Chapitre 3 PROCEDURES ET FONCTIONS

A la fin de cette partie, vous serez capable de:

- structurer un programme en utilisant des fonctions et procédures;
- utiliser le passage de paramètres afin d'écrire des programmes ou modules plus indépendants et réutilisables;
- comprendre la notion de récursivité

3.1 Introduction

Comme nous l'avons vu précédemment, un programme doit être clair et bien structuré afin que son utilisation et sa maintenance posent un minimum de problèmes. Le besoin de structuration ne semble pas toujours essentiel pour les petits programmes. Mais au fur et à mesure que la complexité ou la taille d'un programme augmentent, une bonne structure n'est plus seulement un besoin, mais une nécessité. Il convient donc d'acquérir de bonnes habitudes de programmation dès le début, cet effort étant largement récompensé par la suite.

Le langage Pascal a été conçu pour que les programmes soient décomposés en modules (procédures, fonctions ou unités) de taille raisonnable. Toutefois il faut éviter un découpage artificiel du programme, au profit d'une décomposition logique. Chaque procédure ou fonction correspond à une tâche élémentaire bien déterminée. La décomposition en modules permet de sérier les différentes phases conduisant à la résolution d'un problème. Cette méthode offre entre autres avantages celui d'une maintenance grandement facilitée, puisque la localisation d'un point critique dans un programme est plus précise et rapide.

Les notions de procédure ou fonction s'apparentent à la notion plus classique de "sous-programme", terme encore employé dans certains langages de programmation. En Pascal, les procédures et les fonctions ont beaucoup de caractéristiques communes, et relativement peu de différences. Dans les sections qui suivent nous nous attacherons surtout à l'étude des procédures. Nous reviendrons dans la section 3.8 sur les caractéristiques qui différencient les fonctions des procédures.

La mise en oeuvre de modules contribue également et largement à une bonne structuration et modularisation des programmes. Nous reviendrons sur cette notion plus loin dans le cours.

Une procédure (ou une fonction) possède une structure analogue à celle d'un programme. En tant qu'entité, une procédure apparaît dans un programme lors de deux phases distinctes. La première constitue la **déclaration**, et la seconde l'**invocation** (ou appel).

3.2 Déclaration

La déclaration d'une procédure est constituée par une description complète comprenant: un en-tête, une partie réservée aux déclarations locales et le corps proprement dit de la procédure. L'en-tête comprend le mot réservé **procedure**, suivi du nom de la procédure et, facultativement, d'une liste de paramètres. Les déclarations sont indiquées exactement de la même manière que pour le programme. Le corps de la procédure est un bloc d'instructions, au même titre que le corps du programme:

```
procedure pied;
const ...
type  ...
var   ...
begin
  ...
end;
```

Entête

Déclarations

Corps de la procédure

Comme nous l'avons vu, en Delphi, l'essentiel des instructions d'un programme se trouve dans les unités. Il faut ici distinguer deux situations dans lesquelles on est amené à écrire une procédure.

La première concerne l'écriture d'une procédure indépendante de la fiche courante. Dans ce cas, la procédure n'a pas accès à la fiche et aux objets qu'elle contient. Il s'agit souvent de procédures indépendantes, effectuant des opérations sans lien direct avec les objets de l'interface du programme. Voici un exemple dans lequel le clic sur un bouton provoque l'émission d'un son:

Le code se présente de la manière suivante:

```
unit Unit1;

interface

uses
  Windows, Messages, SysUtils, Classes, Graphics, Controls, Forms,
Dialogs, StdCtrls;

type
  TForm1 = class(TForm)
    Button1: TButton;
    procedure Button1Click(Sender: TObject);
  private
    { Déclarations privées }
  public
    { Déclarations publiques }
  end;

var
  Form1: TForm1;

implementation

{$R *.DFM}

procedure Bip;
begin
  Messagebeep (0);
end;

procedure TForm1.Button1Click(Sender: TObject);
begin
  Bip;
end;
```

Code de la procédure

Appel de la procédure

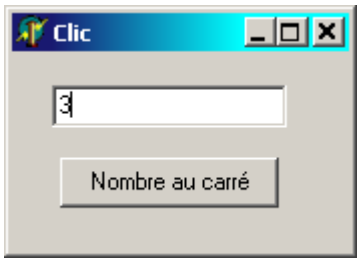
```
end.
```

La procédure Bip ne fait appel à aucun objet de l'interface utilisateur; Messagebeep (0) étant une procédure standard de Windows. Le compilateur n'aurait cependant pas accepté l'instruction

```
Form1.Caption := 'exemple';
```

dans le corps de la procédure Bip.

La seconde situation dans laquelle on peut utiliser une procédure est celle où l'on permet à la procédure d'accéder aux objets de la fiche. Prenons un exemple dans lequel un clic sur un bouton provoque la mise au carré d'un nombre entier contenu dans un Edit:



Cet exemple bien qu'artificiel illustre néanmoins la situation:

```
unit Unit1;
interface
uses
  Windows, Messages, SysUtils, Classes, Graphics, Controls, Forms,
Dialogs, StdCtrls;

type
  TForm1 = class(TForm)
    Edit1: TEdit;
    Button1: TButton;
    procedure Button1Click(Sender: TObject);
  private
    { Déclarations privées }
    procedure AfficherCarre;
  public
    { Déclarations publiques }
  end;

var Form1: TForm1;

implementation
{$R *.DFM}

procedure TForm1.AfficherCarre;
begin
  Edit1.text := inttostr (sqr(strtoint(Edit1.text)));
end;

procedure TForm1.Button1Click(Sender: TObject);
begin
  AfficherCarre;
end;
end.
```

Déclaration anticipée

Procédure proprement dite

Appel de la procédure

On peut faire plusieurs constatations:

- il faut une déclaration anticipée (également appelée prototype) de la procédure. Cette déclaration peut intervenir dans la partie **private** ou dans la partie **public**. Dans le premier cas, la procédure AfficherCarre peut être appelée uniquement depuis l'unité unit1. Dans le second cas, la procédure AfficherCarre pourrait également être appelée depuis une autre unité faisant référence à unit1.
- Dans l'écriture proprement dite de la procédure, le nom de la procédure est préfixé par le nom de la classe à qui elle appartient: Tform1.
- La procédure AfficherCarre peut accéder à la fiche Form1 et à tous les objets qu'elle contient

Si plusieurs procédures doivent être définies, leurs déclarations se suivent:

```
...
      procedure premiere;
      begin
        (* corps de la procédure "premiere" *)
      end;

      procedure deuxieme;
      begin
        (* corps de la procédure "deuxieme" *)
      end;
```

fig. 3.1

3.3 Invocation (ou appel)

Comme nous venons de le voir, une fois définie, une procédure peut être invoquée en indiquant son nom. Le nom tient lieu, en fait, d'instruction. En règle générale, une procédure ne peut être invoquée qu'après avoir été déclarée.

Lors de l'invocation d'une procédure, l'exécution du programme se poursuit par l'exécution de la première instruction du corps de la procédure. Au moment où l'exécution de la procédure est terminée, le contrôle est rendu à la partie appelante du programme, c'est-à-dire à l'instruction qui suit l'appel à la procédure.

3.4 Procédures emboîtées

Par analogie à la déclaration d'une procédure dans un programme ou une unité, il est possible de déclarer une procédure à l'intérieur d'une autre procédure. L'exemple qui suit illustre l'emboîtement des procédures **premiere** et **deuxieme**:

```
      procedure autour;
      var x,z : integer;

        procedure premiere;

          procedure deuxieme;
          begin
            (* corps de la procédure "deuxieme" *)
          end;

        begin
          (* corps de la procédure "premiere" *)
        end;


      begin
        (* corps de la procédure "autour" *)
      end.
```

fig. 3.2

Les exemples des figures 3.1 et 3.2 diffèrent uniquement par l'emplacement où est déclarée la procédure **deuxieme**. Dans le premier cas, les deux procédures sont déclarées au même niveau et chacune d'entre elles peut être appelée depuis l'extérieur. Dans le second cas, seule la procédure **premiere** peut être appelée depuis l'extérieur (par exemple depuis la procédure **autour**); la procédure **deuxieme** étant déclarée à l'intérieur de la procédure **premiere**, elle ne peut pas être appelée en dehors de la procédure **premiere**. Toutefois elle peut être appelée depuis le corps de la procédure **premiere**. La procédure englobante agit comme un écran par rapport à la procédure emboîtée. On dit que la "visibilité" de la procédure emboîtée **deuxieme** est masquée.

En Pascal, le processus d'emboîtement de procédures est une conséquence naturelle de l'application de la méthode de raffinement graduel à la résolution d'un problème. Dans la pratique il est rare de rencontrer des niveaux d'emboîtement supérieurs à quatre ou cinq. Dans la plupart des programmes simples écrits en Delphi, il est même rare de rencontrer des procédures emboîtées, étant donné que la programmation événementielle favorise la structure d'un programme en de nombreuses procédures répondant précisément chacune à un événement.

L'emboîtement des procédures détermine donc des domaines de visibilité au niveau des procédures, mais aussi au niveau des objets déclarés dans les procédures. Le domaine de visibilité d'un objet est lié aux concepts de localité et de globalité. Une bonne compréhension et maîtrise de ces nouvelles notions qui sont l'objet des deux prochaines sections, permettent une meilleure approche de la programmation structurée.

3.5 Niveaux d'emboîtement

Nous avons vu que chaque procédure peut posséder ses propres objets, définis dans la partie réservée aux déclarations. Dans Delphi, la quasi-totalité des instructions d'un programme étant située dans les unités, nous allons prendre l'unité comme structure de référence afin de comprendre les niveaux d'emboîtement des procédures et la notion de domaine de visibilité.

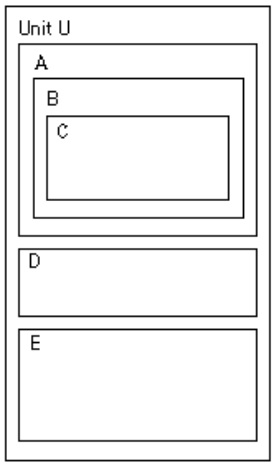


fig. 3.3

L'emboîtement des procédures crée une hiérarchie et définit des niveaux:

- L'unité (objet défini par le mot réservé **unit**) se trouve au niveau 1; on dira que l'unité est un bloc de niveau 1;
- tout objet déclaré dans le bloc de niveau 1 sera global pour les blocs emboîtés;
- toute procédure déclarée dans le bloc de niveau 1 définit un bloc de niveau 2 et ainsi de suite;
- tout objet défini au niveau 2 est local à ce niveau et global pour les blocs emboîtés.

Les concepts de localité et de globalité sont fondamentaux en Pascal. Un découpage tenant compte de la localité et de la globalité des différents objets permet une cohérence de structure optimale, et contribue à rendre le programme plus fiable.

3.6 Règles concernant le domaine de visibilité

Nous appelons *domaine de visibilité* d'un objet, la portion de programme d'où cet objet peut être référencé. Dans un programme Pascal, la visibilité d'un objet est souvent limitée à une procédure. Les règles à connaître pour maîtriser le concept de domaine de visibilité sont simples, mais pas évidentes de prime abord:

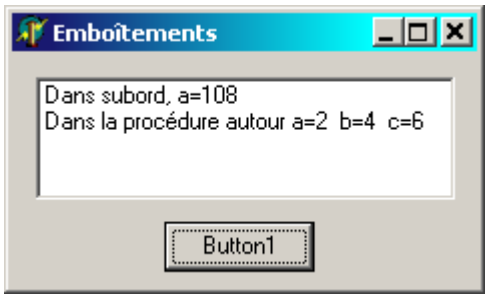
1. *Le domaine de visibilité d'un objet est constitué par le bloc dans lequel il est déclaré (où il est local) et les blocs subordonnés (pour lesquels il est global), sous réserve du point 2.*

2. *Si un objet, défini dans un bloc A, est redéfini dans un bloc emboîté B, alors le bloc B et tous les blocs emboîtés dans B sont exclus du domaine de visibilité de l'objet déclaré dans A.*

Afin de comprendre ces règles, examinons leur application à l'exemple suivant, dans lequel on considère que le niveau global est représenté par la procédure **autour**:

```
...
procedure TForm1.autour;
var  a, b, c : integer;    { variables globales }

  procedure englobe;
  var b, d : integer;     { variables locales à englobe }

    procedure subord;
    var a, e : integer;   { variables locales à subord }
    begin
      a := b * c;
      listbox1.items.add ('Dans subord, a=' + inttostr(a));
    end;

  begin
    b := 3 * c;
    subord;
  end;

begin
  a:=2;   b:=4;   c:=6;
  englobe;
  listbox1.items.add ('Dans la procédure autour a=' + inttostr(a)
                    + ' b=' + inttostr(b) + ' c=' + inttostr(c));
end;

procedure TForm1.Button1Click(Sender: TObject);
begin
  autour;
end;
```

Voici les résultats affichés par ce programme:



L'affectation `b:=3*c` concerne la variable b locale à la procédure **englobe**, le b global étant préservé. Dans la procédure **subord**, l'affectation `a:=b*c` concerne la variable a locale à la procédure **subord**, la variable b de la procédure **englobe** et la variable c globale.

Les variables locales à une procédure ont une durée d'existence correspondant au temps d'exécution de la procédure. Une procédure peut donc jouir d'une certaine indépendance, du moins au niveau de ses objets locaux. Cette possibilité permet de préserver les objets globaux d'éventuelles fausses manoeuvres, et ainsi de faciliter la recherche d'erreurs.

3.7 Transfert d'information avec les procédures

3.7.1 Introduction

Dans les exemples abordés jusqu'ici, l'échange d'information entre une procédure et son environnement s'effectuait par l'intermédiaire des objets déclarés à un niveau global. Par exemple, lors de l'appel à une procédure, celle-ci travaille avec des données contenues dans les variables globales. A la fin de son exécution, le programme principal (ou la procédure appelante) reprend le contrôle et peut utiliser les mêmes variables, dont le contenu aura éventuellement été modifié par la procédure. Ce mode de fonctionnement est commun à d'autres langages de programmation tels le BASIC ou le COBOL. Dans ces langages le domaine de visibilité des objets définis est le programme entier. Tous les objets sont donc globaux.

Cette approche n'est pas forcément la meilleure, mais elle est facile à assimiler. En revanche, elle implique un inconvénient allant à l'encontre des principes d'une programmation claire, sûre et structurée: une protection insuffisante des variables, ou plutôt de leur contenu. En effet, une variable peut être modifiée dans n'importe quelle partie d'un programme, et, de ce fait, la difficulté de la maintenance d'un programme croît exponentiellement en fonction de sa taille.

Lorsque le domaine de visibilité d'un objet est restreint à une procédure, il est plus facile, en cas de mauvais fonctionnement relatif à cet objet, d'intervenir uniquement dans la portion du programme d'où cet objet est visible. La difficulté d'éliminer une erreur est alors moins dépendante de la longueur du programme.

Donc, comme l'utilisation exclusive des variables globales n'est pas satisfaisante, nous allons étudier une autre technique de communication entre procédures: le **passage de paramètres**. Cette technique permet de rendre les procédures plus indépendantes, portables et souples, et est implémentée dans presque tous les langages de programmation modernes (Modula-2, Portal, ADA, etc.).

3.7.2 Passage de paramètres

Une procédure paramétrée se distingue par le fait que son nom est suivi d'une liste de paramètres constituant, en quelque sorte, une boîte aux lettres:

```
procedure nom ( liste de paramètres );
```

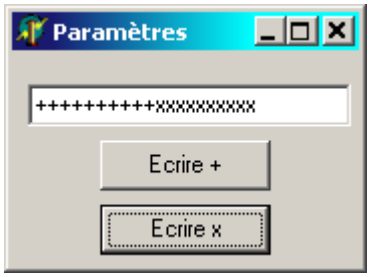
Une liste de paramètres figure à la fois dans la déclaration et dans l'appel d'une procédure. Elle consiste, lors de la déclaration, en une énumération d'identificateurs de variables suivis de leur type, de manière analogue à la déclaration habituelle des variables. Des virgules séparent les variables de même type et des points-virgules séparent les variables de types différents. Lors de la déclaration d'une procédure, on parle de **paramètres formels**, et lors de l'invocation de la procédure on parle de **paramètres effectifs** (ou **réels**). Ces variables particulières sont utilisées pour les échanges d'information entre la procédure et son environnement. Chaque paramètre réel est associé à un paramètre formel.

3.7.3 Nature des paramètres

Les paramètres d'une procédure peuvent appartenir à deux catégories distinctes. La première concerne les paramètres utilisés par la procédure pour recevoir des informations de l'extérieur, fournies lors de son invocation. Nous les appellerons **paramètres d'entrée**. La modification éventuelle du contenu de ces paramètres dans la procédure ne peut être répercutée à l'extérieur de celle-ci. Les paramètres d'entrée constituent donc des variables purement locales, dont la valeur est reçue de l'extérieur. L'exemple qui suit illustre ce premier type de transfert d'information.

```
...
procedure TForm1.Ecrire (c: char);
var i : integer;
begin
  for i := 1 to 10 do
    Edit1.text := Edit1.text + c;
end;

procedure TForm1.Button1Click(Sender: TObject);
var car : char;
begin
  car := '+';
  Ecrire (car);
end;

procedure TForm1.Button2Click(Sender: TObject);
begin
  Ecrire ('x');
end;
...
```

Voici comment se présente le programme après avoir cliqué sur le bouton 1 puis sur le bouton 2:



Lors de la première invocation (clic sur le bouton 1), le caractère "+" contenu dans la variable **car** est transmis à la procédure **Ecrire**. Cette dernière reçoit l'information via le paramètre formel **c**, et l'utilise lors de son exécution pour ajouter le résultat dans Edit1:

```
++++++++++
```

Lors de la seconde invocation, une constante est directement transmise à la procédure qui affiche le texte suivant dans Edit1:

```
xxxxxxxxxx
```

On constate dans cet exemple qu'un paramètre effectif peut être constitué par une variable ou par une constante. En réalité, il est constitué par une expression, au sens complet du terme en Pascal. Ceci comprend, en plus des variables et des constantes, des appels de fonctions, des expressions mathématiques, logiques, etc.

La seconde catégorie concerne les paramètres utilisés par la procédure à la fois pour **recevoir** et pour **transmettre** des informations. Nous les appellerons **paramètres de sortie**. Ce type de paramètres intervient lorsque la procédure fournit des résultats à son environnement. Toute modification, par la procédure, d'un paramètre de sortie est répercutée sur le paramètre réel correspondant. Un paramètre de sortie constitue donc un canal de communication bidirectionnel entre la procédure et l'extérieur. Dans l'exemple qui suit, **a** et **b** sont des paramètres d'entrée, alors que **somme** est un paramètre de sortie.

Déclaration de la procédure:

```
procedure addition (a, b: integer; var somme: integer);
begin
  somme := a + b;
end;
```

fig. 3.4

Appels à la procédure:

```
montant1 := 14;
montant2 := 7;
addition (montant1, montant2, total);
Edit1.text := inttostr(total);
addition (235, 418, resultat);
Edit2.text := inttostr(resultat);
```

fig. 3.5

Lors du premier appel, la procédure **addition** (figure 3.4) reçoit les valeurs de **montant1** et de **montant2** par l'intermédiaire des paramètres **a** et **b**. Elle calcule leur somme et transmet le résultat au programme principal par l'intermédiaire du paramètre effectif **total**. Après exécution, Edit1 contiendra le nombre 21, et Edit2 le nombre 653. On constate qu'un paramètres de sortie est précédé, lors de la déclaration, du mot réservé **var**.

3.7.4 Mécanisme de transfert

Le mécanisme de transfert faisant appel à des paramètres d'entrée est appelé transfert **par valeur**. Dans le cas où des paramètres de sortie interviennent, on parle de transfert **par référence** (ou **par adresse**) .

Lors de l'invocation d'une procédure avec paramètres, ces derniers doivent obéir aux règles de concordance suivantes:

1. *Le nombre de paramètres formels doit être égal au nombre de paramètres effectifs.*

2. *L'ordre des paramètres spécifiés lors de l'appel à la procédure doit être le même que celui spécifié dans la déclaration.*

3. *Le type d'un paramètre effectif doit être le même que celui du paramètre formel correspondant.*

Remarques:

- les paramètres formels sont apparentés à des variables locales à la procédure;
- les paramètres formels transmis par référence sont précédés du mot réservé **var**, alors que ceux transmis par valeur ne le sont pas;
- lors de l'invocation d'une procédure, à la place d'un paramètre réel transmis par valeur, on peut indiquer une constante, une variable, ou une expression;
- lors de l'invocation d'une procédure, un paramètre réel transmis par référence ne peut être qu'une variable.

Pour illustrer ces nouvelles notions, nous allons examiner trois exemples de programmes dont les résultats à l'exécution sont strictement identiques; seul le mode de transfert de l'information change. Le problème à résoudre par les trois programmes consiste à déterminer et afficher le plus grand des deux nombres donnés.

Utilisation des variables globales

```
var Form1: TForm1;
    x, y, max : real;

implementation
{$R *.DFM}

procedure TForm1.maximum;
begin
  if x > y then
    max := x
  else
    max := y;
  edit1.text := floattostr(max);
end;


procedure TForm1.Button1Click(Sender: TObject);
begin
  x := 12;
  y := 23;
  Maximum;
end;
```

Dans ce programme, la procédure utilise uniquement des variables globales, déclarées dans l'entête de l'unité. Ceci présente un gros inconvénient, car la procédure **maximum** peut traiter uniquement les variables **x** et **y**. Elle n'est donc pas assez générale.

Transfert de paramètres par valeur

```
var Form1: TForm1;
    x, y, max : real;

implementation
{$R *.DFM}

procedure TForm1.maximum (a, b: real);
begin
  if a > b then
    max := a
  else
    max := b;
  edit1.text := floattostr(max);
end;

procedure TForm1.Button1Click(Sender: TObject);
```

```
begin
  x := 12;
  y := 23;
  Maximum (x, y);
end;
```

Dans ce programme, les paramètres formels **a** et **b** reçoivent lors de l'appel les valeurs contenues dans les paramètres réels **x** et **y**.

Transfert de paramètres par valeur et par référence

```
var Form1: TForm1;
    x, y, grand : real;

implementation
{$R *.DFM}

procedure TForm1.maximum (a, b: real; var max: real);
begin
  if a > b then
    max := a
  else
    max := b;
end;

procedure TForm1.Button1Click(Sender: TObject);
begin
  x := 12;
  y := 23;
  Maximum (x, y, grand);
  edit1.text := floattostr(grand);
end;
```

Dans ce dernier exemple, la procédure reçoit les deux valeurs à comparer par l'intermédiaire des paramètres d'entrée **a** et **b**. Elle détermine ensuite la plus grande des deux valeurs (**max**) et la transmet au programme appelant. Cette dernière phase fait appel au transfert par référence, **max** étant un paramètre de sortie.

La procédure maximum décrite dans le dernier des trois exemples est la plus proche de l'esprit de la programmation structurée. Dans cette optique, une procédure doit être indépendante et ses échanges avec l'extérieur doivent passer, dans la mesure du possible, par un transfert de paramètres. Lorsque ces conditions sont remplies la procédure est autonome, générale et plus fiable. Il serait, par exemple, très facile de reprendre la procédure maximum du troisième exemple dans un programme totalement différent. Son utilisation implique uniquement la connaissance de ses paramètres et de leur signification.

Voici quelques avantages liés à l'utilisation des procédures:

- mise en pratique aisée des concepts de la programmation structurée et de la conception descendante;
- rentabilisation du travail accompli. Lorsqu'un problème ou une partie de problème a déjà été résolu, par soi-même ou par d'autres, il ne faut pas repartir de zéro. La tendance est plutôt de constituer des bibliothèques de procédures et de fonctions, groupées, par exemple, en unités. Cette méthode évite de récrire des routines semblables dans chaque nouveau programme et est largement répandue dans le domaine de la programmation. Plus un programmeur est expérimenté, plus sa bibliothèque de sous-programmes (procédures, fonctions et unités) grandit, plus ses nouveaux programmes sont réalisés rapidement;
- gain de place. Lorsque la même séquence d'instructions apparaît à plusieurs emplacements d'un programme, il convient d'en faire une procédure. De plus, la paramétrisation d'une procédure permet de traiter des problèmes analogues, mais pas forcément identiques.

- Enfin, lorsque les programmes font appel à des algorithmes récursifs, l'utilisation de procédures ou de fonctions est indispensable.

3.7 Autres modes de passage de paramètres

Les passages de paramètres par valeur ou par référence sont les plus fréquemment utilisés. Delphi propose d'autres possibilités concernant le passage de paramètres à une procédure.

Paramètres "constantes"

Lors d'un passage de paramètre par valeur, bien que qu'aucune information ne sort de la procédure, celle-ci peut utiliser et modifier l'information quelle reçoit. Elle utilise alors le paramètre formel comme une variable locale.

Dans le but d'éviter cette possibilité il est possible de faire appel à des paramètres "constantes" dont voici un exemple d'utilisation:

```
procedure compare (const t1, t2: integer);
```

Dans la procédure **compare** les paramètres **t1** et **t2** ne peuvent pas être modifiés.

Paramètres de sortie

Un paramètre de sortie (précédé de **out**) est transmis par référence comme un paramètre précédé de **var**. Toutefois, avec un paramètre de sortie, la valeur initiale de la variable référencée est ignorée par la procédure à laquelle elle est transmise. Le paramètre **out** n'est utilisé qu'en sortie; il indique simplement à la procédure où placer la valeur de sortie, sans spécifier la valeur d'entrée. Dans l'exemple suivant:

```
procedure TForm1.maximum (a, b: real; out max: real);
begin
  if a > b then
    max := a
  else
    max := b;
end;
```

il est clair que **max** est un paramètre de sortie. En effet, on ne connaît pas sa valeur lors de l'appel de la procédure maximum.

Paramètres sans type

Seuls les paramètres précédés de **var**, **out** et **const** peuvent ne pas avoir de type (alors que les paramètres passés par valeur doivent toujours avoir un type).

Dans le corps d'une procédure les paramètres sans type sont incompatibles avec tous les types. De ce fait, ils doivent être utilisés uniquement en mode transtypage comme dans l'exemple qui suit:

```
function Egal (var Src, Dest; Taille: Integer): Boolean;
type TOctets = array[0..MaxInt - 1] of Byte;
var N: Integer;
begin
  N := 0;
  while (N < Taille) and (TOctets(Dest)[N] = TOctets(Src)[N]) do
    Inc(N);
  Egal := N = Taille;
end;
```

3.8 Fonctions

Les notions abordées dans les sections précédentes, liées aux procédures, sont également valables pour les fonctions. La structure et l'emploi de ces deux types d'objets est très semblable. Voici toutefois quelles en sont les différences essentielles:

- le mot réservé **function** remplace le mot réservé **procedure** lors de la déclaration d'une fonction;
- par analogie avec la notion mathématique, les paramètres (arguments) d'une fonction ne devraient pas être modifiés. Toutefois le transfert de paramètres par référence à une fonction est admis en Pascal;
- une fonction fournit un résultat de type scalaire. Ce type est spécifié, lors de la déclaration de l'en-tête, par le signe ":" suivi de l'identificateur de type; cette indication est placée après le nom de la fonction et la liste de paramètres éventuelle;
- dans le corps d'une fonction, on doit affecter au moins une fois une valeur à l'identificateur de la fonction ou à la variable **result**;
- contrairement à l'appel d'une procédure qui, lui, est considéré comme une **instruction**, l'appel à une fonction est considéré comme opérande d'une **expression**. Cet opérande, ou l'expression qui le contient, peuvent être affectés à une variable ou bien imprimés.

La manière dont une fonction doit être déclarée est illustrée par l'exemple qui suit. La fonction tg fournit la valeur de la tangente d'un angle exprimé en radians:

```
function tg (x: real): real;
begin
  tg := sin (x) / cos (x);
```

```
end;
```

Dans un programme contenant cette déclaration, on pourrait, par exemple, écrire les instructions suivantes:

```
z := tg (alpha);
ctg := 1 / tg (y);
Resultat.text := 'Tangente = ' + floattostr(tg(y));
```

Remarque

L'avantage d'affecter le résultat d'une fonction à la variable **result** plutôt qu'à l'identificateur de la fonction est de pouvoir s'en servir sans risque de récursivité au sein de la fonction, comme dans l'exemple suivant:

```
function tracer (x: real): real;
begin
  result := 3*x-2;
  if result < 0 then
    result := abs (result)
end;
```