme des fichiers particuliers : ce sont des fichiers texte, toujours ouverts et sans fin ; ils sont désignés par les variables prédéfinies `input` et `output` (dont on ne se sert quasiment jamais).

Pour transmettre des données saisies au clavier à un programme (entrées) ou pour afficher à l'écran les données par un programme (sorties), il faut faire appel à un ensemble

Fonction	Description
ABS (X)	valeur absolue de X
ARCTAN (X)	arctangente de X
CHR (X)	caractère dont le numéro d'ordre est X
COS (X)	cosinus de X
EXP (X)	exponentielle de X
LN (X)	logarithme népérien de X
ORD (X)	numéro d'ordre dans l'ensemble de X
PRED (X)	prédécesseur de X dans son ensemble
ROUND (X)	arrondi de X
SIN (X)	sinus de X
SQR (X)	carré de X
SQRT (X)	racine carrée de X
SUCC (X)	successeur de X dans son ensemble
TRUNC (X)	partie entière de X

TABLEAU 2.2 – Fonctions arithmétiques

Fonction	Description
EOF (X)	vrai si la fin de fichier X est atteinte
EOLN (X)	vrai si fin de ligne du fichier
ODD (X)	vrai si X est impair, faux sinon

TABLEAU 2.3 – Fonctions logiques

de fonctions appartenant à l'unité d'entrées/sorties. Il faut donc faire apparaître en début de programme l'instruction suivante :

```
USES crt ;
```

2.5.1 Sortie (Ecriture)

On utilise la fonction `write` ou `writeln` pour l"affichage formaté des données ; Formaté signifie que l'on contrôle la forme et le format des données La fonction admet la syntaxe suivante :

```
Write(argument1, argument2,..., argumentn)
```

ou

```
Writeln(argument1, argument2,..., argumentn)
```

avec `argument1,..., argumentn` : les arguments à afficher

Exemple 21 `write('bonjour ')`

```
writeln('monsieur')
a :=2+3 ;
writeln('la somme de 2 + 3 donne :',a) ;
```

La fonction `write` *écrit ici à l'écran les arguments (chaîne de caractères, constante, variable) La fonction* `writeln` *écrit la même chose. La seule différence est que à la fin de l'écriture du dernier argument, il y a un passage à la ligne suivante*

⟶ Trois écritures équivalentes :

```
writeln (a, b, c, d);
write (a, b, c, d); writeln;
write(a); write(b); write(c); write(d); writeln;
```

Le résultat de l'affichage dépend du type du paramètre :

Exemple 22

```
VAR e : integer; c : char; b : boolean; r : real; s : string[32];
BEGIN
e := 12; c := 'A'; b := true; r := 23.0; s := 'toto';
writeln (e, '|', c, '|', b, '|', r, '|', s);
END.
```

affiche : `12|A|TRUE|2.300000E+01|toto`

Formatage de l'inpression des variables

- Soit `v` un entier, un booléen, un caractère ou un string. `write(v:8)` dit à `write` d'afficher `v` sur au moins 8 caractères. Si le nombre de caractères (signe éventuel compris) est `> 8`, `v` est complétement affiché ; si il est `< 8`, des espaces sont rajoutés à gauche pour compléter.
  Ainsi `writeln (e:5, '|', c:3, '|', b:5, '|', s:6);` affiche :
  ␣␣␣`12|`␣␣`A|`␣`TRUE|`␣␣`toto`
- Soit `r` un réel.
  `write(r:10);` dit à `write` d'afficher `r` en notation scientifique, sur au moins `10` caractères, signes de la mantisse et de l'exposant compris. Cette fois c'est d'abord le nombre de chiffres après la virgule qui change de `1` à `10`, puis au besoin des espaces sont ajoutés à gauche. De plus le dernier chiffre de la mantisse affichée est arrondi.
  ```
  r := 2 / 3;
  writeln (r:8, '|', r:10, '|', r:18 );
  ```

affiche :

␣6.7E-01|␣6.667E-01|␣␣6.6666666667E-01

– Autre formattage de `r` réel. `write(r:8:4);` dit à `write` d'afficher `r` en notation simple, sur au moins `8` caractères, dont 4 chffres après la virgule (le dernier étant arrondi).

Ainsi `writeln (r:8:4);` affiche :

␣␣␣0.6667

2.5.2 Entrées (Lecture)

On utilise la fonction `readln` pour la saisie des données depuis le clavier La fonction admet la syntaxe suivante :

`readln(argument1, argument2,..., argumentn)`

avec `argument1,..., argumentn` : les arguments. Le programme va lire ce que l'utilisateur a tapé et va stocker les valeurs dans les variables `argument1,..., argumentn`

Exemple 23

```
Write('Entrez un nombre entier : ') ;
Readln(a) ;
Writeln('vous avez entré la nombre ',a) ;
Write('Entrez 3 nombre réels : ') ;
Readln(b,c,d) ;
```

La fonction `Readln` lit des valeurs sur le périphérique d'entrée standard (clavier) les interprète dans le format de la variable et les range dans les arguments spécifiés. A chaque valeur saisie il faut valider par la touche entrée pour que la saisie soit prise en compte.

Remarque :

`Readln(...);`

$\Longrightarrow$ passage à la ligne suivante en ignorant ce qui reste sur la ligne

`Readln` ; peut être employé sans paramètre

La procédure `read()` permet de lire un ou plusieurs paramètres. `readln()` fait la même chose puis fait un `readln` ;

$\longrightarrow$Trois écritures équivalentes :

```
readln (a, b, c, d);
```

```
read (a, b, c, d); readln;
read(a); read(b); read(c); read(d); readln;
```

Remarques

- L'exécution d'une de ces lignes, on peut rentrer les données en les séparant par des espaces, des tabulations ou des retours chariot ↩.
- Il faut que les données lues correspondent au type attendu de chaque variable, sinon il y a une erreur à l'exécution.

2.5.3 Lecture directe du clavier

Il existe une fonction avec laquelle on peut entrer une valeur sans valider avec la touche entrée. Cette entrée manipule uniquement des caractères. Il faut donc déclarer des variables de type caractère

Exemple 24

```
C:=readkey; {lit une touche au clavier}
 C:=upcase(readkey); {lit une touche au clavier et la convertit en minuscule}
```

CHAPITRE 3

Structures de contrôle

3.1 Introduction

Dans un langage impératif, on peut définir l'état d'un programme en cours d'exécution par deux choses :

- L'ensemble des variables du programme ;
- L'instruction qui doit être exécutée

L'exécution d'un programme est alors une séquence d'affectations qui font passer d'un état initial à un état final considéré comme résultat. Les structures de contrôle définissent comment les affectations s'enchaînent séquentiellement.

3.2 La structure séquentielle

3.2.1 La notion de séquence

Une structure séquentielle ou séquence est une suite d'instructions rangées dans l'ordre où elles sont écrites. On parle d'une structure séquentielle chaque fois qu'il s'agit d'une séquence d'instructions simples devant être parcourues l'une après l'autre. Il apparaît immédiatement qu'une séquence est une structure algorithmique très élémentaire. Ainsi par exemple, si S1 et S2 représentent deux instructions, leur composition est notée :

`S1; S2;`
respectivement :

```
S1;
S2;
```

Ce qui signifie que :
- les instructions S1 et S2 sont exécutées une par une,
- chacune d'elles est exécutée exactement une fois,
- l'ordre dans lequel elles sont exécutées est le même que celui dans lequel elles ont été écrites,
- terminer l'instruction S2 implique de finir la structure séquentielle.

Il apparaît immédiatement que dans une structure séquentielle l'ordre des instructions joue un rôle fondamental. Ainsi par exemple, dans la séquence d'instructions.Ainsi par exemple, dans la séquence d'instructions

```
X1 := 2;
X2 := X1+3;
Writeln( X2 );
```

d'abord la valeur 2 est affectée à la variable `X1`, puis, la valeur 5 est affectée à la variable `X2` et le contenu de la variable `X2` est écrit sur le fichier standard de sortie, c'est-à-dire l'écran. Il n'est pas possible de revenir en arrière pour recommencer l'opération. L'exécution d'une instruction déterminée n'est possible que lorsque toutes les instructions qui la précèdent ont été exécutées.

Il est évident que la notion de séquence peut être généralisée pour un nombre quelconque N d'instructions : `S1; S2; S3; ... ; SN;`

La figurereprésente une traduction graphique de la structure séquentielle. Une telle représentation est appelée *Algorigramme* , *Ordinogramme* ou encore *Organigramme*.

Exemple 25 *Calcul d'un produit de 2 nombres*

```
Algorithm Produit;
Variable
    a,b : réel; (*opérandes*)
    p : réel ; (* résultat du produit*)
Début
```

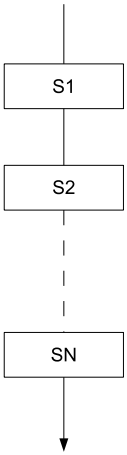


FIGURE 3.1 – Algorigramme de la structure séquentielle.

```
    Ecrire('Saisir le nombre a ');
    Lire(a);
    Ecrire('Saisir le nombre b ');
    Lire(b);
    p:=a * b;
    Ecrire (p);
Fin.
```

3.2.2 Les instructions composées

Une instruction composée est une série de N ($N \geq 2$) instructions qui doivent être exécutées en séquence. Une instruction composée est délimitée par les mots réservés `BEGIN` et `END` qui indiquent respectivement le début et la fin de l'instruction composée.

Voici quelques exemples d'instructions composées :

Exemple 26 `BEGIN`

```
Readln( X, Y );
Writeln( X, Y );
Z := X;
X := Y;
```

```
Y := Z;
Writeln( X, Y )
END;
```

Exemple 27 `S1;`
`S2`
`BEGIN S31; S32; S33 END;`
`S4;`
`BEGIN`
`S51;`
`BEGIN S521; S522 END;`
`S53`
`END;`
`S6;`

3.3 Les structures alternatives

Une instruction alternative permet de faire le choix entre une, deux ou plusieurs actions suivant qu'une certaine condition est remplie ou non. Une telle structure algorithmique est encore appelée sélection. On distingue deux types d'instructions alternatives :

- le choix simple (instruction `IF`) et
- la sélection multiple.(instruction `CASE`)

3.3.1 Le choix simple

Syntaxe :

En pseudo-code

```
SI  <condition> ALORS
                    <bloc_instructions1>
                SINON
                    <bloc_instructions2>
```

en PASCAL

```
IF  <condition> THEN
```

```
        <bloc_instructions1>
ELSE
        <bloc_instructions2>
```

Cette instruction exprime les instructions qui sont exécutées suivant que la condition (l'expression booléenne) ¡condition¿ est remplie. Elle est interprétée de la manière suivante :
SEMANTIQUE :
« Si la condition `<condition>` est *vraie*, alors exécuter l'instruction qui suit le mot réservé `THEN`, sinon exécuter l'instruction qui suit le mot réservé `ELSE`." »

Dans cette interprétation, l'exécution des instructions `<bloc_instructions1>`et `<bloc_instructions2>` est mutuellement exclusive ce qui signifie que seule une des deux instructions sera exécutée.

Points importants :

- Surtout pas de point virgule immédiatement avant le ELSE !!!
- Instruction alternative : facultative En effet les deux formes suivantes sont équivalentes.
  -
```
IF < condition > THEN
                    BEGIN
                        < instruction >;
                    END;
```

  -
```
IF < condition > THEN
                    BEGIN
                        < instruction >;
                    END
                ELSE ; { rien }
```

- Valeur de la condition : booléenne
- Blocs d'instructions :
  - Instructions simples
  - Instructions structurées
  - Instructions composées.

Organigrammes :

– sans SINON

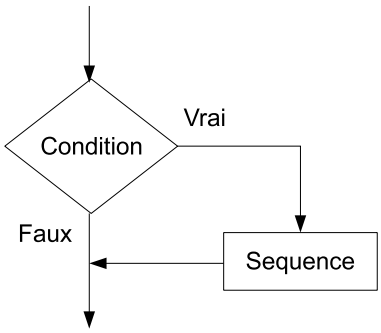


FIGURE 3.2 – Organigramme de la sélection sans alternative

– avec SINON

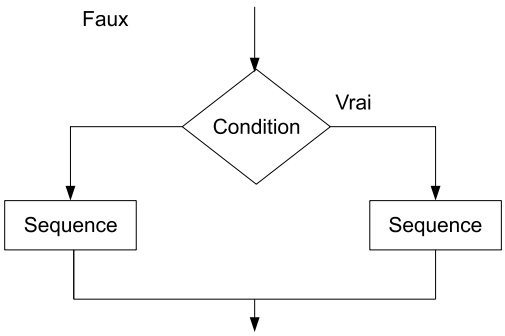


FIGURE 3.3 – Organigramme de la sélection avec alternative

Exemple 28 *Calcul d'une racine carrée (avec* `SI... ALORS`*)*

```
Algorithm SI_ALORS;
Variable
    x: réel ; (*opérande*)
    r: réel ;  (*résultat de la racine carrée*)
Début
    Ecrire ('Saisir le nombre x');
    Lire (x);
    Si x > 0  Alors
```

```
    Début
        r := racine (x);
        Ecrire (r);
    Fin
Fin.
```

Exemple 29 *Calcul d'une racine carrée (*`SI... ALORS... SINON`*)*

```
Algorithm SI_ALORS_SINON;
Variables :
    x: réel; (*opérande*)
    r: réel ;(*résultat de la racine carrée*)
Début
    Ecrire('Saisir le nombre x');
    Lire (x);
    Si x < 0 Alors
                Ecrire ('x est négatif')
             Sinon
                Début
                    r := racine (x);
                    Ecrire (r);
                Fin

Fin.
```

Ambiguïtés syntaxiques

Il faut bien percevoir que la formulation de l'instruction `IF` n'est pas toujours sans ambiguïté. Ainsi par exemple, l'ambiguïté syntaxique de la construction :

`IF < expr1 > THEN IF < expr2 > THEN < st1 > ELSE < st2 >;`

peut être évitée par le réécriture suivante :

```
IF < expr1 > THEN
                          IF < expr2 > THEN < st1 >
                                              ELSE < st2 >;
```

On dit ici que les instructions `IF` sont imbriquées.

Afin d'éviter des ambiguïtés de ce genre lors de la lecture d'un programme, il est vivement recommandé de bien mettre un `THEN` et un `ELSE` de la même structure alternative au même niveau vertical afin de ne pas les mélanger entre eux. Cette façon de procéder est appelée indentation ou paragraphage. L'identation est très souvent effectuée de manière automatique par un outil qui accompagne le compilateur (indenteur ou paragrapheur).

En général, une telle ambiguïté syntaxique est écartée définitivement soit en utilisant les parenthèses symboliques `BEGIN` et `END`, soit en respectant la règle suivante :

Règle : « "*La partie* `ELSE` *se rapporte toujours au mot réservé* `IF` *précédent le plus proche pour lequel il n'existe pas de partie* `ELSE`. »

Dans une construction de structures alternatives imbriquées il doit y avoir autant de mots `THEN` que de mots `IF`. Ainsi par exemple, dans le texte Pascal :

```
IF N>0 THEN IF A>B THEN Max := A ELSE Max := B;
```

la partie `ELSE` se rapporte au mot réservé `IF` situé à l'intérieur de la séquence ce qui peut être élucidé en écrivant :

```
IF N>0 THEN
                IF A>B THEN Max := A
                            ELSE Max := B;
```

Si ce n'est pas ce qu'on a voulu exprimer, il faut se servir des parenthèses symboliques `BEGIN` et `END` pour forcer des appartenances respectives comme par exemple :

```
IF N>0 THEN
                  BEGIN
                       IF A>B THEN Max := A
                  END
             ELSE Max := B;
```

Le lecteur se rendra bien compte de la signification différente des deux constructions précédentes.

Exemple 30 *Equation du premier degré avec C=0*
Ecrire un programme qui résoud une équation du premier degré Ax+b=0 qui lit les valeurs de A et B entrées par l'utilisateur.

```
Program Premier_Degre;
Var
        A, B : real;
Begin
     write('entrez les coefficients A et B : ');
     readln(A,B);
     if A=0   then {évaluation de la condition}
                if B=0   then
                           writeln('Indéterminé !')
                         else
                           writeln('Impossible !')
                       else
                          writeln('La solution est : ',-B/A:10:3);
    End.
```

Exemple 31 ***Maximum de deux nombres***
Ecrire un programme qui calcule le maximum de deux nombres entrés au clavier.

```
Program MAXIMUM_DEUX ;
var  X,Y : real ;
     MAX : real ;
Begin
     writeln('Tapez les deux nombres:')
     read (X, Y) ;
     if  X > Y  then {évaluation de la condition}
                   MAX := X
                else
                   MAX := Y ;
       writeln('Le plus grand nombre est ',MAX);
End.
```

3.3.2 La sélection multiple : le CAS...

Cette méthode est utilisée pour tester une solution parmi N. Par exemple, au cas où un menu est proposé à l'utilisateur (1) pour lire, 2) pour écrire, 3) pour calculer, 4) pour

sortir etc...), il est important de tester si l'utilisateur a tapé 1, 2, 3 ou 4. Au lieu d'utiliser plusieurs IF... THEN... ELSE... imbriqués, il est préférable de choisir une sélection multiple (CASE en Pascal). Ainsi au lieu d'écrire :

```
IF reponse=1 THEN
              Instructions de lecture...
            ELSE
              IF reponse=2 THEN
                            Instructions d'écriture...
                          ELSE
                            IF reponse=3 THEN
                                          instructions de calcul...
```

avec un organigramme représenté par la figure 3.4,

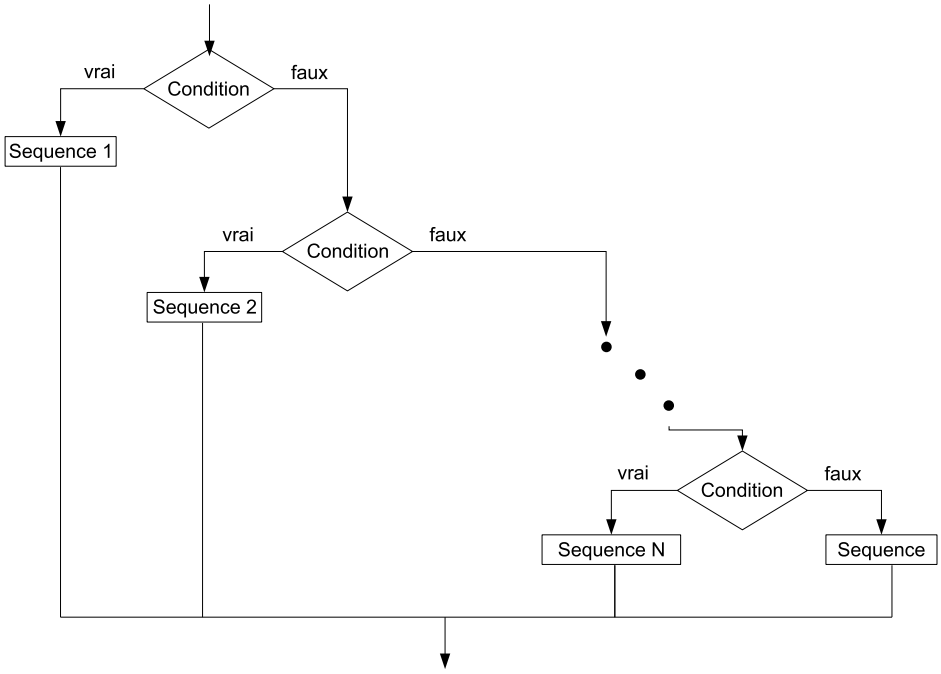


FIGURE 3.4 – Organigramme de la structure avec IF imbriqués

Il serait préférable d'écrire :

```
CASE reponse OF
          1 : Instructions de lecture...
          2 : Instructions d'écriture...
          3 : instructions de calcul...
        End;
```

L'instruction de *sélection multiple* ou instruction `CASE` encore appelée *analyse par cas* est une généralisation supplémentaire de l'instruction `IF`. Elle est fréquemment utilisée en programmation s'il faut faire le choix entre plus de deux alternatives. Sa formulation est simplifiée par l'instruction `CASE`.

Syntaxe

en pseudo-code

```
CAS variable PARMI
  constante1  : suite d'instructions1
  constante2  : suite d'instructions2
  intervalle1 : suite d'instructions3
  ...
  SINON suite d'instructions par defaut
END;
```

en Pascal

```
CASE variable OF
  constante1 : suite d'instructions1
  constante2 : suite d'instructions2
  intervalle1 : suite d'instructions3
  ...
  ELSE suite d'instructions par defaut
END;
```

La séquence précédente est interprétée de la manière suivante :

« Si variable prend la valeur `constante1` , alors exécuter la suite d'`instructions 1`. Si variable prend la valeur `constante2`, alors exécuter la suite d'`instructions2`, si variable est dans l'`intervalle1` alors, exécuter la suite d'`instructions3` etc. , sinon exécuter la suite d'`instructions par défaut` qui suit le mot réservé `ELSE`. »

- Comme dans l'instruction `IF`, l'exécution de chaque branche est mutuellement exclusive.
- La variable variable est appelée sélecteur et doit être d'un type scalaire.
- Les constantes `CASE` doivent être toutes différentes et du même type que le sélecteur. Elles sont interprétées comme des étiquettes.
- Seules les égalités sont possibles au niveau du test (Pas de comparaisons de type `<,>,<=,>=` ou `<>`) On peut néanmoins utiliser des intervalles .
- On peut donner une liste de constantes, ou des intervalles de constantes. **Attention**, chaque valeur possible ne doit être représentée qu'une fois au plus (sinon il y a erreur à la compilation). Par exemple, on ne peut pas faire des intervalles se chevauchant, comme `3..6` et `5..10`, les cas `5` et `6` étant représentés `2` fois.

Exemple 32 *Simuler une calculatrice*

```
Program calculette ;
  var  A, B : real ;
         RESULTAT : real;
         TOUCHE : char;
   Begin
       write('entrez une opération ');
       write('(taper un nombre, un opérateur puis un nombre) : ');
       readln(A,TOUCHE,B);
       case TOUCHE of
           '+' : RESULTAT:= A+B;
           '-' : RESULTAT:= A-B;
           '*' : RESULTAT:= A*B;
           '/' : RESULTAT:= A/B;
          end;
       writeln(A, TOUCHE, B,' = ', RESULTAT);
   end.
```

Exemple 33 *Ecrire un programme qui lit un caractère, puis classe ce caractère comme espace, lettre, digit ou autre.*

```
PROGRAM caractere;
TYPE  nat_t = (Espace, Lettre, Digit, Autre);
```

```
VAR  nat : nat_t; { nature }
       c : char;
BEGIN
    write ('Rentrez un caractere :');
    readln(c);
    { analyse de c }
    case c of
        'a'..'z', 'A'..'Z', '_' : nat := Lettre;
        '0'..'9' : nat := Digit;
        ' ' : nat := Espace;
        else nat := Autre;
    end; { case c }
    { affichage de nat }
    case nat of
        Espace : writeln ('Espace');
        Lettre : writeln ('Lettre');
        Digit : writeln ('Digit');
        Autre : writeln ('Autre');
        else { case nat }
        writeln ('Erreur case nat : ', ord(nat), ' non prevu');
    end; { case nat }
END.
```

3.3.3 Le concept de condition

Une *condition* est une expression booléenne. La valeur d'une condition est du type booléen dont les seules valeurs sont les valeurs de vérité `True` et `False`. Une valeur booléenne est obtenue en appliquant les opérateurs booléens `AND`, `OR` et `NOT` ainsi que les opérateurs relationnels `=`, `<>`, `<`, `<=`, `>`, `>=` à des opérandes booléennes.

On envisage les *conditions de relation simple* et les *conditions de relation composée*. Dans une condition de relation simple deux expressions arithmétiques sont comparées au moyen des opérateurs relationnels. Une condition de relation composée se construit au moyen de conditions de relation simple reliées les unes aux autres par les opérateurs booléens.

Exemple 34 `NOT( A<=B )`
`( A>3 ) OR ( B<5 )`
`( X>3 MOD 4 ) AND ( Sqr( B )<=6 )`

L'opérateur `AND` permet de simplifier l'écriture d'une série d'instructions `IF` imbriquées de la forme suivante :

```
IF < expression 1 > THEN
                        BEGIN
                           IF < expression 2 > THEN
                                                BEGIN
                                                  < statement 1 >;
                                                END
                                             ELSE
                                                BEGIN
                                                  < statement 2 >;
                                                END;
                        END
                           ELSE
                              BEGIN
                                < statement 2 >;
                              END;
```

sous la forme :

```
IF < expression 1 > AND < expression 2 > THEN
                                           BEGIN
                                              < statement 1 >;
                                           END
                                         ELSE
                                           BEGIN
                                              < statement 2 >;
                                           END;
```

L'opérateur `OR` permet de simplifier l'écriture d'une série d'instructions `IF` imbriquées de la forme suivante :

```
IF < expression 1 > THEN
                        BEGIN
                            <statement 1 >;
                        END
                    ELSE
                        BEGIN
                            IF < expression 2 > THEN
                                                BEGIN
                                                   < statement 1 >;
                                                END
                                              ELSE
                                                BEGIN
                                                   < statement 2 >;
                                                END;
                        END;
```

sous la forme :

```
IF < expression 1 > OR < expression 2 > THEN
                                          BEGIN
                                            < statement 1 >;
                                          END
                                       ELSE
                                          BEGIN
                                             < statement 2 >;
                                          END;
```

3.4 Les structures répétitives

3.4.1 Définition

Une structure répétitive permet d'exécuter une séquence d'instructions un certain nombre de fois jusqu'à ce qu'une condition déterminée soit remplie ou non. La répétition se fait par l'intermédiaire de boucles et d'itérations.

Une *boucle* consiste à parcourir 'une partie d'un programme un certain nombre de fois. Une *Itération* est la répétition d'un même traitement plusieurs fois.

La même séquence d'instructions est réitérée plusieurs fois au cours d'une même exécution

On distingue les boucles à bornes définies (`POUR...FAIRE`) et les boucles à bornes non définies (`TANTQUE...FAIRE` et REPETE...JUSQU'A. Toute structure répétitive - est composée de trois éléments :

- d'une initialisation d'un compteur ;
- d'une condition ;
- d'un bloc d'instructions.

Toute modification d'un quelconque de ces trois éléments nécessite un contrôle de cohérence des deux autres.

3.4.2 Boucle à bornes définies (POUR...FAIRE)

Dans le cas d'une boucle à bornes définies, nous connaissons le nombre d'itérations à effectuer, grâce aux valeurs des bornes minimum et maximum fournies dans la définition de la boucle.

Un indice de boucle varie alors de la valeur minimum (initiale) jusqu'à la valeur maximum (finale)

SYNTAXE :

En pseudo-code

```
POUR variable:=valeur_initiale A valeur_finale FAIRE
                <séquence d'instruction>
```

En PASCAL

```
FOR variable := valeur_initiale TO valeur_finale DO
                        Bloc d'instructions;
```

Remarques

- la variable doit être de type scalaire (entier, énuméré, intervalle ou caractère) elle ne peut pas être réelle

– si `valeur_initiale > valeur_finale` le `FOR` est ignoré

Exemple 35
```
program boucle_for;
var     i:integer;
begin
   for i:=1 to 5 do
         writeln('le carré de ', i, ' est :', sqr(i));
   writeln;
   writeln('FIN. A la prochaine...');
end.
```

Il est possible d'imbriquer plusieurs boucles FOR :

```
 For X1 := C1 to C2 do
     Begin
         ...
         For X2 := D1 to D2 do
             Begin
                 ...
             End;
         ...
     End;
```

Exemple 36 *Table de multiplication*

```
PROGRAM table_multiplication;
VAR
    i, j : integer;
BEGIN
    for i := 1 to 10 do
        begin
            for j := 1 to 10 do write (i*j : 3);
            writeln;
        end;
END.
```

3.4.3 Boucles à bornes non définies

Lorsque les bornes ne sont pas connues, il existe deux autres types de boucles :

- Boucle `TANT QUE ... FAIRE ...`
- Boucle `REPETER ... JUSQU'A ...`

Syntaxe de la boucle `TANT QUE` :

En pseudo-code

```
TANT QUE <condition>  FAIRE
    <séquence d'instructions>
```

En PASCAL

```
WHILE expression DO
     Bloc d'instructions;
```

Remarques

- arrêt si expression est fausse
  $\Longrightarrow$pas de boucle si faux au départ
- incrémentation gérée par le programmeur lui-même
  $\Longrightarrow$pas d'augmentation automatique d'une variable (contrairement à la boucle FOR)
- Les variables de l'expression `expression` doivent être initialisées avant le while, pour que au premier passage `expression` puisse être évaluée.

Organigramme

Exemple 37

```
program boucle_while;
var
    i:integer;
begin
    i:=1;
    while i <= 5 do
        begin
            writeln('le carré de ', i, ' est :', sqr(i));
            i:=i+1;  { incrémentation gérée par le programmeur }
        end;
    writeln;
```

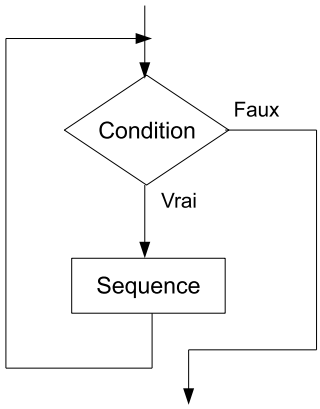


FIGURE 3.5 – Organigramme de la structure TANTQUE... FAIRE

```
    writeln('FIN. A la prochaine...');
end.
```

Syntaxe de la boucle `REPETER` :

EN PSEUDO-CODE

```
REPETER
      <séquence d'instructions>
JUSQU'A  <condition>
```

EN PASCAL

```
REPEAT
   Bloc d'instructions
UNTIL expression;
```

Remarques

La boucle s'effectue tant que l'expression est fausse, arrêt quand l'expression est vraie. C'est le contraire de la boucle `WHILE`. Contrairement au `WHILE`, il y a au moins un passage (1 boucle), même si l'expression est vraie ; De même que pour le `WHILE`, c'est le programmeur qui gère l'incrémentation.

Organigramme

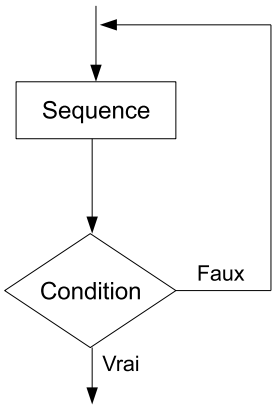


FIGURE 3.6 – Organigramme de la structure REPETER...JUSQUE

Exemple 38

```
program boucle_repeat;
var
    i:integer;
begin
      repeat
          writeln('le carré de ', i, ' est :', sqr(i));
          i:=i+1;      { incrémentation gérée par le programmeur }
      until i>5;
      writeln;
      writeln('FIN. A la prochaine...');
end.
```

Exemple 39 *exécution répétitive d'un programme*

```
Variables :
    a,b : réel; (*opérandes *)
      p : réel; (*résultat du produit *)
      c : caractère; (*réponse de l'utilisateur *)
Début
    Répéter
        Ecrire ('Saisir le nombre a');
        Lire (a);
```

```
        Ecrire ('Saisir le nombre b');
        Lire (b);
        p:=a*b;
        Ecrire (p);
        Ecrire ('encore un calcul ? Non touche N ; Oui autre touche');
        Lire (c);
    Jusqu'à c = 'N';
Fin.
```

NB : Faire attention aux conditions initiales, aux conditions d'arrêt et à l'incrémentation sinon la boucle risque d'être infinie. Les deux boucles peuvent être choisies indifféremment. Cependant, l'une est le contraire de l'autre, au niveau de la condition d'arrêt :

```
TANTQUE condition1 FAIRE
    <Bloc d'instructions>
```

$\Longleftrightarrow$

```
REPETER
    <Bloc d'instructions>
JUSQU'A non (condition1)
```

- Tant que condition1 est vraie, faire bloc d'instructions...
- Répéter bloc d'instructions, jusqu'à ce que condition1 ne soit plus vraie

Dans ce cas, la condition d'arrêt de la boucle `TANTQUE` est l'opposée de la condition d'arrêt de la boucle `REPETER`.

Exemple 40

```
TANTQUE  (i<>10) FAIRE
    i:= i+1 {on fait varier i jusqu'à 10}
```

est équivalent à :

```
REPETER
   i:=i+1
JUSQU'A (i=10)
```

Il est toujours équivalent d'utiliser une boucle `TANTQUE` ou une boucle `REPETER`. Cependant, il existe une petite différence entre les deux boucles : Dans le cas d'une boucle `REPETER... JUSQU'A`, le bloc d'instructions est effectué au moins une fois, ce qui n'est pas forcément vrai pour une boucle `TANTQUE`. En effet, pour ce dernier type de boucle, si la condition est fausse dès le départ, le bloc d'instructions ne sera pas du tout exécuté. En revanche, avec une boucle `REPETER ... JUSQU'A`, si la condition est fausse dès le départ, le bloc d'instructions sera quand même exécuté une fois.

Remarque :
Les boucles `REPETER` et `TANTQUE` peuvent être utilisées même si les bornes sont définies. Dans ce cas, il est bien entendu préférable d'utiliser une boucle `POUR` (vue précédemment).

3.4.4 Exemple comparatif

Nous allons à présent traiter le même exemple, avec trois boucles différentes. Il s'agit de reconstruire l'opération de multiplication, en effectuant des sommes successives. Soit à effectuer le produit des entiers naturels a et b (distincts de 0).
Données : a multiplicande, b multiplicateur
Résultat : P produit Méthode : ajouter b fois le multiplicande

Exemple 41 *Forme 1 avec* `POUR`

```
Algorithme Avec_Pour;
var a,b,i,P:entier;
Debut
    Ecrire('Donner a et b');
    Lire(a,b);
    P:=0;
    Pour i:=1 à b Faire
        P:=P+a;
    Ecrire('Le produit est',P);
Fin.
```

Exemple 42 *Forme 2 avec* `TANT QUE`

```
Algorithme Avec_TANTQUE;
var a,b,i,P:entier;
```

```
Debut
    Ecrire('Donner a et b');
    Lire(a,b);
    P:=0;i:=1;
    TantQue (i<=b) Faire
        Debut
          P:=P+a;
          i:=1+1;
        Fin;
    Ecrire('Le produit est',P);
Fin.
```

Exemple 43 *Forme 3 avec* `REPETER`

```
Algorithme Avec_REPETER;
var a,b,i,P:entier;
Debut
    Ecrire('Donner a et b');
    Lire(a,b);
    P:=0;i:=1;
    Repeter
        P:=P+a;
        i:=1+1;
    Jusque (i>b);
    Ecrire('Le produit est',P);
Fin.
```

Nous pouvons modifier les algorithmes précédents en y rajoutant des contrôles de saisie sur les valeurs de a et de b, puisqu'elles doivent être strictement positives. Le controle peut être fait avec la boucle `TANTQUE` ou la boucle `REPETER`. Pour la forme 1 nous avons :

Exemple 44

```
Algorithme Avec_REPETER_Controle_REPETER;
var a,b,i,P:entier;
Debut
```

```
    Repeter
        Ecrire('Donner a et b');
        Lire(a,b);
    Jusque (a>0) ET (b>0);
    P:=0;i:=1;
    Repeter
        P:=P+a;
        i:=1+1;
    Jusque (i>b);
    Ecrire('Le produit est',P);
Fin.
```

Exemple 45

```
Algorithme Avec_REPETER_Controle_TANTQUE;
var a,b,i,P:entier;
Debut
      Ecrire('Donner a et b');
      Lire(a,b);
      TantQue (a<=0) OU (b<=0) Faire
            Debut
                Ecrire('Donner a et b');
                Lire(a,b);
            Fin
      P:=0;i:=1;
      Repeter
         P:=P+a;
         i:=1+1;
      Jusque (i>b);
      Ecrire('Le produit est',P);
Fin.
```

CHAPITRE 4

Les tableaux et les chaînes de caractères

4.1 Tableaux à 1 dimension

4.1.1 Définition

Un **tableau** est une collection ordonnée de variables (appelées composantes du tableau) ayant toutes le même type. On accède à chacune de ces variables individuellement à l'aide d'un indice.

indice=valeur ordinale, c'est à dire ayant un domaine de valeurs finies et ordonnées. L'indice doit être de type scalaire. Chaque composante du tableau est référencée au moyen d'un indice qui indique la position relative de la composante dans la suite et qui permet donc d'accéder à la composante.

Un tableau T est donc défini par le ***type des indices*** et par le ***type des composantes***. Les tableaux à *une dimension*, c'est-à-dire à un type d'indices, appelés encore en termes mathématiques ***vecteurs***, sont composés d'un ensemble homogène d'éléments de ***même type***.

4.1.2 Déclaration d'un tableau

La déclaration d'un tableau s'effectue en donnant :

– son nom (identificateur de la variable tableau)

- le domaine de variation de l'indice délimité par une borne inférieure correspondant à l'indice minimal et la borne supérieure correspondant à l'indice maximal.
- le type de ses composantes

  Syntaxe de définition :

```
Var <Nom_tab> : Tableau[<indice min>..<indice_max>] de <type des composants>;
```

Exemple 46

```
var tab_entier : Tableau[1..10] d'entiers;
```

En PASCAL, on a :

```
Var <nom_tab> : ARRAY [<indice min>..<indice max>] OF <type des composants> ;
```

Exemple 47

```
var  tab_entier : ARRAY [1..10] OF integer;
```

La structure `ARRAY` n'est pas une structure "dynamique" mais "statique", c'est-à-dire une structure qui ne change pas de taille au cours de l'exécution du programme. Par conséquent, le nombre d'éléments d'un tableau doit être fixé à priori de manière définitive lors de sa définition.

Soit par exemple l'extrait de programme suivant :

Exemple 48

```
CONST MaxElements = 100;
TYPE Dim = 1..MaxElements;
           Alphabet = 'A'..'Z';
. . .
VAR Boole: ARRAY [ Dim ] OF Boolean;
         Frequency: ARRAY [ Alphabet ] OF Integer
```

Dans cet exemple, le tableau `Boole` est un vecteur de `100` composantes qui ne peuvent prendre que les valeurs `True` et `False`. Le tableau `Frequency` représente un vecteur de `26` composantes, chacune étant de type entier et indicée par une lettre majuscule.

Souvent, on représente la structure d'un tableau moyennant la définition d'un type en écrivant :

```
TYPE <identificateur de type> = ARRAY[<type index>] OF <type composant>;
```

Et ensuite en déclarera les variables de ce type tableau en écrivant :

```
VAR < liste variable >: < identificateur de type > ;
```

Soit pour l'exemple ci-dessous :

Exemple 49

```
CONST MaxElements = 100;
TYPE Dim = 1..MaxElements;
          Alphabet = 'A' . . 'Z';
          TBoole = ARRAY [ Dim ] OF Boolean;
          TFrequency = ARRAY [ Alphabet ] OF Integer;
.......
VAR Boole: TBoole;
         Frequency: TFrequency;
```

Si les variables `I` et `J` prennent respectivement leurs valeurs dans les intervalles `Dim` et `Alphabet`, alors `Boole[ I ]` et `Frequency[ J ]` désignent respectivement les composantes situées à la I-ième place dans le tableau `Boole` et à la J-ième place dans le tableau `Frequency`. Donc, pour accéder à une composante d'un vecteur, il suffit de préciser son indice.

Dans la mémoire centrale, les éléments d'un tableau sont stockés de façon linéaire, dans des zones contiguës. Une variable T de type tableau à une dimension peut être représenté comme suit :

1	2	3	4	5	6	7	8
10	0	1	4	7	2	6	25

FIGURE 4.1 – Tableau à une dimension (vecteur)

Remarques

Quand on ne connaît pas à l'avance le nombre d'éléments exactement, il faut majorer ce nombre, quitte à n'utiliser qu'une partie du tableau. Il est préférable de définir un TYPE tableau réutilisable, plutôt que de déclarer à chaque fois, en `VAR`, le tableau en entier. De même, l'utilisation judicieuse des constantes permet de modifier aisément le programme et limite les sources d'erreurs. Il est interdit de mettre une variable dans l'intervalle de définition du tableau.

4.2 Tableaux à deux dimensions

4.2.1 Définition

Un tableau à deux dimensions, c'est-à-dire à deux types d'indices, est un tableau dont le type des composantes est un type tableau de dimension un, donc, un vecteur. En termes mathématiques, un tableau à deux dimensions est appelé une ***matrice***.

La définition d'un tableau à deux dimensions peut s'effectuer de la manière suivante :

```
TYPE <identificateur> = ARRAY[1..M, 1..N] OF < type-composantes >;
```

Où `< type-composantes >` peut être un type quelconque, sauf le type tableau.

Ainsi par exemple, en termes mathématiques, une matrice réelle Mat d'ordre $N \times M$ est représentée par un tableau de M vecteurs composés chacun de N nombres réels.

Exemple 50
```
CONST MaxLine = 10; {nombre maximal de lignes}
      MaxColumn = 20; {nombre maximal de colonnes}
TYPE
      TLine = 0..MaxLine;
      TColumn = 0..MaxColumn;
      Matrix = ARRAY [ TLine, TColumn ] OF Real;
. . .
VAR Mat: Matrix;
```

Si les variables `L` et `C` prennent respectivement leurs valeurs dans les intervalles `TLine` et `TColumn`, alors `Mat[L,C]` ou bien `Mat[L][C]` désigne la composante de la matrice située à la `L`-ième place dans la `C`-ième colonne, respectivement à la `C`-ième place dans la `L`-ième ligne (voir figure). Par convention, nous allons indiquer souvent par `L` l'indice de parcours des lignes et par `C` l'indice de parcours des colonnes d'une matrice. Un tel tableau sera représenté sous la forme indiquée par la figure 4.2 :

Donc, pour accéder à une composante d'une matrice, il est nécessaire de préciser deux indices, à savoir d'abord l'indice de ligne, puis l'indice de colonne.

Remarques :

– Dimension du tableau ci-dessus : 2 (matrice)

– Nombre de cases disponibles : `MaxLine x MaxColumn`

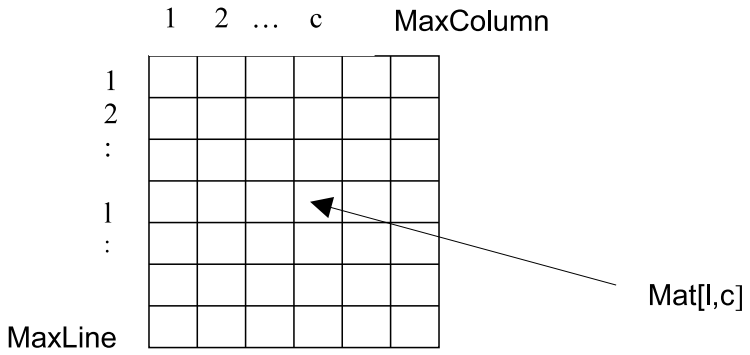


FIGURE 4.2 – Tableau à deux dimensions (matrice)

- le tableau `Mat` est bidimensionnel $\Longrightarrow$ tous les $\texttt{Mat}_{ij}$ sont des variables (i = 1 ... M, j = 1 ... N) de même type
- Accès au contenu d'une case : $\texttt{Mat}_{ij}$

Exemple 51 *Déclarations de tableaux :*

1. *Cas le plus classique*

```
  type LISTE = array [1..100] of real ;
TABLEAU = array [1..10,1..20] of integer ;
CHAINE = array [1..80] of char ;
```

2. *Cas plus complexes*

```
  type POINT = array [1..50,1..50,1..50] of real ;
MATRICE = array [1..M,1..N] of real ;
SYMPTOME = (FIEVRE, DELIRE, NAUSEE) ;
MALADE = array [symptome] of boolean ;
CODE = 0..99 ;
CODAGE = array [1..N_VILLE] of code ;
```

3. *Définition récursive*

```
type CARTE = array [1..M] of array [1..N] of real ;
var  LIEU : carte ;
          --> accès à un élément par   LIEU [I] [J]
```

4.3 Tableaux multidimensionnels

Il est possible de généraliser la définition des tableaux à plusieurs dimensions, c'est-à-dire à plusieurs types d'indices, par simple extension de la notation :

```
TYPE TableauMultidim = ARRAY[<type index>,<type index>,...,<type index>]
                                              OF < type composantes >;
```

où `<type composantes>` peut être un type quelconque, sauf le type tableau. En règle générale, il est peu utile d'utiliser des tableaux à plus de trois dimensions, puisque (1) la clarté diminue considérablement dans ce cas et (2) l'encombrement de l'espace-mémoire devient trop important.

4.4 Manipulations élémentaires de tableaux

Les langages de programmation usuels n'offrent en général qu'un ensemble très réduit de primitives de base définies sur le type tableau. Ainsi est-il presque toujours nécessaire de manipuler les tableaux en décrivant les fonctions que l'on veut réaliser en termes de composantes. Les exemples suivants montrent comment il est possible de réaliser des opérations qui concernent toutes ou une partie des composantes d'un ou de plusieurs tableaux.

4.4.1 Création et affichage d'un tableau

Créer un tableau équivaut à attribuer une valeur à chacune de ses composantes. La création peut se faire soit par une affectation, soit par une saisie des valeurs.

Exemple 52 *Affectation*

```
T[1]:=2; T[2]:=10;...T[20]:=8;
```

Saisie

```
For  i:=1 To 20 Do readln(t[i]);
```

L'Affichage consiste parcourir le tableau à l'aide d'une boucle et à afficher ses différentes composantes

Exemple 53 *Le programme suivant permet de "remplir" un tableau à l'aide d'une boucle Repeat-Until.*

```
Program Mon_tableau;
Const
    Taille_max=10;
Type
    TAB=array[1..Taille_max] of integer;
Var
    Tableau:TAB;
    indice: integer;
Begin
    for indice:=1 to Taille_max do
        Tableau[indice]:=0;
    indice:=1;
    Repeat
        write('entrez le N° ',indice,':');
        readln(Tableau[indice]);
        indice:=indice+1;
    until indice>Taille_max;
End.
```

4.4.2 Maximum et Minimum d'un tableau

La recherche du maximum (minimum) consiste à mettre dans la variable `max` (`min`) le premier élément du tableau `T`, de parcourir ensuite le reste du tableau et de mettre à jour cette variable `max` (`min`) si l'élément en cours d'examen est supérieur (inférieur) au `max` (`min`) courant : pour le maximum on a :

Exemple 54

```
max := T[1] ;
for i :=2 to Taille_max do
        if T[i] > max then
                        max := T[i];
writeln('Le maximum de T:', max) ;
```

et pour le minimum on a :

Exemple 55

```
min := T[1] ;
for i :=2 to Taille_max do
            if T[i] < min then
                            min := T[i];
writeln('Le minimum de T:', min) ;
```

4.4.3 Recherche séquentielle d'un élément dans un tableau

La recherche séquentielle d'un élément `a` dans un vecteur `T` consiste à parcourir ce dernier et de trouver un indice `i` tel que `T[i]=a`.

Exemple 56

```
i := 1;
while (T[i]<>a) and (i<=n) do
i:=i+1;
if (i>n) then
write(a ,'n''appartient pas à T')
else
write(a,' appartient à T');
```

On peut aussi utiliser une variable booléenne trouve qui va contenir le résultat de la recherche :

Exemple 57

```
i := 1; trouve:=false
while  (i<=n)  and (trouve=false) do
    begin
          trouve:=(t[i]=a);
          i:=i+1;
    end;
if trouve then
     write(a,' appartient à T')
        else
     write(a, ' n''appartient à T');
```

4.4.4 Recherche dichotomique d'un élément dans un tableau ordonné

La bonne idée est de comparer l'élément recherché avec l'élément situé au milieu du tableau :

- si l'élément recherché est plus petit, on continue la recherche dans la première moitié du tableau ;
- s'il est plus grand, on continue dans la seconde moitié du tableau ;
- On s'arrête si l'élément recherché a été trouvé ou si on ne peut plus diviser le tableau.

Dans ce qui suit :

- `x` représente l'élément que l'on veut chercher
- `gauche, droite` : délimitent la zone du tableau à l'intérieur de laquelle on veut rechercher `x`

Exemple 58

```
.........................
VARIABLES  gauche,droite: entier;
                        milieu : entier
DEBUT
   (*saisie du tableau T de n éléments et de l'élément x à rechercher*)
   ....................................
```

```
   gauche:=1; droite:=n; trouve :=faux
   TANTQUE (gauche<=droite) ET non Trouve FAIRE
          DEBUT
             milieu:=(gauche +droite ) DIV 2;
             SI (x = t[milieu])  ALORS
                                   trouve :=vrai
                                 SINON
                                   SI (x  < t[milieu]) ALORS
                                                         droite:=milieu-1;
                                                       SINON
                                                         gauche:=milieu+1;
          FIN
   SI trouve ALORS
               ecrire(x,' appartient à T')
             SINON
               ecrire(x, ' n''appartient à T');

FIN.
```

4.4.5 Recherche d'un élément dans une matrice

Nous nous proposons d'écrire un programme qui détermine si l'un au moins des éléments de la matrice `Mat` d'ordre $N \times M$ est égal à a (élément recherché). Voici un extrait de ce programme (la matrice `Mat` a été définie dans le paragraphe 4.2.1) :

Exemple 59 .....

```
trouve := False;
l := 1;
while ( l<=n ) and not trouve do
        begin
            c := 1;
            while ( c<=m ) and not trouve do
                    begin
                        trouve := ( Mat[ l, c ] = a );
```

```
                    c:=c+1;
                end; {-- WHILE, c>m or trouve}
            l:=l+1;
        end; {-- WHILE, l>N or trouve}

if trouve then
            write(a,' appartient à Mat')
          else
            write(a, ' n''appartient à Mat');
```

4.4.6 Initialisation d'une matrice unité

Nous désirons réaliser un programme permettant l'initialisation d'une matrice unité de dimension 10. Il s'agit donc d'une matrice à 10 colonnes et 10 lignes, ne comportant que des 0, sauf sur sa diagonale ou il n'y a que des 1.

Exemple 60

```
Program SOMME_MATRICE ;
const
    L_MAX = 10 ;
    C_MAX = 10 ;
   I, J : Integer;
Type
    TAB = array [1 . . l_max, 1 . . c_max]  of  integer ;
Var
    MAT : tab ;
Begin
    for  I := 1 to  l_max  do
        begin
            for  J := 1  to  c_max  do
                begin
                    if  I = J then
                            MAT [I, J]  := 1
                        else
                            MAT [I, J]  := 0 ;
```

```
                    write (MAT [I, J]) ;
                end ;
            writeln ;
        end ;
    End.
```

4.4.7 Fusion de deux tableaux ordonnés

Soient deux tableaux triés par ordre croissant `T1` et `T2` possédant respectivement `taille1` éléments et `taille2` éléments, on veut fusionner les tableaux `T1` et `T2` en un troisième tableau `T3` de `taille` (= `taille1+taille2`) éléments triés par ordre croissant. Pour ce travail nous allons choisir trois compteurs `i`, `j` et `k` pour décrire respectivement `T1`, `T2` et `T3` où `T3` est le tableau qui recevra le tableau trié résultant de la fusion. Un extrait de l'algorithme sera le suivant :

Exemple 61

```
.................................
Variables i,j,k :entier;
                 T1,T2,T3 : TAB.;
DEBUT
  (*saisie de T1 et de T2*)
  ............................
  i=1;j=1,k=0;
  TANTQUE ((i<=taille1) ET(j<=taille2)) FAIRE
       SI (T1[i]<T2[j]) ALORS   (* copier les éléments du T1 tant qu'ils*)
                          DEBUT   (*sont inférieurs a l'élément courant de T2*)
                            k:=k+1;
                            T3[k]:=T1[i];
                            i:=i+1;
                          FIN
                        SINON    (* et inversement*)
                          DEBUT
                            k:=k+1;
                            T3[k]:=T2[j];
                            j:=j+1;
```

```
                        FIN;
    SI (i>taille1) ALORS   (*recopier des éléments  de T2, s'il en reste*)
                    TANTQUE (j<=taille2) FAIRE
                            DEBUT
                              k:=k+1;
                              T3[k]:=T2[j];
                              j:=j+1;
                             FIN
                   SINON       (* idem pour T1*)
                     TANTQUE (i<=taille1) FAIRE
                             DEBUT
                               k:=k+1;
                               T3[k]:=T1[i];
                               i:=i+1;
                             FIN;
  Taille:=k;
FIN;
```

4.5 Les chaînes de caractères

4.5.1 Définition

Une chaîne de caractères est une suite de caractères regroupés dans une même variable. Elle correspond à un tableau de 255 caractères au maximum.

`Type string = array[1..255] of char`

On utilise le Type `string`, qui n'est pas un type standard du pascal (certains compilateurs Pascal ne l'acceptent pas). Ce type permet de manipuler des chaînes de longueur variable.

Déclaration :

```
var  S  : string;
     S2 : string[20];
```

Il est possible, comme dans tout tableau, d'accéder à un caractère particulier de la chaîne `S`, en écrivant simplement : `S[i]`. On peut aussi manipuler la chaîne de manière

globale, sans passer élément par élément. Ceci est très utile pour des affectations ou des tests

Exemple 62

```
S := 'Bonjour';
S[4] := 's';
S[6] :='i'
```

$\Longrightarrow$ *A présent,* `S` *vaut* `'Bonsoir'`

```
S := 'ok';
```

$\Longrightarrow$ *A présent,* `S` *vaut* `'ok'` *On constate que la taille de* `S` *est variable (7 caractères au départ, 2 caractères ensuite)*

4.5.2 Opérateurs et fonctions

Il est possible de comparer des chaînes de caractères, on utilise alors les opérateurs : `=, <, >, <=, >=, <>`

Dans ce cas, l'ordre utilisé est l'ordre lexicographique (utilisation du code ASCII)

Exemple 63 *Soit* `b` *un booléen ;* `b` *est-il vrai ou faux ?*

```
b := 'A la vanille' < 'Zut'; { vrai }
b := 'bijou' < 'bidon'; { faux, c'est > car 'j' > 'd' }
b := 'Bonjour' = 'bonjour'; { faux, c'est < car 'B' < 'b' }
b := ' zim boum' > 'attends !'; { faux, c'est < car ' ' < 'a' }
```

Exemple 64

```
var
S: String[4];
begin
S := '';
Write(S) ; { rien n'est affiché : la chaîne est vide }
S := 'toto' ;
S[1] := 'm' ;
Write(S) ; { la chaîne de caractère contient « moto » }
end.
```

CONCATÉNATION :

`s:=concat(s1, s2, s3 ...);` (ou parfois `s:= s1 + s2 + s3...`)

Exemple 65

```
s1:='bon';
s2:='jour';
s3 := s1 + s2 ;
```

Nous obtenons alors `s3` *valant* `'bonjour'`

LONGUEUR :

`length (str) -> entier`

Exemple 66

```
var
S: String;
begin
Readln (S);
Writeln('"', S,'"');
Writeln('longueur de la chaîne = ', length(S));
end.
```

Exemple 67

```
s1:='salut';
s2:='bonjour';
```

Nous obtenons alors `length(s1)` *valant* `5` *et* `length(s2)` *valant* `7`

FONCTION POS

`pos(souschaîne, chaîne)`

$\Longrightarrow$position de la sous chaîne dans la chaîne.

Exemple 68

```
var S: String;
begin
S := ' 123.5';
{ Convertit les espaces en zéros }
while Pos(' ', S) > 0 do
S[Pos(' ', S)] := '0';
end.
```

FONCTION COPY :

`copy (source, index, compteur)`

$\Longrightarrow$ `string` avec "`compteur`" caractères à partir de l'index.

Exemple 69

```
s:=copy('bonjour monsieur', 4, 4);
```

Nous obtenons alors `s` *valant* `'jour'`

PROCEDURE DELETE :

`delete(chaine, debut, nb_car)`

$\Longrightarrow$ supprime le nombre de caractères spécifié par `nb_car` à partir de la position indiquée par `debut`.

PROCEDURE INSERT :

`insert(chaine1, chaine2, position)`

$\Longrightarrow$ insère `chaine1` dans `chaine2` à partir de la position indiquée par position.

Exemple 70

```
s:=insert('madame ', 'au revoir  Fall' ,11)
```

Nous obtenons alors `s` *valant* `'au revoir madame Fall'`

FONCTION ORD

`ORD(caractère)`

$\Longrightarrow$ entier (code ASCII).

Exemple 71 `ORD('A')` *vaut* `65` *et* `ORD('a')` *vaut* `97`

FONCTION CHR

`CHR(entier)`

$\Longrightarrow$ caractère ayant ce code ASCII.

Exemple 72 `CHR(65)` *vaut* `'A'` *et* `CHR(97)` *vaut* `'a'`

CHAPITRE 5

Les sous programmes

5.1 La programmation modulaire

5.1.1 Définition

Si vous avez un gros programme à mettre au point, et que certaines parties sont semblables ou d'autres très complexes, alors il faut absolument structurer son programme pour ne pas risquer d'y passer trop de temps, lors de modifications ultérieures, ce qui en facilitera la maintenance.
Conclusion : il faut adopter la stratégie *"diviser pour régner"* et chercher à utiliser au maximum "l'existant".

La programmation modulaire consiste à décomposer le problème initial en sous-problèmes de moindre complexité. Cette décomposition se poursuit jusqu'à l'obtention de sous-problèmes plus compréhensibles et plus simples à programmer. On parle d'analyse descendante (ie, on part du niveau le plus complexe et on aboutit à un niveau le plus simple possible du problème).

Par ailleurs on est souvent amené à effectuer plusieurs fois un même ensemble de traitements dans un programme. Cet ensemble de traitements constitue un sous-programme1 (ou module). Cependant pour éviter cette réécriture, on attribue un nom à ce dernier et à chaque fois que son utilisation est souhaitée, on lui fait appel par l'intermédiaire de ce nom.

Avantages :

- la lisibilité
- la structuration
- possibilité d'utiliser un identificateur significatif

Exemple 73 *au lieu d'écrire :*

```
begin
    bloc1
    bloc2
    ...
    bloc1
    ...
    bloc2
    bloc1
    ...
end.
```

où `bloc1` *et* `bloc 2` *peuvent être des blocs de plusieurs dizaines de lignes de code, on préfèrera écrire :*

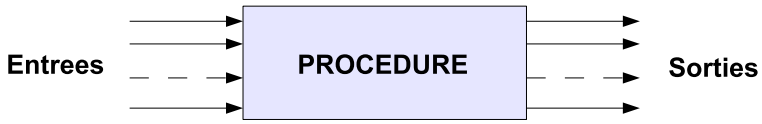
```
procedure P1
begin
    bloc1
end;
procedure P2
begin
    bloc2
end;
begin
    P1;
    P2;
    P1;
    ....
    P2;
    P1;
```

```
    ...
end.
```

5.1.2 Les différents types de modules

En Pascal, on distingue deux types de modules :

Les **procédures** : Il s'agit de sous-programmes *nommés* qui représentent une ou plusieurs actions, et qui peuvent calculer et ***retourner une ou plusieurs valeurs***. Une procédure `P` doit être définie une seule fois dans un algorithme `A` et peut être appelée plusieurs fois dans `A` ou par une autre procédure de `A`.



Les Fonctions : Il s'agit de sous programmes *nommés* et *typés* qui calculent et ***retournent une et une seule valeur***. Le type de cette valeur est celui de la fonction. De même une fonction `F` doit être définie une seule fois dans l'algorithme `A` et peut être appelée plusieurs fois par `A` et par toutes les autres procédures et fonctions de `A`.

5.2 Les Procédures

5.3 Syntaxe et déclarations

Si l'on se réfère donc au chapitre sur les déclarations, les déclarations s'effectuent dans l'ordre suivant :

```
EN-TETE
```

```
 DECLARATIONS
1. CONSTANTES
2. TYPES
3. VARIABLES
4. FONCTIONS / PROCEDURES

BLOC D'INSTRUCTIONS EXECUTABLES
```

Syntaxe de déclaration La définition d'une procédure comprend trois parties :

Partie 1 : *L'entête* :

```
Procedure Nom_proc[(Identif1[, ...]:Type1[;[var] Identif2[, ...]:Type2])];
```

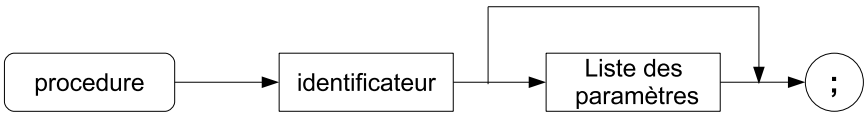


Figure 5.1 – Diagramme de description syntaxique de l'entête d'une procédure

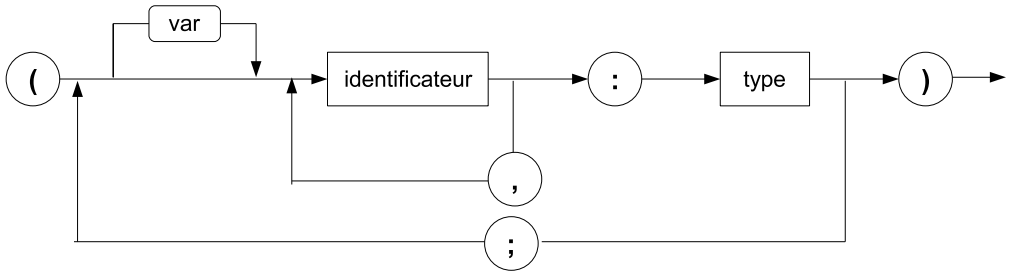


Figure 5.2 – Diagramme de description syntaxique de la liste de paramètres

Partie 2 : *Déclaration des variables locales*

```
Nom_variable_i : type_variable_i;
```

Partie 3 : *Corps de la procédure*

```
Begin
     <instruction1>;
     <instruction2>;
```

```
        ...
        <instructionN>
    End;
```

➥ Les identificateurs qui suivent le nom de la procédure constituent l'ensemble des *paramètres formels*.

➥ Le mot var (optionnel) n'est utilisé que pour les paramètres de sortie et d'entrée/sortie.

Une fois la procédure déclarée, elle peut être utilisée dans le programme principal par un "appel" à cette procédure, à partir du programme principal ou à partir d'une autre procédure.

Appel d'une procédure

```
Nom_procedure( expression { , expression } ) ;
```

5.3.1 Exemples

Exemple 74 *Ecrire un programme qui calcule successivement la moyenne de la taille de trois individus ainsi que la moyenne de leur poids et celle de leur âge.*

Version sans emploi de procédure (à rejeter) :

```
Program Moyenne;
    Var  A, B, C, M : real;
    Begin
        writeln ('Entrez la taille des 3 individus : ');
        readln (A, B, C);
        M := (A+B+C) / 3;
        writeln ('La moyenne est : ', M:8:2);
        writeln('Entrez le poids des 3 individus : ');
        readln (A, B, C);
        M := (A+B+C) / 3;
        writeln ('La moyenne est : ', M:8:2);
        writeln('Entrez l''âge des 3 individus : ');
        readln (A, B, C);
        M := (A+B+C) / 3;
        writeln ('La moyenne est : ', M:8:2);
  End;
```

Cette version n'est pas idéale, car nous constatons qu'une même partie du programme est écrite trois fois. Il est préférable de regrouper ces instructions dans une procédure. La nouvelle version, avec emploi d'une procédure, est la suivante :

```
Program  MOYENNE_BIS;
Var    A, B, C, M : real;
Procedure  CALCUL;
Begin
      readln (A, B, C);
      M := (A + B + C) / 3;
      writeln ('La moyenne est ', M:8:2);
End;
Begin  { Début du programme principal }
      writeln ('Entrer les tailles de 3 individus');
      CALCUL;
      writeln ('Entrer les poids de 3 individus');
      CALCUL;
      writeln('Entrez l''âge des 3 individus : ');
     CALCUL;
End.
```

Exemple 75 *Pour calculer la somme de deux matrices A et B, il faut lire d'abord ces deux matrices. Ainsi au lieu d'écrire deux fois le sous-programme de lecture d'une matrice, on écrit plutôt une procédure et on l'utilise pour saisir A et B.*

```
Program  SOMME_MATRICIELLE;
Const nmax=50;
Type MATRICE = ARRAY[1..nmax,1..nmax] of integer;
Var    A, B, S : MATRICE;
                  n : integer;
{Procédure de saisie d'une matrice }
Procedure  SAISIE(Var M : MATRICE; dim : integer);
```

```
Var i, j : integer;
Begin
      For i:=1 To dim  Do
          For j:=1 To dim  Do
                Begin
                     write('Donnez l''élément M[',i,j,'] ');
                     readln(M[i,j]);
                End;
End;
{Procédure de calcul de la somme de deux matrices }
Procedure  SOMME(M1  : MATRICE;M2 : MATRICE; var C : MATRICE , dim : integer);
Var i, j : integer;
Begin
      For i:=1 To dim  Do
          For j:=1 To dim  Do
                 C[i,j]:=M1[i,j]+M2[i,j];
End;

{Procédure d'affichage d'une matrice }
Procedure  AFFICHAGE(M : MATRICE; dim : integer);
Var i, j : integer;
Begin
      For i:=1 To dim  Do
         Begin
             For j:=1 To dim  Do
                     Write(M[i,j],' ');
             writeln;
         End;
End;

Begin  { Début du programme principal }
      write ('Entrer la dimension des deux matrices ');
      readln(n);
      writeln (' Lecture de la première matrice ');
```

```
      SAISIE(A,n);
      writeln (' Lecture de la seconde matrice ');
      SAISIE(B,n);
      SOMME(A,B,S,n); {Calcul de la somme de A et B}
      AFFICHAGE(S,n);{affichage du résultat}
      readln;
End.
```

5.4 Fonctions

5.4.1 Utilité des fonctions

D'une manière générale :

- Les procédures ont pour effet de modifier l'état des variables d'un programme. Elles constituent une partie du programme qui se suffit à elle-même. Une procédure peut ne pas retourner des résultats au (sous-) programme appelant. En effet, il peut arriver qu'une procédure exécute un certain nombre d'instructions et que l'on n'ait pas besoin de la valeur résultat (cas de la fonction CALCUL dans l'exemple précédent).
- Les fonctions permettent de définir le calcul d'une valeur d'un type donné et de rendre un résultat au (sous-) programme. Le type de la valeur retournée est très souvent un type de données simple.

On distingue deux catégories de fonction en Turbo-Pascal :

a) **Les fonctions implicites**.

Parmi les fonctions implicites, encore appelées primitives de base du système de programmation on distingue :

- les fonctions usuelles représentées par les opérateurs et,
- les fonctions standard prédéfinies telles que `Sqr`, `Sqrt`, `Sin`, etc. mises à la disposition de l'utilisateur par le système de programmation utilisé.

b) **Les fonctions explicites**.

les fonctions explicites regroupent toutes les autres fonctions calculables à définir par l'utilisateur lui-même. En générale, ces fonctions sont de la forme :

`F(X1,X2,...,XN)`

où `F` est le nom de la fonction et `X1,X2,...,XN` les arguments

5.4.2 Définition

Une fonction est un bloc d'instructions qui doit retourner une valeur de type simple au point d'appel. En général, le rôle d'une fonction est le même que celui d'une procédure. Il y a cependant quelques différences :

- Arguments de la fonction : paramètres typés
- Un ensemble d'arrivée typé
- Renvoi d'une valeur unique

Type de la fonction :

- scalaire, réel, string
- intervalle
- pointeur

Le type doit être précisé par un identificateur, et non par une déclaration explicite.

On déclare les fonctions au même niveau que les procédures (après VAR, TYPE, CONST)

Syntaxe de déclaration :

La définition d'une fonction comprend aussi trois parties :

Partie 1 : *L'entête*

```
Function Nom_fonc[(Ident1[,...]:Type1[;[var] Ident2[,...]:Type2])]:Typfonc;
```

Partie 2 : Déclaration des variables locales

```
Nom_variable_i : type_variable_i;
```

Partie 3 : *Corps de la fonction.*

```
Begin
        <instruction1>;
        <instruction2>;
        ...
        nom_fonction:=<expression> {valeur retournée}
        ...
        <instructionN>
End;
```

➥ L'affectation `nom_fonction:=<expression>` ne peut se faire qu'une seule fois par appel à la fonction.

- ➥ La principale différence avec une procédure est que la fonction renvoie toujours une valeur (un résultat) comme après son appel, d'où la nécessité de déclarer un type pour cette fonction (Toute fonction « vaut »quelque chose.
- ➥ L'identificateur de fonction est donc considéré comme une expression, c'est-à-dire comme une valeur (calculée par la fonction elle-même).

Ainsi, le simple fait d'écrire l'identificateur de fonction avec ses paramètres a pour conséquence d'appeler la fonction, d'effectuer le traitement contenu dans cette fonction, puis de placer le résultat dans l'identificateur de fonction.

Exemple 76

```
Ecrire un programme qui calcule la puissance d'un nombre réel.
program  EXPOSANT ;
var
      N : integer;
      Y, T : real;
function  PUISSANCE (X : real; N : integer) : real;
    begin
         Y := 1;
         if  N > 0  then
                      for  I := 1 to  N  do
                           Y := Y * X
                  else
                      for  I := N  to  -1  do
                           Y := Y / X;
         PUISSANCE := Y; { affectation de la valeur à l'identificateur}
    end;   { Fin du code de la fonction }

    { Debut du programme principal}
begin
      readln (T, N);     { appel de la fonction dans le writeln }
      writeln('La valeur de ', T,' puissance ', N,' est : ', PUISSANCE(T,N));
end.     { Fin du programme }
```

5.5 Différence entre procédure et fonction

- ➥ Une procédure peut être considérée comme une instruction composée que l'utilisateur aurait créée lui-même. On peut la considérer comme un **petit programme**.
- ➥ Une fonction quant à elle renvoie **toujours une « valeur »**(comprendre le mot valeur comme objet de type simple, comme par exemple un entier ou un caractère). Elle nécessite donc un type (entier, caractère, booléen, réel, etc...).

NB : Il est interdit d'utiliser l'identificateur d'une fonction comme nom de variable en dehors du bloc correspondant à sa déclaration.

Exemple 77 *Examinons le programme suivant*

```
program exemple;
var x,y : integer;
function double (z : integer) : integer;
    begin
        double := z*2;
    end;
begin
        Readln(x);
        y := double(x);
        double := 8; {erreur à cette ligne lors de la compilation}
 end.
```

Ce programme ne pourra pas fonctionner, car on lui demande d'affecter la valeur `8` *à la fonction* `double`*. Or, Il est interdit d'utiliser l'identificateur d'une fonction comme nom de variable en dehors du bloc correspondant à sa déclaration, d'où l'ERREUR.*

```
        PROGRAM difference;
VAR a, b, c, d : real;
PROCEDURE Produit (x, y : real; | FUNCTION Produit (x, y : real) : real;
                 var z : real); | VAR res : real;
BEGIN                           | BEGIN
   z := x * y;                  |    res := x * y;
END;                            |    Produit := res;
```

```
                                | END;
                        BEGIN
                             write ('a b ? '); readln (a, b);
Produit (a, b, c);              | c := Produit (a, b);
Produit (a-1, b+1, d);          | d := Produit (a-1, b+1);
                             writeln ('c = ', c, ' d = ', d);
                        END.
```

5.6 Variables globales et variables locales

5.6.1 Déclarations

Dans certains exemples présentés dans ce cours, nous avons effectué toutes les déclarations de variables en tête de programme. Or il est également possible (comme nous l'avons fait dans d'autres circonstances) de déclarer des variables, **au sein d'un bloc fonction ou procédure**. Dans ce cas, ces déclarations se font dans le même ordre : constantes, types, variables (et même procédure(s) et/ou fonction(s)). Lorsque la déclaration est effectuée en en-tête du programme, on parle de **variable globale**. Dans le cas contraire, où la déclaration est faite à l'intérieur même de la procédure ou de la fonction, on parle de **variable locale**.

Exemple 78 *Reprenons le programme qui calcule la puissance d'un nombre réel. Nous désirons de plus calculer la puissance T^N avec des exposants (N) variant de $-N$ à $+N$, N étant le nombre entier entré par l'utilisateur. Par exemple, si on tape 3 pour T et 2 pour N, le programme calculera les valeurs 3^{-2}, 3^{-1}, 3^0, 3^1, et 3^2 Cette fois-ci, nous utiliserons des variables locales, ce qui est bien meilleur*

```
Program  EXPOSANT ;
var   I, N : integer;
         T : real;
Function  PUISSANCE (X : real; N : integer) : real;
var               { utilisation de variables locales }
```

```
      Y : real;
      I : integer;
Begin    { Code de la fonction }
     Y := 1;
     if  N > 0  then
                  for  I := 1 to  N  do
                            Y := Y * X
                else
                  for  I := -1  downto  N  do
                            Y := Y / X;
     PUISSANCE := Y;
end;          { Fin du code de la fonction }
{ Debut du programme principal}
begin
      readln (T, N);
      for I := -N to N do
            writeln (PUISSANCE (T, I));  { appel de la fonction }
end.
```

5.6.2 Portée des variables

- **Variable locale** : portée limitée à la procédure ou à la fonction en question
- **Variable globale** : active dans le bloc principal du programme, mais également dans toutes les procédures et fonctions du programme.

  Chaque variable située dans un bloc de niveau supérieur est active dans toutes les procédures de niveau inférieur.

Dans le cas où un même nom est utilisé pour une variable globale et pour une variable locale, cette dernière à la priorité dans la procédure ou dans la fonction où elle est déclarée (niveau local).

ATTENTION : Dans le cas où un même nom est utilisé pour une variable locale et pour une variable globale, le programme considère ces variables comme **deux variables différentes** (mais ayant le même nom).

Exemple 79

```
program PORTEE;
    var
        A, X : real;
    procedure INTERNE;
    var
        B, X : real;
    begin
      B := A / 2;
      writeln (B);
      writeln (A + X);
    end;
    begin { programme principal }
      readln (A, X);
      writeln (A / 2);
      writeln(A + X);
      INTERNE;
    end.
```

Question : que se passe-t-il à l'exécution ?

Réponse : le programme ne fonctionnera pas comme on l'espérait, car `X` est considérée comme une variable locale dans `INTERNE`, et cette variable locale n'a pas été initialisée ! Dans la procédure `INTERNE`, `X` n'aura donc aucune valeur (il ne faut surtout pas la confondre avec le `X` du programme principal, qui est une variable globale).

Pour que le programme fonctionne correctement, il faudra alors ne pas déclarer `X` comme locale dans `INTERNE`.

5.6.3 Locale ou globale ?

Il vaut toujours mieux privilégier les variables locales aux variables globales.

- *Inconvénients d'une utilisation systématique de variables globales* :
  - manque de lisibilité
  - présence de trop nombreuses variables au même niveau
  - ne facilite pas la récursivité
  - risque d'effets de bord si la procédure modifie les variables globales
- *Avantages d'une utilisation de variables locales* :

- meilleure structuration
- diminution des erreurs de programmation
- les variables locales servent de variables intermédiaires (tampon) et sont « oubliées »(effacées de la mémoire) à la sortie de la procédure.

DÉFINITION

```
Procedure NIVEAU_SUP;
VAR X, Y, ...

Procedure NIVEAU_INF;
var X,Z,...
begin
    X:=...
    Y:=...
    Z:=...
end;

begin
end.
```

- La variable `X` déclarée dans `NIVEAU_SUP` est locale à cette procédure.
- Dans la procédure `NIVEAU-INF`, cette variable est occultée par l'autre variable `X`, déclarée encore plus localement.
- La portée de `Y` est celle de toute la procédure `NIVEAU-SUP`, sans restriction (elle est locale à `NIVEAU-SUP`, et donc globale dans `NIVEAU-INF`).
- En revanche, `Z` a une portée limitée à la procédure `NIVEAU-INF` (elle est locale à `NIVEAU-INF`, et ne peut être utilisée dans `NIVEAU-SUP`).

Exemple 80 *Le programme suivant utilise une procédure pour calculer le double d'un nombre entier. On a ici un exemple d'imbrication d'une fonction dans une procédure.*

```
program test;
   var  i,j : integer;  { variables globales }
   procedure INITIALISE;
   begin
        i := 0;
```

```
     j := 0;
end;
procedure CALCUL;
    function DOUBLE (i : integer) : integer;
    begin
        double := 2 * i;
    end;
begin
     i := 3;
     j := DOUBLE(i);
end;
Begin
     INITIALISE;
     readln(i,j);
     CALCUL;
     writeln(i,j);
End.
```

Comme il n'y a pas de déclarations locales de `i` *et* `j` *dans les procédures et la fonction mises en jeu, les variables* `i` *et* `j` *globales seront utilisées pour les calculs.*

IMPORTANT : Bien que tolérée, l'utilisation de variables globales est fortement déconseillée (risques de modifications de valeur non souhaitées) ; préférer les passages de paramètres. Si toutefois des variables globales doivent être utilisées, leur donner des noms suffisamment évocateurs pour éviter le risque de confusion.

Les règles à retenir sont les suivantes :

1. une variable est globale à un (ensemble de) bloc dès lors qu'elle est déclarée dans un bloc les emboîtant ; elle est visible dans le bloc où elle est déclarée et dans tous les blocs internes ; sa durée de vie est celle du bloc où elle est déclarée.
2. une variable est locale à un bloc si elle est déclarée dans ce bloc ; elle est visible seulement dans le bloc où elle est déclarée ; sa durée de vie est celle du bloc où elle est déclarée. une déclaration locale masque toujours une déclaration plus globale (si une déclaration locale porte le même nom qu'une déclaration plus globale, c'est le nom local qui l'emporte).

3. une variable déclarée localement n'existe que pendant l'exécution de la procédure, et ne sert que à cette procédure.
4. le programme principal n'a jamais accès à une variable locale de procédure.
5. une procédure n'a jamais accès à une variable locale d'une autre procédure.
6. les règles pour les paramètres sont les mêmes que pour les variables locales.

Exemple 81 *Programme avec procédures non imbriquées*

```
Program VisibiliteDesDeclarations ;
Const Pi = 3.14159 ;
Var Global : Real ;
Procedure Visibilite ( Var X : Real ) ;
Var  Pi : Real ;          { Local , masque la constante }
     T : tTableau ;       { Erreur : tTableau  inconnu car non encore déclaré }
Begin
      Pi := 3.14 ;
      X := Pi ;
      Global := Global - Pi
End ;
Type tTableau = Array  [ 1..5000 ] Of Real ;
Var R : Real ;
Begin { Du programme principal }
     Global := Pi ;
     Visibilite ( R ) ;
     Writeln ( X ) ;   { Erreur : X ( paramètre ) est local }
     Writeln ( R : 8 : 5  , '  ' , Global : 8 : 5 ) ;
End .
Après correction des 2 erreurs, ce programme fournit comme résultat :
3.14000    0.00159
```

Exemple 82 *Programme avec procédures emboîtées*

```
Program ProceduresEmboitees ;
```

```
Var  G : Real ;
     A : Integer ;
     Procedure Proc1 ;
         Procedure Proc1Bis ;
         Var A : Real ;
         Begin              { de Proc1Bis }
           A := 9.81 ;
           Writeln ( A : 5 : 2 , '  ' , G : 5 : 2 )
         End ;               { de Proc1Bis }
     Begin              { de Proc1 }
       Proc2 ;          { Interdit car non encore définie }
       Proc1Bis ;
       Writeln ( A : 5 , '  ' , G : 5 : 2 )
     End ;             { de Proc1 }
     Procedure Proc2 ;
     Begin              { de Proc2 }
       Proc1Bis ;    { Interdit car inaccessible }
       G := 10.5 ;
       Writeln ( A : 5 , '  ' , G : 5 : 2 )
     End ;
Begin    { du programme principal }
     G := 3.14 ;
      A := -5 ;
   Proc1 ;
      Proc2 ;
      Writeln ( A : 5 , '  ' , G : 5 : 2 ) ;
End .
```

Après correction des 2 erreurs, ce programme fournit comme résultat :

```
    9.81 3.14
      -5 3.14
      -5 10.5
      -5 10.5
```

5.7 Paramètres

5.7.1 Définition

Problématique : Exemple

Supposons qu'on ait à résoudre l'équation du second degré en divers points du programme :

- la première fois $Rx^2 + Sx + T = 0$,
- la deuxième fois $Mx^2 + Nx + P = 0$,
- la troisième fois $Ux^2 + Vx + W = 0$.

Comment faire en sorte qu'une ***même procédure*** puisse les traiter toutes les trois, c'est-à-dire ***travailler sur des données différentes*** ?

➥ 1ère possibilité : utiliser des variables globales `A`, `B`, `C` pour exprimer les instructions dans la procédure, et, avant l'appel de la procédure, faire exécuter des instructions telles que :
`A := R, B := S, C := T`, etc.
problème des variables globales, multiplication des affectations, etc ... . Solution à écarter !

➥ 2ème possibilité : définir une procédure de résolution d'une équation du second degré avec une liste d'arguments, et transmettre les données à l'aide de paramètres. La déclarations sera :
`procedure SECOND_DEGRE(A,B,C:integer);` à l'appel, on écrira :

```
SECOND_DEGRE (M, N, P);   { appel avec les valeurs de M, N et P }
  SECOND_DEGRE (R, S, T);    { appel avec les valeurs de R, S et T }
```

Lors de la déclaration de la procédure, donner une liste d'arguments (paramètres) signifie une transmission de l'information entre le programme principal et le sous-programme

Il faut préciser le type des arguments par un identificateur et par une déclaration de type.

Exemple 83 `Procedure OK (x,y : real);` *Déclaration correcte*
préférer : `Procedure OK (tab : tableau);`

à :

```
Procedure OK (tab : array[1..100] of integer);
```

5.7.2 Paramètres formels et paramètres effectifs

Dans la déclaration de la procédure `SECOND_DEGRE` , les paramètres `A`, `B` et `C` sont appelés paramètres formels dans la mesure où ils ne possèdent pas encore de valeurs. Lors de l'utilisation de la procédure dans le programme principal (donc à l'appel) `M`, `N` et `P` sont des paramètres effectifs (ou réels).

Lors de l'appel de la procédure, il y a remplacement de chaque paramètre formel pa un paramètre effectif, bien spécifié.

Soit la déclaration de procédure suivante :

Exemple 84 `Procedure SECOND_DEGRE (A, B, C : integer);`

Lors de l'appel de la procédure, si on écrit :

```
SECOND_DEGRE (M, N, P);
```

A l'entrée du bloc de la procédure, `A` *prendra la valeur de* `M`*,* `B` *prendra celle de* `N` *et la variable* `C` *sera affectée de la valeur de* `P`*.*

Dans une procédure, le nombre de paramètres formels est exactement égal au nombre de paramètres effectifs. De même à chaque paramètre formel doit correspondre un paramètre effectif de **même type**.

Un paramètre formel est toujours une variable locale et ne peut être utilisé en dehors de la procédure (ou de la fonction) où il est défini. En revanche un paramètre effectif peut être une variable globale.

En Pascal, lors de la déclaration d'une procédure, il est possible de choisir entre deux modes de transmission de paramètres :

- passage par valeur et
- passage par adresse ;

5.7.3 Passage de paramètre par valeur

Rappelons d'abord que :

Déclarer une variable de nom `X` $\Longrightarrow$ réserver un emplacement mémoire pour `X`.

Ainsi lors de l'appel d'une procédure un emplacement mémoire est réservé pour chaque paramètre formel. De même un emplacement mémoire est aussi réservé pour chaque paramètre effectif lors de sa déclaration. Cependant lors de l'appel, **les valeurs des paramètres effectifs sont copiées dans les paramètres formels**. Ainsi l'exécution des instructions de la procédure se fait avec les valeurs des paramètres formels et toute modification sur ces dernières ne peut affecter en aucun cas celles paramètres effectifs. Dans ce cas on parle de passage par valeur. C'est le cas dans la procédure `SAISIE` (paragraphe 5.3.1) du paramètre `dim` et dans la procédure `AFFICHAGE` (paragraphe 5.3.1) des paramètres `M` et `dim`. Notons que dans ce type de passage, les valeurs des paramètres effectifs sont connues avant le début de l'exécution de la procédure et jouent le rôle uniquement d'entrées de la procédure.

Exemple 85 `procedure ID_PROC (X, Y : real; I : integer; TEST: boolean);`

Points importants :

- Les paramètres formels sont des variables locales à la procédure, qui reçoivent comme valeur initiale celles passées lors de l'appel On parle alors d'`AFFECTATION`. Exemple : `ID_PROC (A, B, 5, true);` `X` a alors pour valeur initiale celle de `A`, `Y` celle de `B`. `I` a pour valeur initiale `5`, et `TEST true`
- Le traitement effectué dans la procédure, quel qu'il soit, ne pourra modifier la valeur des paramètres effectifs. Exemple : après exécution de `ID_PROC`, `A` et `B` auront toujours la même valeur qu'auparavant, même s'il y a eu des changements pour ces variables, dans la procédures.
- Le paramètre spécifié lors de l'appel peut être une expression.
  Ainsi, `ID_PROC (3 / 5, 7 div E, trunc (P), true)` ; est un appel correct.

Exemple 86 *Nous cherchons à écrire un programme qui échange les valeurs de deux variables saisies par l'utilisateur.*

```
program  TEST;
var
    A, B : real;
```

```
procedure  ECHANGE (X, Y : REAL);
var
    T : real;
begin
     T := X;
     X := Y;
     Y := T;
     Writeln(X,Y);
end;
Begin
     readln (A, B);
     ECHANGE (A, B);
     writeln (A, B);
End.
```

Cette solution est mauvaise. En effet, une simulation de son exécution donnera :

`A = 5` `B = 7` *(saisie)*

`X = 7` `Y = 5`

`A = 5` `B = 7` *(*`A` *et* `B` *restent inchangés !)*

Le résultat de l'action accomplie par la procédure n'est pas transmis au programme appelant, donc `A` *et* ***B*** *ne sont pas modifiés.*

La Transmission est unilatérale.

Avantage : grande liberté d'expression dans l'écriture des paramètres effectifs. Cela évite les erreurs et les effets de bord (que nous verrons plus loin).

Tout paramètre est utilisé, pour passer des valeurs dans la procédure, mais n'est jamais modifié.

Si on désire récupérer l'éventuelle modification du paramètre, il faut alors utiliser un autre procédé, décrit dans la section suivante (passage par adresse).

Exemple 87 *Ceci est un programme assez qui ne fait pas grand chose et dont le seul intérêt est d'illustrer le passage de paramètres ainsi que la notion d'effet de bord.*

```
program Effet-de_bord;
var
       i,j : integer;
```

```
 function double (i : integer) : integer;
    begin
          double := 2 * i;
    end;
 function plus_un (j : integer) : integer;
    var
        i: integer;
    begin
          i := 10;
          plus_un := j + 1;
    end;
 function moins_un (j : integer) : integer;
    begin
          i := 10;
          moins_un := j - 1;
    end;
Begin     {programme principal}
     i := 0; j := 0;      {i=0 ; j=0}
     j := plus_un(i);     {i=0 ; j=1}
     j := double(j);      {i=0 ; j=2}
     j := moins_un(j);    {i=10 ; j=1}
End.
```

La variable `i` *a été modifiée dans la procédure, alors que l'on s'attendait à des modifications sur* `j`*. On appelle cela un effet de bord. Un tel phénomène peut être très* ***dangereux*** *(difficulté à retrouver les erreurs).*

Efet de bord Voici le scénario catastrophe à éviter

- On est dans une procédure `P` et on veut modifier une variable `x` locale à `P`.
- Il existe déjà une variable globale ayant le même nom `x`.
- On oublie de déclarer la variable locale `x` au niveau de `P`.
- A la compilation tout va bien !
- A l'exécution, `P` modifie le `x` global alors que le programmeur ne l'avait pas voulu.
- Conséquence : le programme ne fait pas ce qu'on voulait, le `x` global a l'air de changer de valeur tout seul !

$\Longrightarrow$ Erreur très difficile à détecter ; être très rigoureux et prudent !

5.7.4 Passage de paramètre par adresse

En pascal, la différence principale entre le passage par valeur et le passage par adresse c'est que dans ce dernier, un seul emplacement mémoire est réservé pour le paramètre formel et le paramètre effectif correspondant. Autrement dit, dans ce cas chaque paramètre formel de la procédure utilise directement l'emplacement mémoire du paramètre effectif correspondant. Par conséquent toute modification du paramètre formel entraîne la même modification du paramètre effectif correspondant. Soulignons que dans ce mode de passage de paramètres, les valeurs des paramètres effectifs peuvent être inconnues au début de l'exécution de la procédure. Cependant un paramètre formel utilisant ce type de passage ne peut être que le résultat de la procédure.

Pour spécifier dans une procédure qu'il s'agit du mode par adresse, il suffit, lors de la déclaration de la procédure, d'ajouter le mot clé `VAR` devant la déclaration du paramètre concerné. C'est le cas par exemple du paramètre `S` dans la procédure `SOMME` de l'exemple du paragraphe 5.3.1.

Exemple 88 `procedure ID_PROC (var X, Y : real; Z : integer);`

Points importants :

- Lors de l'appel, des paramètres réels sont substitués aux paramètres formels : **SUBSTITUTION**
- Seule une variable peut être substituée aux paramètres réels, il est impossible de faire l'appel avec une constante ou une expression évaluable !

  Exemple 89 `ID_PROC (U, V, 7);` *est correct.*

  `ID_PROC (4, A - B, 8);` *est tout à fait incorrect.*

- Tout changement sur le paramètre formel variable change aussi le paramètre effectif spécifié lors de l'appel.

Exemple 90
```
program essai;
var
      i : integer;
procedure double (x : integer ; var res : integer);
begin
```

```
      res := 2 * x;
end;
Begin
     i := 1;          {i=1}
     double (5,i);    {i=10}
End;
```

Dans cet exemple, `x` *est un paramètre transmis par valeur (donc non variable), alors que res est un paramètre passé par adresse (donc considéré comme variable).*

Principe : s'il y a nécessité de renvoyer les modifications au programme appelant, on emploi des paramètres variables.

Exemple 91 *Reprenons et corrigeons le programme qui échange les valeurs de deux variables saisies par l'utilisateur.*

```
program  TEST_BIS;
var
    A, B : real;
procedure  ECHANGE (var  X, Y : real);
var
    T : real;
begin
     T := X;
     X := Y;
     Y := T;
     writeln(X,Y);
end;
begin
     readln (A, B);
     ECHANGE (A, B);
     writeln (A, B);
end.
```

Une simulation du déroulement du programme donnera :

`A =  5     B = 7` *(saisie)*

`X = 7` `Y = 5`

`A = 7` `B = 5` *(*`A` *et* `B` *ont été modifiés !)*

Le résultat de l'action de la procédure a été transmis au programme appelant !

5.7.5 Bon réflexes

Le seul moyen pour une procédure de communiquer avec l'extérieur, c'est à dire avec le reste du programme, ce sont les variables globales et les paramètres.

Il faut toujours éviter soigneusement les effets de bords. Le meilleur moyen est de paramètrer compllétement les procédures, et d'éviter la communication par variables globales.

Les variables de travail tels que compteur, somme partielle, etc doivent être locales à la procédure, surtout pas globale.

Prendre l'habitude de prendre des noms de variables différents entre le programme principal et les procédures : on détecte plus facilement à la compilation les effets de bords.

Chaque fois que l'on appelle une procédure, on vérifie particuliérement le bon ordre des paramètres et la correspondance des types. La compilation est trés pointilleuse sur les types, mais par contre elle ne détecte pas les inversions de paramètres de même type

5.7.6 Cas des fonctions

Pour les fonctions, on peut agir de même. Le principe est strictement analogue à celui des procédures. On distingue donc, là encore, les passages de paramètres par valeur des passages de paramètres par adresse (ou variables).

Exemple 92
```
function LETTRE (c : char) : boolean;
  begin
      if (c in ['A'..'Z']) or (c in ['a'..'z'])  then
                              lettre := true
                                  else
                                 lettre := false;
    end;
function MAJ (var c : char) : boolean;
```

```
begin
  if not LETTRE(c)  then
                  maj : false
              else
                   begin
                       if c in ['a'..'z'] then
                                begin
                                    c := chr (ord(c) - ord('a') + ord('A'));
                                    maj := true;
                                end
                                  else
                                            maj := false;
                     end;
  end;
```

Ce programme résume ce que nous avons pu voir avec les procédures. La première fonction utilise un passage de paramètre par valeur car nous n'avons pas besoin de modifier ce paramètre. Cette fonction est utilisée pour tester si un caractère est une lettre (minuscule ou majuscule).

La fonction `MAJ` *(qui utilise la fonction précédente) modifie un caractère qui est une lettre minuscule en sa majuscule correspondante.*