Chapitre 1 : Sous-Programmes - Procédures et Fonctions

1.1. Introduction

En général, les problèmes qui se posent en pratique sont suffisamment complexes et nécessitent leurs décomposition en sous problèmes plus faciles à résoudre séparément. De là vient l'idée de décomposition d'un programme complexe en sous-programmes (fonctions et procédures) plus facile à écrire et à contrôler. Ces fonctions et procédures une fois rassemblées, peuvent communiquer des résultats entre elles et avec le programme principal.

Une autre raison d'usage des fonctions et procédures est pour éviter des redondances de codes dans le programme. Considérons l'exemple suivant pour illustrer ce dernier cas.

Soit à calculer le nombre de combinaisons de *k* éléments à partir de *n* élément (*n* et *k* deux nombres naturels) :

$$C_n^k = \frac{n!}{k! \times (n-k)!} \quad si\ k <= n$$

On remarque qu'il y a trois factoriels à calculer. Si on n'utilise pas de sous-programmes (fonctions ou procédure), on risque de répéter dans le programme des séquences semblables d'instructions (redondance) ce qui nuit à la clarté du programme, sa lisibilité et contiendra plusieurs séquences semblables d'instructions (dans l'exemple,le nombre d'instructions est multiplié par 3).

Afin de rendre le programme plus lisible et réduire le nombre d'instruction (moins d'espace mémoire), on fait appel à des fonctions et procédures (sous-programmes).

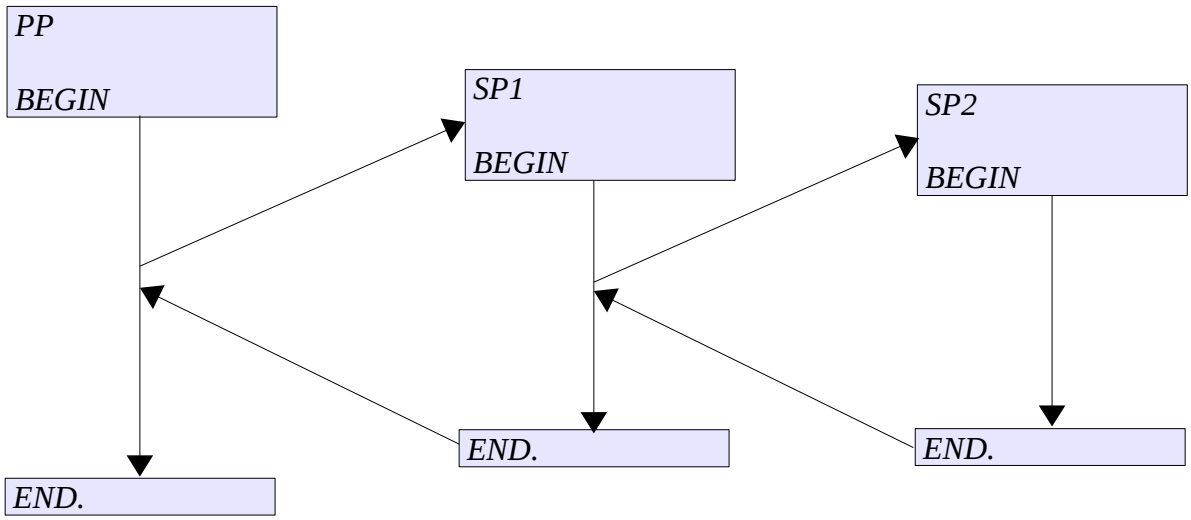
Dans le cas précédent, du moment que les trois factoriels se calculent de la même façon, on écrit une seule fonction *FACT* qui permet de calculer le factoriel d'une valeur naturelle quelconque *n* ($n! = 1 \times 2 \times 3 \times ... n$)

Pour calculer le nombre de combinaisons C_n^k, il suffira d'appeler la fonction *FACT* trois fois en transmettant *n*, *k* et *(n-k)* comme paramètre à la fonction.

1.2. Notion du Sous-Programme

Un sous-programme (procédure ou fonction) est un programme à l'intérieur d'un autre programme. Il possède la même structure que le programme principal. Il peut être appelé par le programme principal ou par un autre sous-programme pour réaliser un certain traitement et leur retourner des résultats (un ou plusieurs résultats).

Le schéma suivant illustre le mécanisme *d'appel* et *retour* des sous-programmes.



PP : Programme Principal

SP1 : Sous-Programme 1 END.

SP2 : Sous-Programme 2

Dans le langage PASCAL, on a l'habitude d'utiliser les *procédures* ***read*** et ***write*** pour réaliser respectivement des entrées et sorties (lectures et écritures).

En plus de ces deux procédures, on a aussi l'habitude d'utiliser les *fonctions* ***ln***, ***sin***, ***cos***, ***exp***, … pour calculer des valeurs et des expressions arithmétiques.

Donc, le langage PASCAL nous offres des procédures et fonctions prédéfinies. Si nous avons besoins d'un traitement que PASCAL n'offre pas dans sa bibliothèque, on peut définir nos propres procédures et fonctions.

La différence principale entre les procédures et les fonctions est que les procédures ne possède pas de valeurs (elles ne retournent pas directement de résultats). Par contre, les fonctions retournent directement une valeur (un résultat). Cependant, toute procédure peut être convertie en une fonction et vice-versa (toute fonction peut être convertie en procédure).

1.3. Structure Générale d'un Programme PASCAL comportant des Procédures et des Fonctions (Sous-Programmes)

La structure générale d'un programme PASCAL comportant des fonctions et des procédures est comme suit :

```
Program  <Nom_Programme>;
```

```
        uses wincrt;

        <Déclaration des étiquettes éventuelles>;
        <Déclaration des constantes éventuelles>;
        <Déclaration des types éventuels>;
        <Déclaration des variables éventuelles>;
        <Déclaration des fonctions>;
        <Déclaration des procédures>;

    BEGIN
        <Instruction 1>;
        <Instruction 1>;
        <Instruction 1>;
        ......
        ......
        ......
        <Instruction N>;
    END.
```

Ainsi, les sous-programmes (procédures et fonctions) sont déclarés dans la partie déclaration du programme principal avec les constantes, variables, types, étiquettes, etc.

1.4. Déclaration de sous-programme

En langage PASCAL, il y a deux types de *sous-programmes* : *Procédures* et *Fonctions*.

1.4.1. Déclaration d'une procédure

Une procédure possède la même structure que le programme principal, pour définir une procédure, on suit le modèle suivant :

```
Procedure <Nom_Procedure>[(<param1>:<type1>; <param2>:<type2>;… <paramN:typeN)];
      <Déclaration des constantes>;
      <Déclaration des variables>;
      <Déclaration des étiquettes>;
Begin
      <instruction 1>;
      <instruction 2>;
      <instruction 3>;
      .......
      .......
      <instruction N>;
End;
```

`<Nom_Procdure>` : identificateur de la procédure (il faut respecter les règles des identificateurs).

`<param1>, <pram2> … <paramN>` : sont des paramètre de la procédure.

`<type1>, <type2> … <typeN>` : sont les types respectifs des paramètres de la procédure.

Comme le programme principal, une procédure possède une partie déclaration et une partie instructions qui délimitée par ***Begin*** et ***End***

Ce qui entre crochets dans le modèle de définition de procédure, n'est pas obligatoire si on a pas besoin. Par exemple, on peut enlever les paramètres avec les parenthèse pour une procédure sans paramètres.

En résumé : Une procédure est déclarée en utilisant le mot clé ***Procedure***, elle possède éventuellement des paramètres. On peut avoir des procédures sans paramètres, ou avec un ou plusieurs paramètres. Elle possède aussi, éventuellement, une partie déclarations pour déclarer des constantes et/ou des variables (et aussi même des procédures/fonctions). Enfin, une procédure possède un corps qui contient les instructions à exécuter (le traitement que la procédure réalise).

Exemple

- Soit la procédure suivante :

```
procedure afficher;
begin
     writeln('Exemple de procédure ...');
end;
```

Cette procédure permet d'afficher toujours la chaîne de caractères *'Exemple de procédure …'*. Elle possède pas de paramètre et son traitement est figé (non paramétré). Donc elle réalise toujours la même chose de la même façon.

On peut améliorer cette procédure en paramétrant son traitement (l'instruction writeln), comme suit :

- Première amélioration de la procédure *afficher* :

```
procedure afficher (message : String);
begin
     writeln(message);
end;
```

Cette fois-ci, la procédure *afficher* permet d'afficher la chaîne de caractères contenue dans le paramètre message de type String. Pour cela, il faut passer ce qu'on veut afficher comme paramètre. Par exemple : `afficher('Hello world !!!')`.

On remarque que la procédure afficher, saute la ligne toujours après l'affichage de la variable message, on peut contrôler ce saut de ligne comme suit :

- deuxième amélioration de la procédure *afficher* :

```
procedure afficher(message:String ; sauter_Ligne:boolean);
begin
     write(message);
     if (sauter_ligne = TRUE) then
          writeln;
end;
```

Cette fois-ci, on passe un deuxième paramètre booléen qui indique si on saute la ligne ou non.

Remarques :

– Dans l'exemple précédent, on a vu trois forme de la même procédure *afficher*. Première sans paramètres, deuxième avec un seul paramètre chaîne de caractères et la dernière avec deux paramètres chaîne de caractères et booléen. Les différents paramètres permettent de généraliser le traitement de la procédure.

– Les paramètres vus jusqu'à maintenant sont dits avec ***passage par valeur*** (ou transmission par valeur). Il y aussi le **passage *par variable*** (transmission par variable).

– Une procédure ne retourne pas de valeur (par de résultat direct). On appelle directement la procédure sans l'affecter à une variable. Si on veut qu'une procédure nous transmis des résultats, on ajoute des paramètres par *transmission par variables*. (Voir plus loin dans le chapitre).

1.4.2. Déclaration d'une fonction

Comme les procédures, une fonction possède la même structure que le programme principal, à la différence, elle possède un type de retour (de résultat). Pour définir une fonction, on suit le modèle suivant :

```
Function <Nom_Funciton>[(<P1>:<T1>; <P2>:<T2>;… <Pn:Tn)] : <type_fonction>;
      <Déclaration des constantes>;
      <Déclaration des variables>;
      <Déclaration des étiquettes>;
Begin
      <instruction 1>;
      <instruction 2>;
      <instruction 3>;
      .......
      .......
      <instruction N>;
      <Nom_Fonction> := <resultat>;
End;
```

`<P1>, <P2>, … <Pn>`: les paramètres éventuels de la fonction.

`<T1>, <T2>, …, <Tn>`: les types respectifs des paramètres de la fonction.

`<type_fonction>` : type de la fonction. C'est-à-dire, le type de résultat de la fonction.

La dernière instruction dans le corps d'une fonction est toujours :

```
<Nom_Fonction> := <resultat> ;
```

Pour que la fonction aie la valeur correcte calculée dans le traitement spécifié dans la partie instructions de la fonction.

En résumé : Une fonction est déclarée en utilisant le mot clé **Function**, elle possède éventuellement des paramètres. On peut avoir des fonctions sans paramètres, ou avec un ou plusieurs paramètres. Elle possède aussi, éventuellement, une partie déclarations pour déclarer des constantes et/ou des variables (et aussi même des procédures/fonctions). La différence entre une fonction et une procédure est qu'une fonction possède un type de retour. Enfin, une fonction possède un corps qui contient les instructions à exécuter (le traitement que la fonction réalise). Une deuxième différence entre une procédure et une fonction est que cette dernière doit avoir toujours à la fin de son corps l'instruction `<Nom_Fonction> := <resultat>;` qui représente la valeur que cette fonction doit retourner (le résultat de la fonction).

Exemple

- Soit la fonction suivante :

```
function somme : integer;
begin
     somme := 5 + 7;
end;
```

Cette fonction permet de calculer toujours la somme des deux nombre *5* et *7* et elle retourne toujours le résultat *12*. Elle possède pas de paramètre et son traitement est figé (non paramétré). Donc elle réalise toujours la même chose de la même façon.

On peut améliorer cette fonction en paramétrant son traitement, comme suit :

- Amélioration de la fonction *somme* :

```
function somme (a, b : integer) : integer;
begin
     somme := a + b;
end;
```

Cette fois-ci, la fonction réaliser la somme de deux nombre quelconque *a* et *b*. Lors de l'appel à cette fonction, on doit passer des paramètres, par exemple : *somme (10, 18)*. qui retourne la valeur *28,* qu'on peut soit afficher soit affecter à une autre variable.

- Une autre exemple du calcul du factoriel :

```
function fact(n:integer) : integer;
     var
          i, f:integer;
begin
     f:=1;
     for i:=1 to n do
          f := f*i;
     fact := f; {toujours la dernière instruction <nom_fonction> := <résultat>}
end;
```

La fonction *fact* prend comme paramètre un nombre n et retourne son factoriel. La dernière instruction de la fonction est : `fact := f;`

Remarques :

- La différence entre une procédure et une fonction et le type de retour. Une procédure ne possède pas de type de retour et une fonction possède un type de retour.
- Toute fonction doit avoir, dans son corps, comme dernière instruction : <nom_fonction> := <result> ;
- Toute procédure peut être convertie en une fonction et toute fonction peut être converti en une procédure.

1.4.3. Notion de paramètres formels et paramètres effectifs

Les paramètres ***formels*** sont les paramètres utilisé dans la déclaration du sous-programme (procédure ou fonction). Para exemple, dans la fonction *fact* déclarée ci-dessus, l'entier *n* représente un paramètre formel. (les paramètres formels sont séparé par des ***point-virgules*** **;**).

Un autre exemple, dans la procédure afficher possédant deux paramètres : message et sauter_Ligne représentent des paramètres formels.

Lors de l'appel à une fonction ou programme, nous devant donner des valeurs ou des variables à la place des paramètres formel : ce sont les paramètre ***effectif***. Par exemple, si on appel la fonction *fact* :

- *fact(5) :* **5** c'est un paramètre effectif
- *fact(2+5) :* l'expression ***2+5*** c'est paramètre effectif
- *afficher ('bonjour', true)* : les deux paramètres ***'bonjour'*** et ***true*** sont des paramètres effectifs.

Donc, les *paramètres formels* sont les paramètres utilisés dans la déclaration des procédures et fonctions. Par contre, les *paramètres effectifs* sont les paramètres utilisées lors de l'appel aux procédures et fonctions. (les paramètres formels sont séparé par des ***virgules*** **,**).

1.5. Passage de paramètres (Transmission de paramètres)

La communication des fonctions et procédures entre elles et avec le programme principal se fait par le mécanisme de transmission de paramètres (passage des paramètres) ; on distingue quatre types de transmission de paramètres :

- ✗ les paramètres transmis par valeurs ;
- ✗ Les paramètres transmis par variables ;
- ✗ les paramètres fonction ;
- ✗ les paramètres procédure.

1.5.1. Transmission des paramètres par valeur

Pour la transmission de paramètres par valeur, il suffit de déclarer les paramètres formels sans utiliser le mot clé ***VAR***. Par exemple dans la fonction *fact* définit comme suit :

```
function fact(n:integer) : integer;
```

Le paramètre formel *n* est *transmis par valeur*. Dans le paramètre effectif correspondant, on peut passer une valeur constante, une expression, une variables, etc. À condition que le paramètre formel et le paramètre effectif correspondant soit du même type. Par exemple :

```
fact(5) ; fact(4+6) ; fact(a+y-4) ;
```

Dans la transmission des paramètres par valeur, les changements réalisés sur les paramètres formels n'affectent pas les paramètres effectifs.

1.5.2. Transmission des paramètres par variables

On utilise la transmission par variable lorsqu'on veuille que la variable du paramètre formel dans la fonction ou procédure affecte le paramètre effectif correspondant au niveau du programme appelant. Dans ce cas, le paramètre formel doit être précédé du mot-clé ***VAR***. Le paramètre effectif correspondant ***doit être, obligatoirement, une variable***.

En d'autres termes, on utilise une transmission par variable lorsqu'on veut recueillir le résultat dans cette variable. ***(Variable résultat)***.

Exemple :

Soit la procédure Produit suivante :

```
Procedure Produit(a, b : real ; var p : real);
begin
     p := a*b;
end;
```

Dans cette procédure, les deux paramètres *a* et *b* sont définis sans utiliser le mot clé ***VAR***, donc la transmission (lors de l'appel à cette procédure) sera par valeur. Par contre, le paramètre *p* est défini par le mot clé ***VAR***, donc la transmission sera par variable (le paramètre effectif doit être une variable).

Exemple d'appels : *Produit (5, 6, x); 5* et *6* sont des paramètres effectifs sont transmis par valeur. Le paramètre effectif x est transmis par variables.

1.5.3. Transmission des paramètres par fonction

Dans ce cas, un nom de fonction est utilisé comme paramètre. On doit alors indiquer que le paramètre formel correspondant est une fonction le précédant du mot clé ***FUNCTION***.

Exemple :

```
program Params_Fonctions;
var
      ......
      PROCEDURE Proc1(..., Function F:real, ...);
            var ...
      Begin
            ....
            ....
      End;

      FUNCTION G(...) : real;
            var ...
      Begin
            ....
            ....
      End;
BEGIN {Début du programme principal}
      ...
      Proc1(..., G, ...); {Appel de la procédure Proc1 avec un paramètre fonction}
      ...
END. {Fin du programme principal}
```

1.5.4. Transmission des paramètres par procédure

Dans ce cas, un nom de procédure est utilisé comme paramètre. On doit alors indiquer que le paramètre formel correspondant est une procédure en le précédant du mot-clé ***PROCÉDURE***.

1.6. Exercices d'Application

1.6.1. Calcul de combinaisons

Écrire un programme PASCAL qui permet de calculer le nombre de combinaisons de k éléments à partir de n éléments. Tel-que :

$$C_n^k = \frac{n!}{k! \times (n-k)!} \quad \textbf{si } n >= k >= 1$$

$$C_n^k = 0 \quad \textbf{si } n < k$$

```
Program combinaisonSansFonction;
  uses wincrt;

  var
    n, k, c : integer;
    nf, kf, nkf : integer;
    i:integer;

Begin {Début du programme principal}
  read(n, k);

  nf := 1;
  For i:=1 To n Do
      nf := nf * i;

  kf := 1;
  For i:=1 To k Do
      kf := kf * i;

  nkf := 1;
  For i:=1 To (n-k) Do
      nkf := nkf * i;

  c := kf / (nf * nkf);
  write('Combinaison n k = ', c);
End. {Fin du programme principal}
```

```
Program combinaisonAvecFonction;
  uses wincrt;
  var
    n, k, c : integer;
    nf, kf, nkf : integer;
    i:integer;

  Function factoriel(n:integer):integer;
    var
      j, f:ineger;
  Begin
     f:=1;
     for j:=1 to n do
        f:= f*j;
     factoriel := f;
  End;

BEGIN {Début du programme principal}
  read(n, k);

  nf := factoriel(n);
  kf := factoriel(k);
  nkf := factoriel(n-k);

  c := kf / (nf * nkf);
  write('Combinaison n k = ', c);
END. {Fin du programme principal}
```

Variables globales du programme

Paramètre formel

Variables locales

Dernière instruction de la fonciton

Appels à la fonction factoriel, avec les paramètre effectifs: n, k et (n-k)