CHAPITRE 7

LES PROCÉDURES ET LES FONCTIONS

OBJECTIFS

- EXPOSER LE PRINCIPE DE LA DÉCOMPOSITION DES PROGRAMMES DANS LE LANGAGE PASCAL.
- PRÉCISER LE RÔLE ET L'UTILITÉ DES DIFFÉRENTS TYPES DE PROCÉDURE ET DE FONCTION DANS UN CONTEXTE DE PROGRAMMATION MODULAIRE.

Nous avons étudié au chapitre premier différentes approches de résolution de problème, pour lesquelles nous avons fait ressortir la nécessité de décomposer les problèmes à résoudre en sous-problèmes ou modules. Chaque module est alors traité séparément et l'intégration des différents modules conduit à la solution globale du problème. Les termes *sous-programme* et *routine* sont souvent utilisés pour désigner la codification de tels modules. Dans le langage PASCAL, on parle plutôt de *procédure* et de *fonction*. Ce chapitre présente ces deux notions qui permettent, somme toute, d'assurer la modularité des programmes écrits dans ce langage.

7.1 LES PROCÉDURES

On désigne par *procédure* une séquence d'instructions de programme à laquelle on associe un nom spécifique représenté par un identificateur. Dans les sections qui suivent, nous présentons quelques généralités sur cette notion, puis élaborons sur les deux types de procédures définis dans le langage PASCAL : les procédures sans paramètres et les procédures avec paramètres.

7.1.1 Généralités

On distingue deux catégories de procédures : les procédures standards et les procédures non standards. Par *procédure standard*, on entend toute procédure connue du langage, donc qui se passe d'une déclaration préalable lors de son utilisation dans un programme. C'est le cas des instructions de lecture et d'écriture READ et WRITE et de leurs variantes READLN et WRITELN.

Par *procédure non standard*, on entend toute procédure non connue du langage et qui, par conséquent, nécessite une définition préalable à son utilisation. C'est précisément cette catégorie de procédures qui retient notre attention dans ce chapitre.

Si on considère une procédure comme un bloc indépendant auquel on attribue un nom, il est pratique, une fois testé, de réutiliser ce bloc à plusieurs reprises, évitant ainsi la duplication d'une même partie de programme. Cela constitue un avantage majeur qui facilite l'application et la mise en œuvre du concept de modularité servant de base à la programmation structurée. De plus, une même procédure peut être utilisée dans plusieurs programmes différents, ce qui permet d'économiser aussi bien du temps de saisie que de l'espace en mémoire.

On distingue deux étapes fondamentales dans le traitement d'une procédure : sa déclaration et son activation. La *déclaration d'une procédure* est l'opération par laquelle on définit, dans la section appropriée du programme, un en-tête et un corps qui spécifient cette procédure. La figure 7.1 présente le diagramme syntaxique d'une déclaration de procédure.

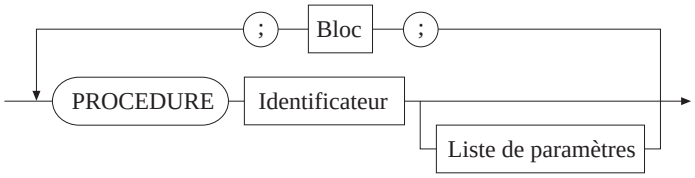

FIGURE 7.1
DIAGRAMME SYNTAXIQUE D'UNE DÉCLARATION DE PROCÉDURE.

L'*activation d'une procédure* est l'opération par laquelle on appelle la procédure en vue de l'exécution des instructions qui constituent son corps; une procédure peut être activée ou appelée plusieurs fois dans un même programme.

Par ailleurs, tout comme le programme, la procédure contient ses propres variables, qui sont dites *locales* par rapport à celles du programme, qualifiées de *globales*. Toute variable locale doit être déclarée dans le corps de la procédure et n'est connue de l'ordinateur qu'au moment de l'exécution de cette procédure : l'espace qui lui est alloué en mémoire ne l'est que pendant l'exécution de la procédure. Ainsi, la portée d'une variable locale se circonscrit à la seule procédure où elle a été définie, alors

qu'une variable globale influe sur l'ensemble du programme, y compris les procédures et les fonctions éventuelles de ce programme. Ces remarques sont aussi valables pour tous les objets (étiquette, constante, type, etc.) qui peuvent être déclarés dans un programme, dans une procédure ou dans une fonction.

Faisons un bref retour sur les procédures non standards du langage PASCAL et considérons, par exemple, l'instruction :

WRITELN(fichier, var1, var2, ..., var*n*);

Les identificateurs qui apparaissent entre les parenthèses constituent les paramètres ou arguments de la procédure standard WRITELN. Le premier paramètre, *fichier*, qui spécifie le fichier de sortie, n'est pas toujours présent. En effet, si la sortie s'effectue à l'écran considéré comme un fichier standard (OUTPUT), ce paramètre devient facultatif. Quant aux autres paramètres, ils servent à spécifier les éléments à afficher ou à imprimer, et ne sont donc pas toujours présents. Dans cet ordre d'idée, l'instruction :

WRITELN;

est valide et permet un saut de ligne à l'affichage : elle n'a aucun paramètre. Cette caractéristique s'étend également aux procédures non standards, qui peuvent être avec ou sans paramètres.

7.1.2 Les procédures sans paramètres

Nous avons vu précédemment que le traitement d'une procédure s'effectue en deux étapes : la déclaration et l'appel. Conformément au diagramme syntaxique d'une déclaration de procédure illustrée à la figure 7.1, l'en-tête d'une procédure sans paramètres a pour format :

PROCEDURE NomProcedure;

Considérons l'exemple 7.1 d'une procédure qui permet de réaliser un saut de trois lignes à l'affichage ou à l'écriture. Cette procédure est sans paramètres et son nom est *Saut3*. Le corps de la procédure comprend deux parties : une partie déclarative où l'on déclare la variable locale *i* de type entier, et un bloc d'instructions exécutables situé entre le BEGIN et le END. On notera que la fin de la procédure est indiquée par un point-virgule, plutôt que par un point comme c'est le cas pour un programme.

Exemple 7.1

```
PROCEDURE Saut3; {En-tete de la procedure Saut3}
{Declaration des variables locales a Saut3}
VAR
      i  : INTEGER;

BEGIN {Debut de la procedure}
      FOR i := 1 TO 3 DO
            WRITELN;
END; {Fin de la procedure}
```

Considérons le programme de l'exemple 7.2 qui appelle la procédure *Saut3*; les variables globales sont *reel1*, *reel2*, *entier1* et *entier2*. La déclaration de la procédure suit immédiatement celle des variables globales. La procédure *Saut3* est appelée deux fois dans le programme principal.

Exemple 7.2

```
PROGRAM Prog71;
USES FORMS;
{Declaration des constantes globales}
CONST   dix = 10;
        cinq = 5;
        deux = 2;
{Declaration des variables globales}
VAR
      reel1, reel2 : REAL;
      entier1, entier2 : LONGINT;

PROCEDURE Saut3; {En-tete de la procedure Saut3}
{Declaration de variables locales}
VAR
      i  : INTEGER;
BEGIN {Debut de la procedure Saut3}
      FOR i := 1 TO 3 DO
            WRITELN;
END; {Fin de la procedure Saut3}
```

```
BEGIN {Debut du programme principal}
      READLN(entier1, entier2);
      READLN(reel2, reel1);
      WRITELN('LA VALEUR DU PREMIER ENTIER EST ', entier1 : cinq);
      WRITELN('LA VALEUR DU DEUXIEME ENTIER EST ', entier2 : 5);
      WRITELN('LA VALEUR DU PREMIER REEL EST ', reel1 : 2*cinq : 4);
      WRITELN('LA VALEUR DU DEUXIEME REEL EST ', reel2 : dix : 4);
      Saut3; {Premier appel de la procedure Saut3}
      WRITE('LA SOMME DES DEUX REELS EST ');
      WRITELN(reel1 + reel2  : dix  : deux);
      WRITE('LA DIFFERENCE DES DEUX REELS EST ');
      WRITELN(reel1 – reel2  : dix  : deux);
      WRITE('LE PRODUIT DU PREMIER ENTIER PAR 10 EST ');
      WRITELN(entier1 * dix  : cinq);
      Saut3; {Deuxieme appel de la procedure Saut3}
      WRITE('LE RESULTAT DE LA DIVISION DU REEL 1 PAR ');
      WRITELN('LE 2 EST ', reel1/reel2  : cinq  : deux);
END. {Fin du programme principal}
```

Si nous entrons les données suivantes dans le programme de l'exemple 7.2 :

```
248     360
5436.85   54368.5
```

la sortie du programme sera conforme à ce qui suit. On notera que chaque appel de la procédure *Saut3* fait sauter trois lignes à l'affichage.

```
LA VALEUR DU PREMIER ENTIER EST   248
LA VALEUR DU DEUXIEME ENTIER EST   360
LA VALEUR DU PREMIER REEL EST 54368.5000
LA VALEUR DU DEUXIEME REEL EST  5436.8500



LA SOMME DES DEUX REELS EST   59805.35
LA DIFFERENCE DES DEUX REELS EST   48931.65
LE PRODUIT DU PREMIER ENTIER PAR 10 EST  2480



LE RESULTAT DE LA DIVISION DU REEL 1 PAR LE 2 EST 10.00
```

7.1.3 Les procédures avec paramètres

On distingue en PASCAL deux types de paramètres : les paramètres formels et les paramètres effectifs. Les *paramètres formels* ou *muets* ne sont que des identificateurs dont les types doivent être spécifiés dans l'en-tête de la procédure. Les *paramètres effectifs* sont tout simplement les valeurs prises par les paramètres formels lors de l'appel de la procédure, de sorte qu'ils doivent correspondre en ordre, en nombre et en type à ces derniers.

Conformément au diagramme syntaxique d'une déclaration de procédure, illustrée à la figure 7.1, l'en-tête d'une procédure avec paramètres a pour format :

PROCEDURE NomProcedure(ListeDeParametres);

où *ListeDeParametres* désigne la liste (non vide) déclarée des paramètres formels de la procédure; comme l'indique la figure 7.2, ces paramètres peuvent être aussi bien des procédures que des fonctions, standards ou non.

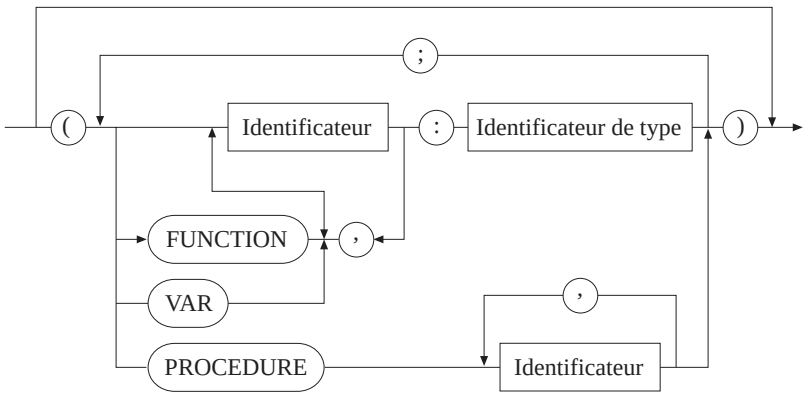

FIGURE 7.2
DIAGRAMME SYNTAXIQUE DE LA LISTE DES PARAMÈTRES D'UNE PROCÉDURE OU D'UNE FONCTION.

Considérons le programme de l'exemple 7.1 dans lequel la procédure permettait de réaliser un saut de trois lignes à l'affichage; cette procédure *Saut3* était sans paramètres On peut la rendre plus générale en la transformant en une procédure avec paramètres. En effet, plutôt que de sauter trois lignes à l'affichage, on peut vouloir sauter un nombre

variable de lignes. C'est ce que réalise la procédure paramétrée *SautLigne* de l'exemple 7.3, procédure qui a pour seul paramètre *nblignes*.

Exemple 7.3

```
PROCEDURE SautLigne(nblignes : INTEGER); {En-tete de la procedure}
{Declaration des variables locales a SautLigne}
VAR
      i : INTEGER;

BEGIN {Debut de la procedure}
      FOR i := 1 TO nblignes DO
            WRITELN;
END; {Fin de la procedure}
```

Considérons le programme de l'exemple 7.4 qui appelle la procédure *SautLigne*. On notera que chaque appel de la procédure *SautLigne* fait sauter le nombre de lignes indiqué par le paramètre effectif, soit deux lignes la première fois et quatre la deuxième fois. On notera également la correspondance de nombre et de type entre paramètres formel et effectif au moment des appels de la procédure.

Exemple 7.4

```
PROGRAM Prog72;
USES FORMS;
{Declaration des constantes globales}
CONST   dix = 10;
        cinq = 5;
        deux = 2;
VAR {Declaration des variables globales}
      reel1, reel2  : REAL;
      entier1, entier2  : LONGINT;

PROCEDURE SautLigne(nblignes : INTEGER); {En-tete de la procedure}
VAR {Declaration des variables locales}
      i : INTEGER;
BEGIN {Debut de la procedure}
      FOR i := 1 TO nblignes DO
            WRITELN;
END; {Fin de la procedure}
```

```
BEGIN {Debut du programme principal}
      READLN(entier1, entier2);
      READLN(reel2, reel1);
      WRITELN('LA VALEUR DU PREMIER ENTIER EST ', entier1 : cinq);
      WRITELN('LA VALEUR DU DEUXIEME ENTIER EST ', entier2 : 5);
      WRITELN('LA VALEUR DU PREMIER REEL EST ', reel1:2*cinq : 4);
      WRITELN('LA VALEUR DU DEUXIEME REEL EST ', reel2 : dix : 4);
      SautLigne(2); {Premier appel : saut de deux lignes}
      WRITE('LA SOMME DES DEUX REELS EST ');
      WRITELN(reel1 + reel2 : dix : deux);
      WRITE('LA DIFFERENCE DES DEUX REELS EST ');
      WRITELN(reel1 – reel2 : dix : deux);
      WRITE('LE PRODUIT DU PREMIER ENTIER PAR 10 EST ');
      WRITELN(entier1 * dix : cinq);
      SautLigne(4); {Deuxieme appel : saut de quatre lignes}
      WRITE('LE RESULTAT DE LA DIVISION DU REEL 1 PAR ');
      WRITELN('LE 2 EST ', reel1/reel2 : cinq  : deux);
END. {Fin du programme principal}
```

Si nous entrons les données suivantes :

```
248    360
5436.85    54368.5
```

la sortie sera conforme à ce qui suit :

```
LA VALEUR DU PREMIER ENTIER EST   248
LA VALEUR DU DEUXIEME ENTIER EST   360
LA VALEUR DU PREMIER REEL EST 54368.5000
LA VALEUR DU DEUXIEME REEL EST  5436.8500


LA SOMME DES DEUX REELS EST   59805.35
LA DIFFERENCE DES DEUX REELS EST   48931.65
LE PRODUIT DU PREMIER ENTIER PAR 10 EST  2480



LE RESULTAT DE LA DIVISION DU REEL 1 PAR LE 2 EST 10.00
```

7.1.4 La transmission des paramètres

On désigne par *transmission de paramètres* le mécanisme par lequel on attribue aux paramètres formels d'une procédure des valeurs, ou paramètres effectifs, au moment de l'appel de cette procédure. En PASCAL, les paramètres sont transmis soit *par valeur* (ou par nom), soit *par variable*.

Dans le premier cas, les paramètres effectifs sont des valeurs, ou des expressions qui peuvent être évaluées au moment de l'appel; et les paramètres par valeur correspondent à des variables locales initialisées avec les valeurs d'appel du paramètre effectif. Ce type de transmission ne permet pas de transmettre des résultats à un autre module (procédure ou fonction) ou au programme principal par l'intermédiaire du paramètre effectif, étant donné que le contenu de celui-ci n'est pas modifié au retour de la procédure. La déclaration d'un paramètre par valeur s'effectue selon le format suivant :

```
PROCEDURE NomProcedure(parametre : TypeParametre);
```

Notons que, dans la procédure *SautLigne* du programme de l'exemple 7.3, le paramètre *nblignes* est déclaré par valeur.

Dans le deuxième cas, les paramètres sont des identificateurs de variable dont le contenu, ou la valeur, est susceptible d'être modifié par la procédure. C'est ce type de transmission qui doit être adopté si on veut utiliser les paramètres pour transmettre des résultats à un autre module ou au programme principal. La déclaration d'un paramètre par variable s'effectue selon le format suivant :

```
PROCEDURE NomProcedure(VAR parametre : TypeParametre);
```

On notera le recours au mot clé VAR pour introduire le ou les paramètres que l'on veut transmettre par variable.

Pour illustrer la différence entre les deux types de transmission, nous allons analyser deux procédures qui permettent d'échanger deux valeurs X et Y. En effet, il est courant en programmation de vouloir permuter la valeur de deux variables; nous l'avons vu en particulier dans les programmes de tri du chapitre précédent. Une telle opération ne peut pas se faire directement, étant donné que l'affectation d'une valeur à une variable rend inaccessible la valeur antérieure de cette variable. La solution consiste alors à

utiliser une troisième variable dite *tampon* qui permet de sauvegarder temporairement la valeur d'une des deux variables X et Y. Ainsi, pour X = 20 et Y = 35, l'échange des valeurs de X et de Y pourrait se réaliser par la séquence suivante :

```
tampon := X; { sauvegarde de la valeur 20 }
X := Y; { X prend la valeur 35 }
Y := tampon;{ Y prend la valeur 20 }
```

Ceci correspond à la procédure *Permuter* du programme de l'exemple 7.5, dans laquelle les paramètres X et Y sont transmis par valeur. Si les valeurs lues pour *val1* et *val2* sont respectivement 200 et 400, l'appel suivi de l'exécution de la procédure *Permuter*, soit l'instruction *Permuter(val1, val2)* du programme principal, afficherait le résultat suivant :

```
X = 400  Y = 200
```

Exemple 7.5

```
PROGRAM Prog73;
USES FORMS;
{Declaration des variables globales}
VAR
      val1, val2  : REAL;

PROCEDURE Permuter(X, Y : REAL); {En-tete de la procedure}
{Declaration des variables locales}
VAR
      tampon : REAL;
BEGIN {Debut de la procedure}
      tampon := X;
      X := Y;
      Y := tampon;
      WRITELN('X = ', X : 4, 'Y = ', Y : 4);
END; {Fin de la procedure}

BEGIN {Debut du programme principal}
      READLN(val1, val2);
      Permuter(val1, val2);
      WRITELN('X = ', val1 : 4, 'Y = ', val2 : 4);
END. {Fin du programme principal}
```

On notera que l'affichage a été produit par l'instruction d'écriture qui se trouve dans le corps de la procédure, et que les valeurs de X et de Y ont été effectivement échangées. En effet, lors de l'appel de la procédure *Permuter(val1, val2)* dans le programme principal, les paramètres formels X et Y prennent respectivement les valeurs des paramètres effectifs *val1* et *val2*, en l'occurrence X = 200 et Y = 400. Ainsi, l'exécution de la procédure a donné lieu aux affectations suivantes :

```
tampon := 200;
X := 400;
Y := 200;
```

Ces nouvelles valeurs de X et de Y, paramètres définis par valeur, ne sont pas transmises à l'extérieur de la procédure. Et c'est la raison pour laquelle l'exécution de l'instruction d'écriture se trouvant dans le programme principal affiche le résultat suivant :

```
X = 200  Y = 400
```

comme si la procédure *Permuter(val1, val2)* n'avait jamais été exécutée. Pour remédier à la situation, les paramètres X et Y doivent être transmis par variable, ce qui aura pour effet de rendre disponibles, à la sortie de la procédure, les nouvelles valeurs effectives de ces paramètres. C'est ce que réalise le programme de l'exemple 7.6 dont le résultat affiché est :

```
X = 400  Y = 200
```

Exemple 7.6

```
PROGRAM Prog74;
USES FORMS;
{Declaration des variables globales}
VAR
      val1, val2 : REAL;

PROCEDURE Permuter(VAR X, Y : REAL); {En-tete de la procedure}
{Declaration des variables locales}
VAR
      tampon : REAL;
BEGIN {Debut de la procedure}
      tampon := X;
      X := Y;
      Y := tampon;
END; {Fin de la procedure}
```

```
BEGIN {Debut du programme principal}
      READLN(val1, val2);
      Permuter(val1, val2);
      WRITELN('VAL1 = ', val1 : 4, 'VAL2 = ', val2 : 4);
END. {Fin du programme principal}
```

Mentionnons enfin qu'une procédure peut elle-même appeler d'autres procédures : c'est ce qu'on appelle l'*imbrication de procédures*.

En guise d'application, nous avons réécrit les programmes de tri du chapitre précédent en introduisant des procédures. Le programme de l'exemple 7.7 correspond au tri par sélection et le programme de l'exemple 7.8 au tri par insertion.

Exemple 7.7

```
PROGRAM Prtrisel;
USES FORMS;
CONST
      nombremax = 25;
TYPE
      tableau = ARRAY[1..nombremax] OF INTEGER;
VAR
      lesnombres : tableau;
      nombre : INTEGER;

PROCEDURE LireDonnees(VAR tab  : tableau);
(****************************************)
(* Procedure de lecture des nombres a trier *)
VAR
      indice : INTEGER;
BEGIN
      WRITELN('QUEL EST LE NOMBRE DE DONNEES A TRIER?');
      READLN(nombre);
            IF (nombre > nombremax) THEN
                  nombre := nombremax;
            WRITELN('ENTREZ LES NOMBRES');
            FOR indice := 1 TO nombre DO
                  READLN(tab[indice])
END;
```

```
PROCEDURE EcrireDonnees(tab : tableau);
(************************************)
(* Procedure d'ecriture des nombres apres le tri *)
VAR
        indice : INTEGER;
BEGIN
        FOR indice := 1 TO nombre DO
        WRITELN(tab[indice]);
END;

PROCEDURE Triselection(VAR tab : tableau);
(****************************************)
VAR
        minj, j, minx, i  : INTEGER;
BEGIN
        FOR i := 1 TO (nombre –1) DO
        BEGIN
                minj := i;
                minx := tab[i];
                FOR j := (i + 1) TO nombre DO
                BEGIN
                        IF (tab[j] < minx) THEN
                        BEGIN
                                minj := j;
                                minx := tab[j];
                        END;
                END;
                tab[minj] := tab[i];
                tab[i] := minx;
        END;
END;

(* Programme principal *)

BEGIN
        LireDonnees(lesnombres);
        Triselection(lesnombres);
        EcrireDonnees(lesnombres);
END. (* Fin du programme *)
```

Exemple 7.8

```
PROGRAM Prtrinsr;
USES FORMS;
CONST
      nombremax = 25; (* Nombre maximum de donnees *)
TYPE
      tableau = ARRAY[1..nombremax] OF INTEGER;
VAR
      lesnombres : tableau;
      nombre : INTEGER;

PROCEDURE LireDonnees(VAR tab : tableau);
(*************************************)
(* Procedure de lecture des nombres a trier *)
VAR
      indice : INTEGER;
BEGIN
      WRITELN('QUEL EST LE NOMBRE DE DONNEES A TRIER?');
      READLN(nombre);
            IF (nombre > nombremax) THEN
                  nombre := nombremax;
            WRITELN('ENTREZ LES NOMBRES');
            FOR indice := 1 TO nombre DO
                  READLN(tab[indice])
END;

PROCEDURE EcrireDonnees(tab : tableau);
(***********************************)
(* Procedure d'ecriture des nombres apres le tri *)
VAR
      indice : INTEGER;
BEGIN
      FOR indice := 1 TO nombre DO
      WRITELN(tab[indice]);
END;

PROCEDURE Trinsertion(VAR tab  : tableau);
(***************************************)
(* Tri des nombres *)
VAR
      i, j, tampon : INTEGER;
```

```
BEGIN
 j : = 0;
      FOR i := 2 TO nombre DO
      BEGIN
            tampon := tab[i];
            j := i – 1;
            WHILE ((j > 0) AND (tab[j] > = tampon)) DO
            BEGIN
                  tab[j + 1] := tab[j];
                  j := j – 1;
            END;
            tab[j + 1] := tampon;
      END;
END; {Trinsertion}

(* Programme principal *)
BEGIN
      LireDonnees(lesnombres);
      Trinsertion(lesnombres);
      EcrireDonnees(lesnombres);
END. (* Fin du programme *)
```

7.2 LES FONCTIONS

En PASCAL, on désigne par *fonction* une séquence d'instructions de programme à laquelle on associe un nom spécifique représenté par un identificateur, un type qui spécifie le résultat produit par cette séquence et une liste de paramètres ou d'arguments, conformément au diagramme syntaxique de la figure 7.2. Dans les sections qui suivent, nous présentons quelques généralités sur cette notion, puis élaborons sur les fonctions avec paramètres.

7.2.1 Généralités

Tout comme les procédures, on distingue deux catégories de fonctions : les fonctions standards et les fonctions non standards. Par *fonction standard*, on entend toute fonction connue du langage, donc qui se passe d'une déclaration préalable lors de son utilisation dans un programme. C'est le cas, entre autres, des fonctions ABS, TRUNC, ROUND, SQR, SQRT, PRED, SUCC que nous avons déjà vues au chapitre 4.

Par *fonction non standard*, on entend toute fonction non connue du langage et qui, par conséquent, nécessite une définition préalable à son utilisation. C'est précisément cette catégorie de fonctions qui retient notre attention dans ce chapitre. Tout comme les procédures, les fonctions permettent d'éviter la duplication d'une même partie de programme. De plus, une même fonction peut être utilisée dans plusieurs programmes différents, ce qui permet une économie de temps de saisie et d'espace en mémoire.

On distingue également deux étapes fondamentales dans le traitement d'une fonction : sa déclaration et son activation. La *déclaration d'une fonction* est l'opération par laquelle on définit, dans la section appropriée du programme, un en-tête et un corps qui spécifient cette fonction. La figure 7.3 présente le diagramme syntaxique d'une déclaration de fonction.

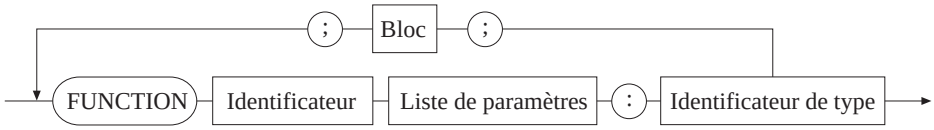

FIGURE 7.3
DIAGRAMME SYNTAXIQUE D'UNE DÉCLARATION DE FONCTION.

L'*activation d'une fonction* est l'opération par laquelle on appelle la fonction en vue de l'exécution des instructions qui constituent son corps. Tout comme une procédure, une fonction peut être activée ou appelée plusieurs fois dans un même programme et elle intègre ses propres variables (locales).

Même s'il est concevable qu'une fonction n'ait pas de paramètres, comme c'est le cas de la fonction constante en mathématique, la notion de fonction sans paramètres n'est pas d'un grand intérêt. En effet, en PASCAL, la liste des arguments d'une fonction est rarement vide : elle contiendra au moins un élément.

7.2.2 Les fonctions paramétrées

Conformément au diagramme syntaxique d'une déclaration de fonction, illustrée à la figure 7.3, l'en-tête d'une fonction avec paramètres a pour format :

FUNCTION NomFonction(ListeDeParametres) : TypeFonction;

où *ListeDeParametres* désigne la liste (non vide) déclarée des paramètres formels de la fonction, telle qu'elle est définie à la figure 7.2, et *TypeFonction* le type de la fonction, c'est-à-dire celui du résultat calculé ou généré par cette fonction. L'exemple 7.9 définit une fonction *Somme* qui calcule la somme de deux nombres réels : *nombre1* et *nombre2*.

Exemple 7.9

```
FUNCTION Somme(nombre1, nombre2 : REAL); Real; {En-tete de la fonction}
{Declaration des variables locales a Somme}
VAR
      temp : REAL;

BEGIN {Debut de la fonction}
      Somme := nombre1 + nombre2;
END;{Fin de la fonction}
```

En PASCAL, il existe une fonction standard, SQR(*x*), permettant d'évaluer le carré d'un nombre *x*. Cependant, il n'existe pas une fonction standard permettant de calculer la puissance *n*ième d'un nombre *x*, avec $n > 2$. À cette fin, la fonction *Puissance* du programme de l'exemple 7.10 peut être utilisée; elle est basée sur l'égalité suivante :

$$x^n = \underbrace{x.x. \ldots .x}_{n \text{ fois}}$$

Exemple 7.10

```
PROGRAM Prog77;
USES FORMS;
{Declaration des variables globales}
VAR
      val1, val2, val3, val4, val5 : INTEGER;

FUNCTION Puissance(x : INTEGER, n : INTEGER); Integer; {En-tete de la fonction}
VAR
      temp : INTEGER;
      i  : INTEGER;
```

```
BEGIN
      temp := 1;
      FOR i := 1 TO n  DO
            temp := temp * x;
      Puissance := temp;
END;

BEGIN {Debut du programme principal}
      READLN(val1, val2, val3);
      val4 := Puissance(val1, 2) + Puissance(val2, 2);
      val5 := Puissance(val3, 2);
      WRITELN('val4 = ', val4 : 4, 'val5 = ', val5 : 4);
END. {Fin du programme principal}
```

En se référant au programme de l'exemple 7.10, on peut constater que l'appel d'une fonction s'effectue différemment de l'appel d'une procédure. En effet, alors qu'une procédure est appelée par son nom et utilisée comme une instruction exécutable, une fonction est plutôt traitée comme une expression à évaluer lors de son appel. Par ailleurs, si on attribue à *val1* la valeur 3, à *val2* la valeur 4 et à *val3* la valeur 5, ce programme indiquera que *val4* est égal à *val5*, ce qui débouche sur la conjecture de Fermat dont on a parlé au premier chapitre.

Mentionnons enfin qu'il est possible d'utiliser des fonctions comme paramètres de procédures ou d'autres fonctions.

7.3 LES PROCÉDURES ET LES FONCTIONS RÉCURSIVES

On dit d'une procédure ou d'une fonction qu'elle est *récursive* lorsque cette procédure ou cette fonction s'appelle elle-même par une relation dite de récurrence. Concrètement, cela correspond à la situation où le nom de la procédure ou de la fonction est utilisé dans le bloc des instructions exécutables qui composent le corps de celle-ci. À chaque appel récursif d'une procédure ou d'une fonction, de nouvelles variables locales sont créées et les résultats correspondants sont empilés, de sorte que le résultat du dernier appel demeure le premier qui soit accessible. Les structures itératives vues au chapitre précédent constitue souvent une solution de remplacement pour la récursivité.

7.3.1 La somme de nombres

La somme des *n* premiers nombres entiers (positifs ou nuls) peut être calculée par une fonction récursive, en observant que cette somme est égale à ce nombre *n* additionné de la somme de tous ceux qui le précèdent. Si on désigne par *Somme(*n*)* cette fonction, cela se traduit par la relation de récurrence suivante :

$$\text{Somme}(n) = n + \text{Somme}(n - 1)$$

En effet, la somme des 10 premiers entiers, par exemple, est égale à 10, additionné de la somme des neuf entiers qui précèdent le nombre 10. Pour que cette définition soit consistante, il faut poser que :

$$\text{Somme}(0) = 0$$

Car, la définition générale de la fonction ne permet pas de calculer la somme de zéro nombre. L'exemple 7.11 définit la fonction *Somme*.

Exemple 7.11

```
FUNCTION Somme(Nombre : INTEGER) : INTEGER;
BEGIN
      IF Nombre = 0 THEN Somme := 0
      ELSE Somme := Nombre + Somme(Nombre – 1);
END;
```

7.3.2 Le produit de nombres

Tout comme la somme, le produit des *n* premiers nombres entiers (positifs ou nuls) peut être calculé par une fonction récursive, en observant que ce produit est égal à celui du nombre *n* par le produit de tous ceux qui précèdent ce nombre. En mathématique, une telle opération ou fonction s'appelle la *factorielle de n* et se note *n!*. Si on désigne par *Fact(*n*)* cette fonction, cela se traduit par la relation de récurrence suivante :

$$\text{Fact}(n) = n * \text{Fact}(n - 1)$$

En effet, le produit des 10 premiers entiers, par exemple, est égal à 10 fois le produit des neuf entiers qui précèdent le nombre 10. Tout comme la somme, pour que cette définition soit consistante, il faut poser que :

$$Fact(0) = 1$$

Car, la définition générale de la fonction ne permet pas de calculer la factorielle de zéro. L'exemple 7.12 définit la fonction *Fact* appelée par le programme de l'exemple 7.13.

Exemple 7.12

```
FUNCTION Fact(Nombre : INTEGER) : INTEGER;
BEGIN
      IF Nombre = 0 THEN Fact := 1
      ELSE Fact := Nombre * Fact(Nombre – 1);
END;
```

Exemple 7.13

```
PROGRAM Prog78;
USES FORMS;
VAR
      val1, val2, val3, val4 : INTEGER;

FUNCTION Somme(Nombre : INTEGER) : INTEGER;
BEGIN
      IF Nombre = 0 THEN Somme := 0
      ELSE Somme := Nombre + Somme(Nombre – 1);
END;

FUNCTION Fact(Nombre : INTEGER) : INTEGER;
BEGIN
      IF Nombre = 0 THEN Fact := 1
      ELSE Fact := Nombre * Fact(Nombre – 1);
END;

BEGIN {Debut du programme principal}
      READLN(val1, val2);
      val3 := Somme(val1);
      val4 := Fact(val2);
      WRITELN('val3 = ', val3 : 4, 'val4 = ', val4 : 4);
END. {Fin du programme principal}
```