Chapitre IV
Sous-Programmes : Procédures & Fonctions

Sommaire

Cours Elearning :
https://elearning.univ-bejaia.dz/course/view.php?id=2749

Page facebook :
https://www.facebook.com/InitiationAlgoProgrammation/

La chaîne Youtube :
https://www.youtube.com/c/AlgoProgrammation1èreAnnéeTechnologie

La playlist sur le langage C :
https://youtube.com/playlist?list=PLwHHAvorm5F-tL9EXDEH0miOKmAj7iUTU

Adapté par : Redouane OUZEGGANE
rouzeggane@gmail.com - redouane.ouzeggane@univ-bejaia.dz

Chapitre IV : Sous-Programmes - Procédures et Fonctions

IV.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 faciles 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 (* $n! = 1 \times 2 \times 3 \times ... n$ *)*

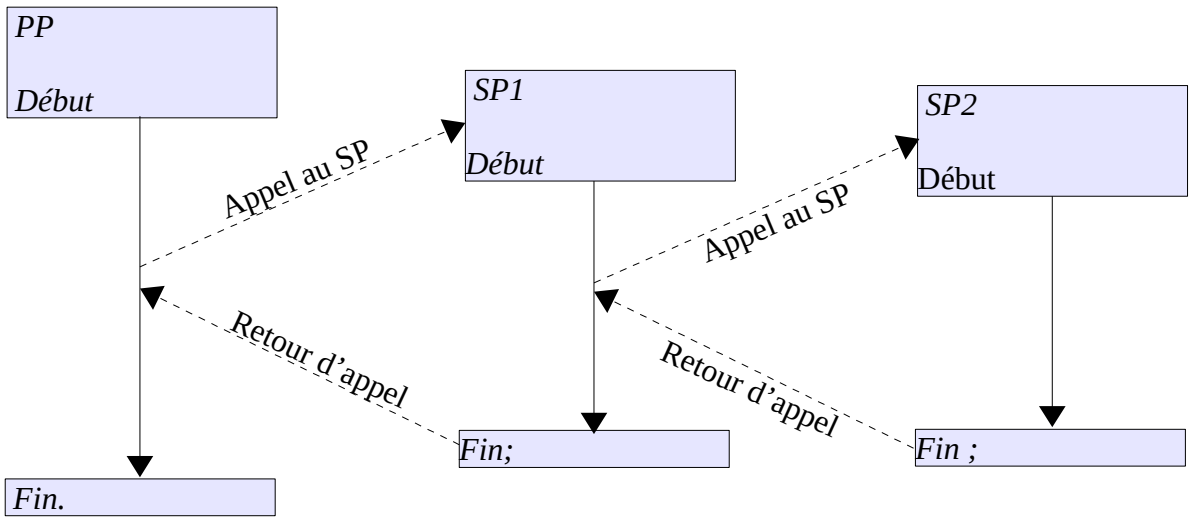
Pour calculer le nombre de combinaisons C_n^k, il suffira d'appeler la fonction *FACT* trois fois en transmettant *n*, *k* et *(n-k),* respectivement, comme paramètre effectif à la fonction.

IV.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 : ***à éviter***).

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



PP : Programme Principal

SP1 : Sous-Programme 1

SP2 : Sous-Programme 2

Dans le langage C, nous avons l'habitude d'utiliser les *procédures/fonctions* ***scanf*** et ***printf*** pour réaliser respectivement des entrées et sorties (lectures et écritures). En plus de ces deux sous-programme, nous avons aussi l'habitude d'utiliser les *fonctions* ***log***, ***sin***, ***cos***, ***exp***, … pour calculer des valeurs et des expressions arithmétiques (voir *math.h*). Donc, le langage C nous offres des procédures et fonctions prédéfinies. Si nous avons besoins d'un traitement que le langage C n'offre pas dans sa bibliothèque, on peut définir nos propres sous-programme.

Un sous-programme est une suite d'instructions possédant un nom (*identificateur*) et des paramètres d'entrées et de sortie afin de communiquer avec lui.

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

IV.3. Structure Générale d'un Algorithme / Programme C comportant des Procédures et des Fonctions (Sous-Programmes)

La structure générale d'un algorithme (et un programme en langage C) comportant des fonctions et des procédures est comme suit :

```
Algorithme  <Nom_Programme>;
      //Déclarations de :
      <Types Personnalisés>;
      <Constantes & Variables>;
      <Sous-programme>;

Début
      <Instruction 1>;
      <Instruction 2>;
      <Instruction 3>;
      ......
      <Appels-Sous-Programmes>
      ......
      <Instruction N>;
Fin.
```

```
#include <stdio.h>
//Déclarations de :
<Types Personnalisés>
<Constates & Variables globales>
<Sous-Programme>

int main()
{
      <Instruction 1>;
      <Instruction 2>;
      ......
      <Appels-Sous-Programmes>
      ......
      <Instruction N>;
}
```

Ainsi, les sous-programmes (procédures et fonctions) sont déclarés dans la partie *déclaration* de l'algorithme (avec les constantes, variables, types , etc) et ils sont appelés dans la partie *instruction*.

IV.4. Déclaration de sous-programme

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

IV.4.1. Déclaration d'une procédure

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

```
Procédure <Nom_Procedure>(<param1>:<type1>; <param2>:<type2>;… <paramN>:<typeN>);
      <Constantes et/ou variables>;
Début
      <instruction 1>;
      <instruction 2>;
      <instruction 3>;
      .......
      .......
      <instruction N>;
Fin;
```

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

`<param1>, <pram2> … <paramN>` : sont des paramètre de la procédure (*paramètres formels*).

`<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 est délimitée par ***Début*** et ***Fin.***

En langage C, une procédure peut être écrite comme suit :

```
void <Nom_Procedure>(<type1> <param1>, <type2> <param2>, … , <typeN> <paramN>);
{
      <Constantes et/ou variables>;

      <instruction 1>;
      <instruction 2>;
      <instruction 3>;
}
```

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

Les paramètres utilisés dans la déclaration de la procédure sont dits : *paramètres formels*. Nous avons, selon le mode de transmission, deux types de paramètres : paramètres d'entrée (passage par valeur) et paramètres d'entrée / sortie (passage par adresse).

Exemple

– Soit la procédure suivante :

```
procedure afficher;
Début
   Écrire('Exemple de procédure ...');
Fin;
```

```
void afficher()
{
    printf("Exemple de procédure ...");
}
```

Cette procédure permet d'afficher toujours la chaîne de caractères '**Exemple de procédure …**'. Elle ne 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 :

```
procedure afficher(msg:Chaine_C);
Début
   Écrire(msg);
Fin;
```

```
void afficher(char msg[100])
{
    printf("%s", msg);
}
```

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

Remarques :

- Dans l'exemple précédent, on a vu deux formes 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 paramètres d'entrée ou ***passage par valeur*** (ou transmission par valeur). Il y aussi les *paramètre de sortie* ou **passage *par adresse*** (transmission par adresse).
- 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 de sortie (*transmission par variables*) (Voir *IV.5*).

IV.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 :

```
Fonction <Nom_Funciton>(<P1>:<T1>; <P2>:<T2>;… <Pn:Tn) : <type_fonction>;
      <Constantes et/ou variables>;
Début
      <instruction 1>;
      <instruction 2>;
      <instruction 3>;
      .......
      .......
      <instruction N>;
      retourner <resultat>;
Fin;
```

`<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 : `retourner <resultat>;`

Cette dernière instruction permet à la fonction d'avoir la valeur correcte calculée dans le traitement spécifié dans la partie instructions de la fonction.

En langage C, on écrit comme suit une fonction :

```
<type_fonction> <Nom_Fonction>(<T1> <p1>, <T2> <p2>, … , <Tn> <pN>)
{
      <Constantes et/ou variables>;

      <instruction 1>;
      <instruction 2>;
      <instruction 3>;

      return <resultat>;
}
```

En résumé, Une fonction est déclarée en utilisant le mot clé ***Fonction***, 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 (*données locales*). 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 `Retourner <resultat>;` qui représente la valeur que cette fonction doit retourner.

Exemple

- Soit la fonction suivante :

```
fonction somme : entier;
début
    retourner (5 + 7);
fin;
```

```
int somme()
{
    return (5 + 7);
}
```

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 paramètres 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 :

```
fonction somme(a,b:entier): entier;
début
    retourner (a + b);
fin;
```

```
int somme(int a, int b)
{
      return (a+b);
}
```

Cette fois-ci, la fonction réalise la somme de deux nombres 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 être soit affiché soit affecté à une autre variable.

- Une autre exemple du calcul du factoriel :

```
fonction fact(n:entier):entier;
    var
        i, f:entier;
Début
    f ← 1;
    pour i←2 à n faire
        f ← f*i;
    retourner f; {toujours la dernière instruction retourner <résultat>}
Fin;
```

La fonction *fact* prend comme paramètre un nombre *n* et retourne son factoriel. La dernière instruction de la fonction est : `retourner 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 : retourner <result>;
- Toute procédure peut être convertie en une fonction et, vice-versa, toute fonction peut être converti en une procédure.

IV.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.

Un autre exemple, dans la procédure afficher possédant un paramètres message qui représente un paramètre formel.

Lors de l'appel à une fonction ou procédure, nous devant donner des valeurs ou des variables à la place des paramètres formel : ce sont les paramètres ***effectifs***. 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')* : le paramètre ***'bonjour'*** est un paramètre effectif.

Donc, les *paramètres formels* sont les paramètres utilisés dans la déclaration des sous-programmes. Par contre, les *paramètres effectifs* sont les paramètres utilisées lors de l'appel aux sous-programmes..

IV.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 deux types de transmission de paramètres :

- ✗ les paramètres transmis par valeurs : paramètre d'entrées;
- ✗ Les paramètres transmis par adresse : paramètres d'entrées/sortie;

IV.5.1. Transmission des paramètres par valeur (Paramètres d'entrée)

Pour la transmission de paramètres par valeur, il suffit de déclarer les paramètres formels ***sans aucune indiction (***ou utiliser la lettre ***E*** avant le paramètre : paramètre d'entrée***)***. Par exemple dans la fonction *fact* définit comme suit :

```
fonction fact(n:entier): entier;
```

En C :

```
int fact(int n);
```

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

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

Dans la transmission des paramètres par valeur, les changements réalisés sur les paramètres formels n'affectent pas les paramètres effectifs : *paramètre d'entrée uniquement*.

IV.5.2. Transmission des paramètres par adresse (Paramètres Entrée/Sortie)

On utilise la transmission par adresse lorsqu'on veut 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 de l'indication ***E/S*** (C'est à dire : Entrée / Sortie). Le paramètre effectif correspondant ***doit être, obligatoirement, une variable***. Cependant, en langage C, le paramètre d'entrée/sortie est implémenté en utilisant le pointeur, et le paramètre effectif doit être l'adresse mémoire de la variable.

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

Par exemple, soit la procédure Produit suivante :

```
Procédure produit(a, b:réel; E/S p:réel);
début
      p := a*b;
fin;
```

```
void produit(float a, float b, float *p)
{
      *p = a*b;
}
```

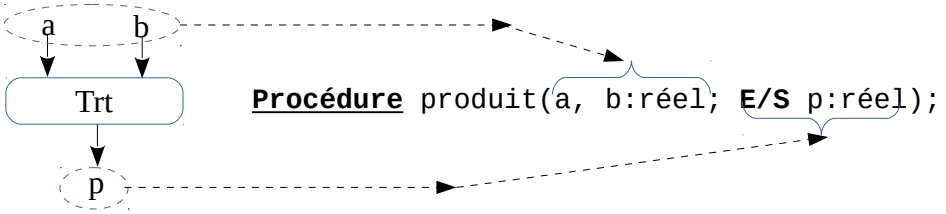
Dans cette procédure, les deux paramètres *a* et *b* sont définis sans utiliser aucune indication, donc la transmission (lors de l'appel à cette procédure) sera par valeur ; sont des *paramètres d'entrée*. Par contre, le paramètre *p* est défini par l'indication ***E/S***, donc la transmission sera par adresse : *paramètre d'entrée / sortie*.

Exemple d'appels : `produit(5, 6, x);` *5* et *6* sont des paramètres effectifs transmis par valeur (Paramètres d'entrées), et le paramètre effectif x est transmis en mode E/S.

En algorithme, l'appel : `produit(5, 6, x);` devient en langage C : `produit(5, 6, &x);`

`&x` : *signifie l'adresse mémoire de la variable x (transmission par adresse).*

La procédure *produit* peut être schématisée comme suit :



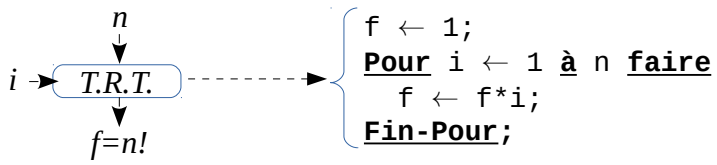
Un exemple de la procédure produit est sur le lien : https://onlinegdb.com/5dezKQ8Tr3

IV.6. Transformation procédure ↔ fonction

IV.6.1. Transformer une procédure à une fonction

Nous allons voir, dans cette section, comment convertir une procédure à une fonction. Prenons l'exemple du calcul factoriel. On sait que : $f = n! = 1 \times 2 \times 3 \times ... \times n = \prod_{i=1}^{n} i$.

Ce calcul est schématisé comme suit :



Dans ce schéma, et comme nous l'avons vu en algorithmique (voir *chapitre 2*), nous avons *n* est une variable d'entrée, *f* une variable de sortie et *i* est une variable intermédiaire. Par ailleurs, dans les sous-programme procédure, *n* représente un paramètre d'entrée (*transmission par valeur*), *f* est un paramètre de sortie (mode *Entrée/Sortie*) (*transmission par adresse*) et i une variable locale dans la procédure fact. Pour cela, nous pouvons écrire l'entête de la procédure *fact*, qui permet de calculer *f=n!* , comme suit :

```
Procédure fact(n:entier; E/S f:entier);
```

Le code complet de *la déclaration* de la procédure fact peut êtes écrit comme suit :

ALGORITHME

```
Procédure fact(n:entier; E/S f:entier);
  Variable
    i : entier;
Début
   f ← 1;
   Pour i ← 1 à n faire
      f ← f*i;
   Fin-Pour;
Fin.
```

LANGAGE C

```
void fact(int n, int *f)
{
   int i;

   *f = 1;
   for (i=1; i<=n; i++)
      *f *= i;
}
```

Remarques :

- Un sous-programme peut utiliser les variables globales du programme principal (de l'algorithme). Ceci dit, cette pratique est *très déconseillée* : parmi les critères de bon code des sous-programmes, ces derniers (sous-programmes) ne doivent pas utiliser les variables globales : une procédure ou une fonction utilise uniquement ses propres paramètres ou ses données locales (variables locales)

- Dans un sous-programme, on peut avoir un paramètre ou une variable locale avec le même nom

qu'une variable globale : il n'y a aucune relation entre le paramètre *n* et la variable globale *n*.

Pour l'appel à la procédure fact, le tableau suivant montre quelques exemple d'appel :

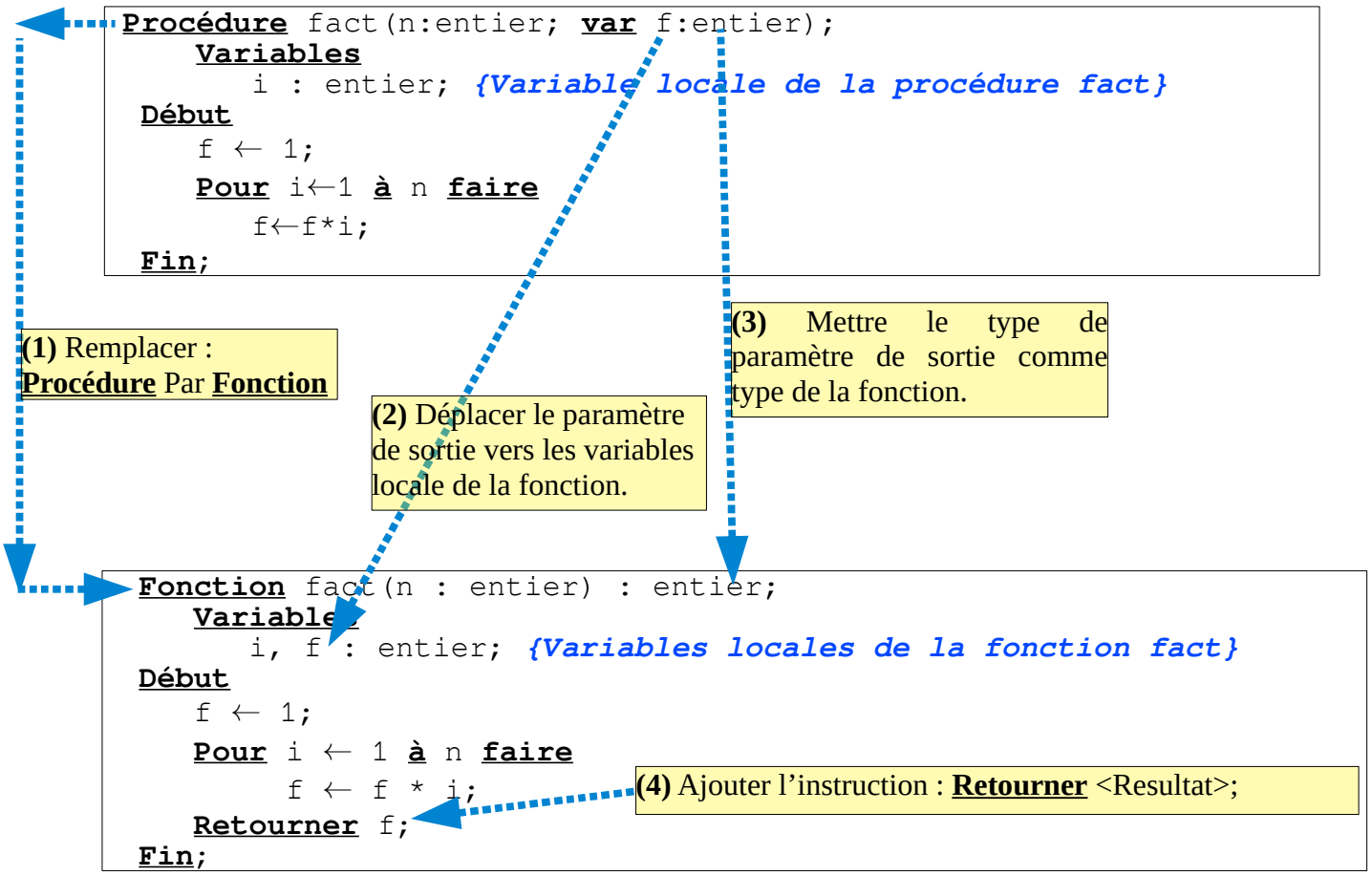
Formules Mathématiques	***Appels à la procédure fact***
M = 9!	`fact(9, M);`
x = (n - k)!	`fact(n-k, x);`
$C = \frac{n!}{k! \times (n-k)!}$	`fact(n, f1); fact(k, f2); fact(n-k, f3);` `C ← f1 / ( f2 * f3 );`

Maintenant, nous sommes arrivé à la question de cette section : comment transformer une procédure à une fonction ?

Cette transformation se fait à deux niveau :

- Au niveau de la déclaration
- Au niveau de l'appel

Au niveau de la déclaration, le schéma suivant illustre comment réaliser la transformation :



Donc, pour transformer une procédure à une fonction, nous appliquons 4 étapes :

1- remplacer le mot clé : **Procédure** par **Fonction** ;

2- déplacer le paramètre de sortie comme variable locale de la fonction

3- mettre le type de paramètre de sortie comme type de la fonction

4- en fin, ajouter l'instruction **Retourner** <Resultat>; tel-que <Resultat> et le paramètre de sortie de la procédure.

Au niveau de l'appel, les appels deviennent comme illustré par le tableau ci-dessous :

Appel à la procédure fact	*Appel à la fonction fact*
`fact(9; M);`	`M ← fact(9)`
`fact(n-k; x);`	`x ← fact(n-k);`

`fact(n, f1);`
`fact(k, f2);`
`fact(n-k, f3);`
`C ← f1 / ( f2 * f3 );`
	`f1 ← fact(n);`
	`f2 ← fact(k);`
	`f3 ← fact(n-k);`
	`C ← f1 / ( f2 * f3 );`

`fact(n, f1);`
`fact(k, f2);`
`fact(n-k, f3);`
`C ← f1 div ( f2 * f3 );`
	`C←fact(n) div (fact(k)* fact(n-k));`

Vous remarquez, que chaque appel à une procédure, devient un appel à une fonction dans une affectation, comme suit :

Appel Procédure		Appel Fonction
fact(P1, P2);	⇒	P2 ← *fact*(P1);

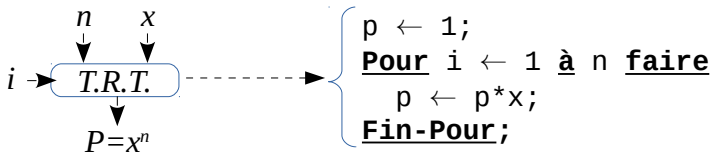
Donc, une fonction est, souvent, appelée dans la partie droite d'une affectation, puisque, en réalité, une fonction est une variable spéciale dont sa valeur est calculée par ses propres instructions.

IV.6.2. Transformer une fonction à une procédure

Pour transformer une fonction à une procédure, on réalise les étapes précédentes dans le sens inverse. Pour illustrer ça, on prends l'exemple de calcule de puissance :

On sait que : $p = x^n = x \times x \times x \times ... \times x (n\,fois) = \prod_{i=1}^{n} x$.

Ce calcul est schématisé comme suit :



Dans ce schéma, *n* et *x* représentent des paramètres d'entrée (*transmission par valeur*), *P* est la variable résultat (dans une procédure, P représente un paramètre de sortie) qui est contient la valeur que la fonction *Puissance* doit retourner. Donc, P représente une variable locale à la fonction *Puissance*. La variable *i* est aussi une variable locale de la fonction *Puissance*. Pour cela, nous pouvons écrire l'entête de la fonction *Puissance*, qui permet de calculer x^n , comme suit :

```
Fonction puissance(n:entier; x:réel):réel;
```

Vous remarquez que la fonction *Puissance* est de type réel (elle retourne un résultat réel).

Le code complet de *la déclaration* de la procédure fact peut êtes écrit comme suit :

ALGORITHME

```
Fonction puissance(n:entier; x:réel):réel;
  Variable
    i : entier;
    p : réel;
Début
   p ← 1;
   Pour i ← 1 à n faire
      p ← p*x;
   Fin-Pour;

   retourner P;
Fin.
```

LANGAGE C

```
float puissance(int n, float x)
{
   int i;
   float p;

   p = 1;
   for (i=1; i<=n; i++)
      p *= x;

   return p;
}
```

Formules Mathématiques	*Appels à la fonction puissance*
$y = 2.5^6$	`y ← Puissance(2.5, 6);`
$x = (n - k)^m$	`x ← Puissance(n-k, m);`

$Z = \frac{(x^3+y^5)}{(x+y-3)^{10}}$	`p1 ← Puissance(x, 3);`
	`p2 ← Puissance(y, 5);`
	`p3 ← Puissance(x+y-3, 10);`
	`Z ← (p1 + p2) / p3`

Pour transformer une fonction à une procédure, nous appliquons les quatre étapes suivante :

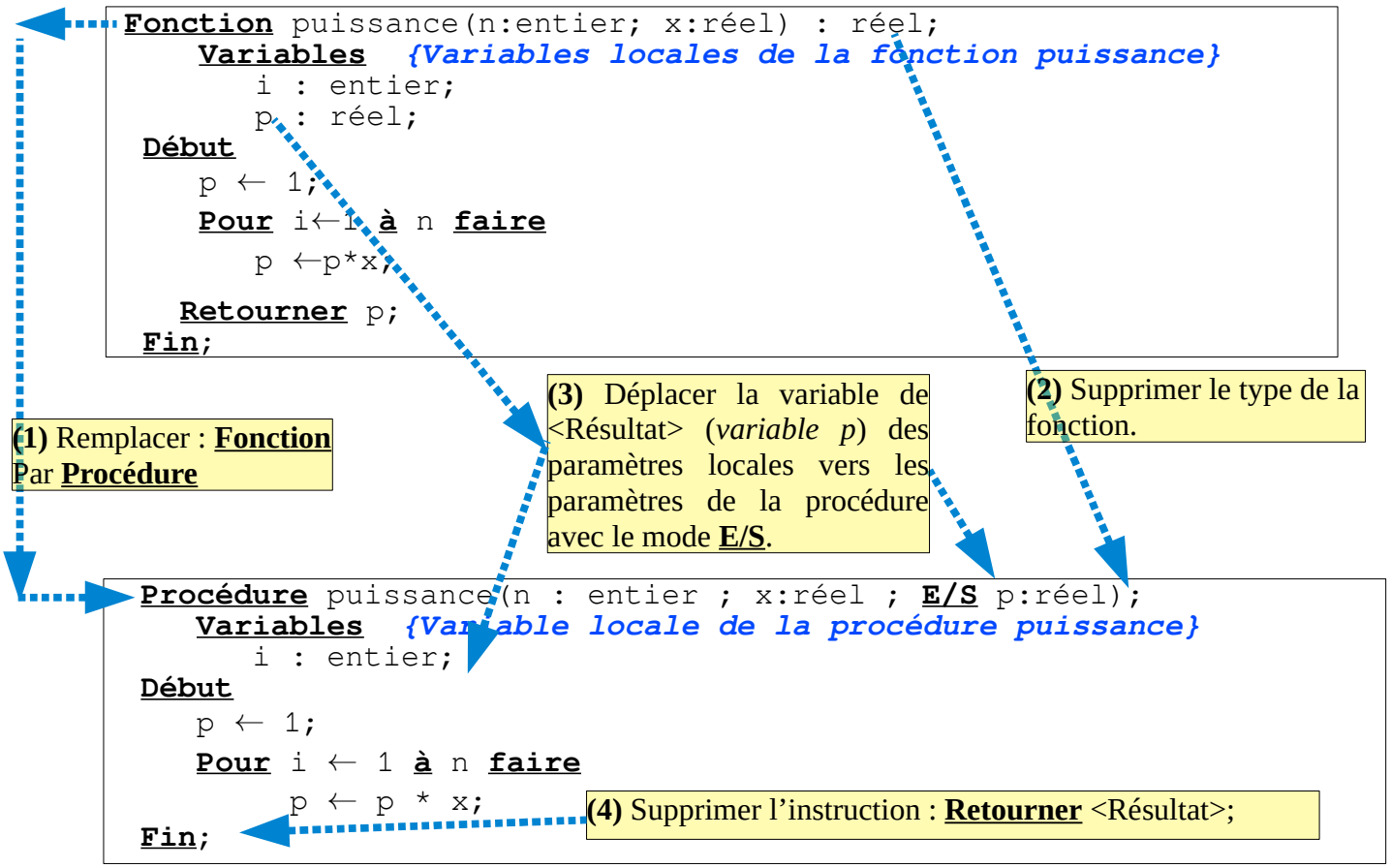
1- remplacer le mot clé : **Fonction** par **Procédure**;

2- supprimer le type de la fonction

3- déplacer la variable résultat de la fonction, vers les paramètres de la procédure, en créant un paramètre de entrée/sortie (passage par adresse).

4- en fin, supprimer l'instruction **Retourner** <Resultat>; tel-que <Resultat> et la variable contenant le résultat de la fonction (e qui devient t le paramètre de sortie de la procédure).

La figure suivante, illustre les étapes ci-dessus citées :



Pour les appels, ça sera comme suit :

Appels à la fonction Puissance	*Appels à la Procédure Puissance*
`y ← Puissance(2.5, 6);`	`Puissance(2.5, 6, y);`
`x ← Puissance(n-k, m);`	`Puissance(n-k, m, x);`

`p1 ← Puissance(x, 3);`
`p2 ← Puissance(y, 5);`
`p3 ← Puissance(x+y-3, 10);`
`Z ← (p1 + p2) / p3`
	`Puissance(x, 3, p1);`
	`Puissance(y, 5, p2);`
	`Puissance(x+y-3, 10, p3);`
	`Z ← (p1 + p2) / p3;`

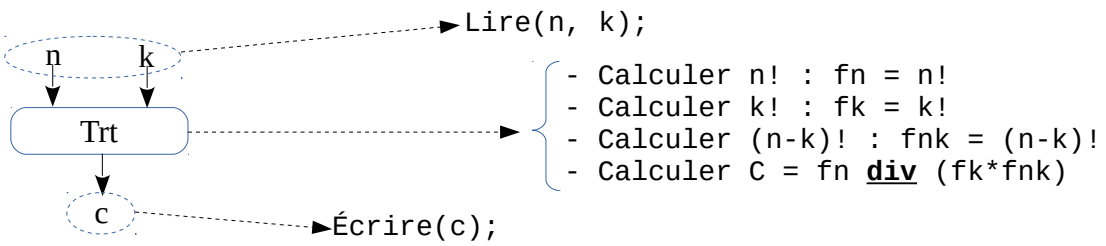
IV.7. Exercices d'Application

IV.7.1. Calcul de combinaisons (Fonction)

Écrire deux algorithmes (le premier sans fonction et le deuxième avec fonction) qui permettent de calculer le nombre de combinaisons de k éléments à partir de n éléments. Tel-que :

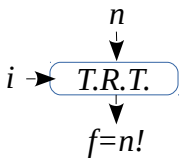
$$C_n^k = \frac{n!}{k! \times (n-k)!} \quad \textbf{si} \quad n \geq k >= 1$$

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



Le premier algorithme, sans fonctions, dans lequel nous allons écrire le code de calcul de factoriel 3 fois. Le deuxième algorithme, avec fonctions, on écrit le code du factoriel une seul fois, et on l'appel trois fois pour calculer *n!*, *k!* et *(n-k)!*

```
Algorithme combinaisonSansFonction;
  variables
    n, k, c : entier;
    nf, kf, nkf : entier;
    i : entier;

DEBUT //Début de l'algorithme
  Lire(n, k);

  nf ← 1;
  Pour i←1 à n faire
      nf ← nf * i;
  Fin-Pour;
  kf ← 1;
  Pour i←1 à k faire
      kf ← kf * i;
  Fin-Pour;
  nkf ← 1;
  Pour i←2 à (n-k) faire
      nkf ← nkf * i;
  Fin-Pour;
  c ← nf div (kf * nkf);

  Écrire('Combinaison n k = ', c);
FIN. //Fin de l'algorithme
```




```
Algorithme combinaisonAvecFonction;
  variables
    n, k, c : entier;
    nf, kf, nkf : entier;

Fonction fact(n:entier):entier;
  variables
    j, f:entier;
Debut
   f←1;
   Pour j←2 à n faire
      f ← f*j;
   retourner f;
Fin;

DÉBUT //Début de l'algorithme
  Lire(n, k);

  nf ← fact(n);
  kf ← fact(k);
  nkf ← fact(n-k);

  c ← nf div (kf * nkf);

  Écrire('Combinaison n k = ', c);
FIN. //Fin de l'algorithme
```

Variables globales du programme

Paramètre formel

Variables locales

Dernière instruction de la fonciton

3 appels à la fonction *factoriel*, pour calculer : *n!* *k!* et *(n-k)!* respectivement

Voir le lien : https://onlinegdb.com/LRXYKywTZT pour le programme en code C correspondant.

Le déroulement du deuxième algorithme (l'appel à la fonction factoriel) avec *n=5* et *k=3* :

Instructions	***Variables***									
	Variables Globales						***La fonction fact***			
	n	***k***	***nf***	***kf***	***nkf***	***C***	***n***	***i***	***f***	***fact***
Lire(n, k)	**5**	**3**				/	La fonction *fact* est désactivée			
fn← fact(n) ;(l'appel à fact avec le paramètre n=5)	**5**	``								
=> La transmission des paramètres → n=5 *f←1* *for i=1 f←f*i → f=01*1 = 1* *for i=2 f←f*i → f=01*2 = 2* *for i=3 f←f*i → f=02*3 = 6* *for i=4 f←f*i → f=06*4 = 24* *for i=5 f←f*i → f=24*5 = 120* ***retourner** f; ⇒ **fact = 120***	L'algorithme en attente de la réponse de la fonction *fact*						**5**	/ 1 2 3 4 5	/ 1 1 2 6 24 120	/ **120**
fn← 120; {fact(n) est remplacé par 120}	``	``	**120**				La fonction *fact* est désactivée			
fk ← fact(k) ; (l'appel à fact avec le paramètre k=3)	**5**	**3**	``	/	/					
=> La transmission des paramètres → n=3 *f←1* *for i=1 f←f*i → f=1*1 = 1* *for i=2 f←f*i → f=1*2 = 2* *for i=3 f←f*i → f=2*3 = 6* ***retourner** f; ⇒ **fact = 6***	L'algorithme en attente de la réponse de la fonction *fact*						**3**	/ 1 2 3	/ 1 1 2 6	/ **6**
fk← 6; {fact(n) est remplacé par 6}			**120**	**6**	/		La fonction *fact* est désactivée			
fnk← fact(n-k) ;(l'appel à fact avec Par. n-k=2)	``	``	``	``	/					
=> La transmission des paramètres → n=2 *f←1* *for i=1 f←f*i → f=1*1 = 1* *for i=2 f←f*i → f=1*2 = 2* ***retourner** f; ⇒ **fact = 2***	L'algorithme en attente de la réponse de la fonction *fact*						**2**	/ 1 2	/ 1 1 2	/ **2**
fnk← 2; {fact(n-k) est remplacé par 2}			**120**	**6**	**2**		La fonction *fact* est désactivée			
*C← 120 **div** (6 * 2) → C = 120 div 12 = 10* *Écrire(c);*	``	``	``	``	``	**10** **10**				

L'algorithme affiche C = 10.

IV.7.2. Calcul de combinaisons (Procédure)

Nous allons transformer l'algorithme précédent (avec fonction) à un algorithme avec procédure.

La fonction fact devient une procédure fact.

ALGORITHME (FONCTION FACT)

```
Algorithme combinaisonAvecFonction;
  variables
    n, k, c : entier;
    nf, kf, nkf : entier;

Fonction fact(n:entier):entier;
  variables
    j, f:entier;
Debut
   f←1;
   Pour j←2 à n faire
      f ← f*j;
   retourner f;
Fin;

DÉBUT //Début de l'algorithme
  Lire(n, k);

  nf ← fact(n);
  kf ← fact(k);
  nkf ← fact(n-k);

  c ← nf / (kf * nkf);

  Écrire('Combinaison n k = ', c);
FIN. //Fin de l'algorithme
```

*Appliquer les étapes de transformation d'une fonction vers procédure (**Déclaration**)*

*Appliquer les étapes de transformation d'une fonction vers procédure (**Appels**)*

ALGORITHME (PROCÉDURE FACT)

```
Algorithme combinaisonAvecProcedure;
  variables
    n, k, c : entier;
    nf, kf, nkf : entier;

Procédure fact(n:entier; E/S entier);
  variables
    j:entier;
Debut
   f←1;
   Pour j←2 à n faire
      f ← f*j;
   //On a supprimé : retourner f;
Fin;

DÉBUT //Début de l'algorithme
  Lire(n, k);

  fact(n, nf);
  fact(k, kf);
  fact(n-k, nkf);

  c ← nf / (kf * nkf);

  Écrire('Combinaison n k = ', c);
FIN. //Fin de l'algorithme
```

Nous avons appliqué les quatre étapes de transformation d'une fonction à une procédure pour avoir la déclaration de la procédure *fact*. Aussi, au niveau de l'appel, l'appelle à chaque fonction fact a été transformé à un appel à une procédure :

Appel Fonction		Appel Procédure
$nf \leftarrow fact(n);$	$\Rightarrow$	$fact(n, nf);$
$kf \leftarrow fact(k);$	$\Rightarrow$	$fact(k, kf);$
$nkf \leftarrow fact(n\text{-}k);$	$\Rightarrow$	$fact(n\text{-}k, nkf);$

Nous allons réaliser le déroulement de l'algorithme `combinaisonAvecProcedure` avec *n=5* et *k=3* :

Instructions	***Variables***								
	Variables Globales						***La procédure fact***		
	n	***k***	***nf***	***kf***	***nkf***	***C***	***n***	***i***	***f (E/S)***
Lire(n, k)	**5**	**3**	/	/	/	/	La procédure *fact* est désactivée		
fn← fact(n) ;(l'appel à fact avec le paramètre n=5)	**5**	``							
=> La transmission des paramètres → n=5 *f←1* *for i=1 f←f*i → f=01*1 = 1* *for i=2 f←f*i → f=01*2 = 2* *for i=3 f←f*i → f=02*3 = 6* *for i=4 f←f*i → f=06*4 = 24* *for i=5 f←f*i → f=24*5 = 120 (**Sortie**)*	L'algorithme en attente de la réponse de la procédure *fact*						**5**	/ 1 2 3 4 5	/ 1 1 2 6 24 **120**
fn← 120; {fact(n) est remplacé par 120}	``	``	**120**						
fk ← fact(k) ; (l'appel à fact avec le paramètre k=3)	**5**	**3**	``	/	/		La procédure *fact* est désactivée		
=> La transmission des paramètres → n=3 *f←1* *for i=1 f←f*i → f=1*1 = 1* *for i=2 f←f*i → f=1*2 = 2* *for i=3 f←f*i → f=2*3 = 6 (**Sortie**)*	L'algorithme en attente de la réponse de la procédure *fact*						**3**	/ 1 2 3	/ 1 1 2 **6**
fk← 6; {fact(n) est remplacé par 6}	``	``	**120**	**6**	/				
fnk← fact(n-k) ;(l'appel à fact avec Par. n-k=2)	``	``	``	``	/				
=> La transmission des paramètres → n=2 *f←1* *for i=1 f←f*i → f=1*1 = 1* *for i=2 f←f*i → f=1*2 = 2 (**Sortie**)*	L'algorithme en attente de la réponse de la procédure *fact*						**2**	/ 1 2	/ 1 12 **2**
fnk:= 2; {fact(n-k) est remplacé par 2}			**120**	**6**	**2**		La fonction *fact* est désactivée		
*C← 120 **div** (6 * 2) → C = 120 div 12 = 10* *Écrire(c);*	``	``	``	``	``	**10** **10**			

Vous avez remarqué, dans le déroulement ci-dessus, à chaque appel à la procédure fact, il y a deux flèches en entrée (bleus), et une flèche en sortie (rouge). Lien : https://onlinegdb.com/V7OlPalXv

IV.8. Sous-Programmes récursifs (Récursivité)

IV.8.1. Notion de récursivité

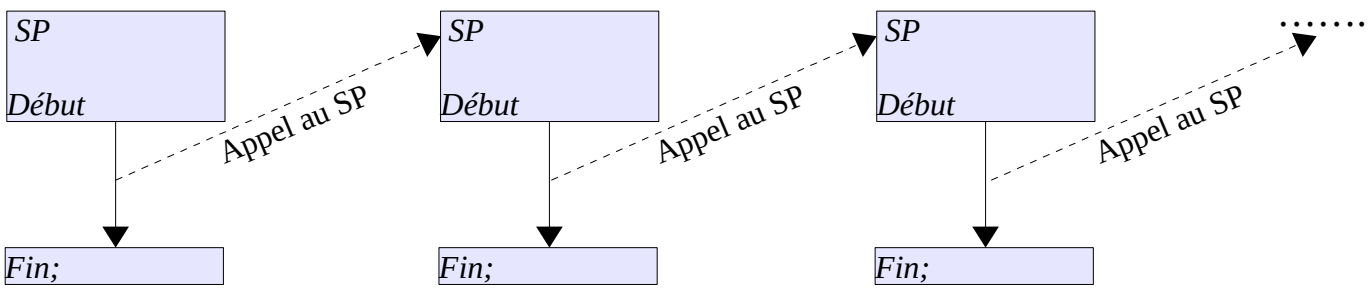
Un sous-programme récursif est un sous-programme qui fait appel à lui même, d'une manière directe ou indirecte. Les sous-programmes récursifs sont généralement utilisés pour résoudre les problèmes mathématiques avec récurrence (comme le factoriel, la puissance, suite de Fibonacci, *etc.*). La récursivité des sous programmes est aussi utilisée pour manipuler quelques structures de données avec une définition récurrente, comme par exemple, les listes chaînées, les piles, les files, les arbres, *etc.*

Une procédure ou une fonction récursive permet de réaliser le même traitement plusieurs fois, donc, et dans quelques situations, on peut les utiliser pour remplacer les boucles (**Pour**, **Tant-quee** et **Répéter**).

Soit un sous-programme récursive SP, donc, à l'intérieure des instructions du SP, on trouve un appel à SP aussi, comme indiqué ci-dessous (on suppose que SP est une procédure) :

```
procédure SP(p1:Type1, p2:Type2)
Début
   <instruction1>;
   <instruction2>;
   ...
   SP(pe1, pe2); // appeler SP dans le code de SP
   ...
   <instructionN>;
Fin;
```

Ceci, peut être schématisé comme suit :

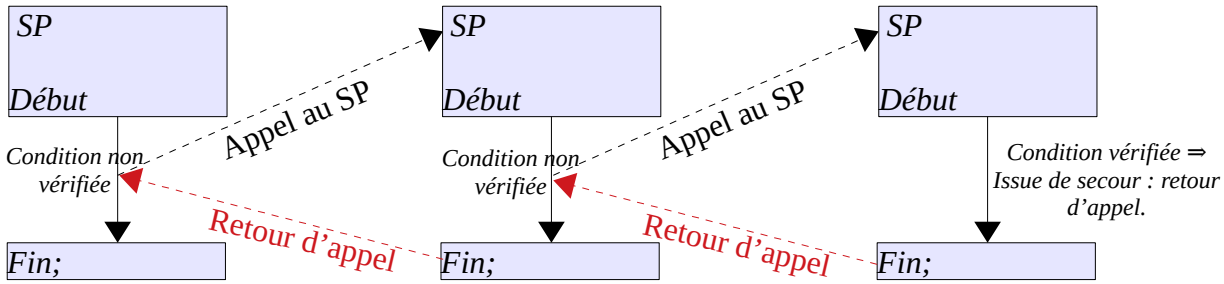


Si une procédure, ou une fonction, contient un appel à elle même, si nous ne faisons pas attention, nous allons avoir des appels interminables (comme une boucle infinie), et il n y aura pas de retour.

IV.8.2. Issue de secours

Pour s'assurer que les appels récursifs aient une fin (une terminaison), on doit indiquer une ou plusieurs conditions (expressions booléennes) pour arrêter l'appel récursif. Ces conditions sont dites issue de secours.

Donc, dans une procédure / fonction récursive, un issue de secours est une condition qui indique la fin des appels et on commence à retourner les résultats aux appelant dans le sens in verse.



IV.8.3. Exemples pratiques

Dans cette section, nous allons voir quelques exemples mathématiques pour réaliser des procédures / fonctions récursives.

IV.8.3.1. Factoriel

Pour calculer le factoriel de *n (nombre entier positifs)*, nous avons la formule suivante :

$$n! = \begin{cases} 1 & si \ \ n=0 \ \ ou \ \ n=1 \\ 1\times 2\times ...\times n & si \ \ n>0 \end{cases}$$

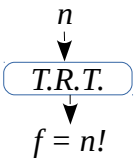
On peut écrire alors :

$$n! = \begin{cases} 1 & si \ \ n=0 \ \ ou \ \ n=1 \\ \underbrace{1\times 2\times ...\times (n-1)}_{(n-1)!}\times n & si \ \ n>0 \end{cases} \Rightarrow n! = \begin{cases} 1 & si \ \ n=0 \ \ ou \ \ n=1 \\ (n-1)!\times n & i \ \ n>0 \end{cases}$$

Donc le factoriel de *n* égale *1* si *n=0* ou *n=1*, ou bien égale au factoriel de *(n-1)* multiplié par *n*. Donc le factoriel de *n* dépend du factoriel de *(n-1)*, et le factoriel de *(n-1)* dépend du factoriel de *(n-*

2) et ainsi de suite (jusqu'à arriver à *n=1* ou bien *n=0*).

$$n! = (n-1)! \times n$$
$$(n-1)! = (n-2)! \times (n-1)$$
$$(n-2)! = (n-3)! \times (n-2)$$
$$\vdots$$
$$(1)! = (0)! = 1$$



D'après les formules précédentes, chaque factoriel d'un nombre dépend du factoriel du nombre précédent. Jusqu'à arriver à 1 ou à 0, où par définition *0! =1* et *1!=1*

Comment réaliser une fonction permettant d'exploiter cette propriété.

Le code suivant montre la première fonction factoriel itérative (utilisant les boucles) :

```
Fonction factoriel (n:entier):entier;
   variables
     i, f:entier;
Début
    f←1;
    Pour i←1 à n faire
         f:= f*i;
    Fin-Pour;
    retourner f;
Fin;
```

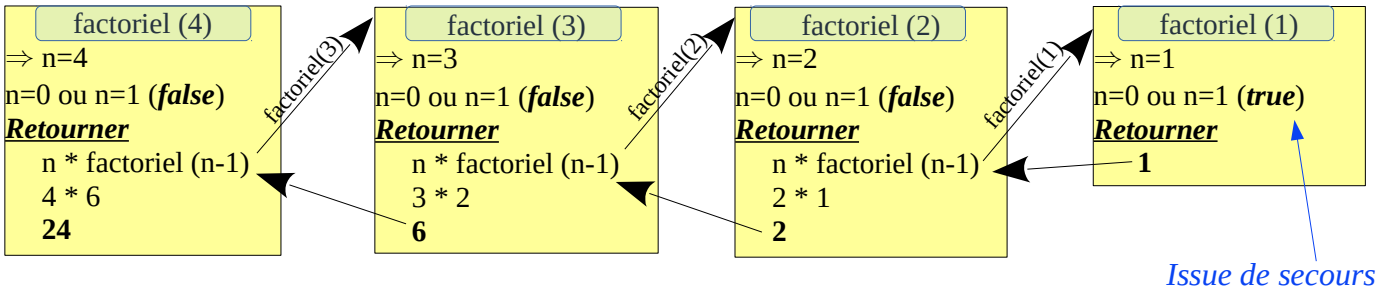
Pour la fonction factoriel récursive, ça sera comme suit :

```
Fonction factoriel (n:entier):entier;
   variables
     f:entier;
Début
    Si ( (n=0) OU (n=1)) alors
       f ← 1;
    Sinon
       f ← n * factoriel(n-1);
    Fin-Si;
    retourner f;
Fin;
```

Ou, d'une façon plus simplifier :

```
Fonction factoriel(n:entier):entier;
Début
    si ( (n=0) ou (n=1)) alors
       retourner 1; //Sortir de la fonction avec 1 (Quitter la fonction)

    retourner ( n * factoriel(n-1) );
Fin;
```

On déroule la fonction récursive *factoriel* (la seconde) **pour** n=4 ⇒ l'appel : *factoriel(4)*



Un exemple de la fonction récursive factoriel, en langage C, est sur le lien suivant : https://onlinegdb.com/T1HddoM0X

Si on veut réaliser la procédure factoriel récursive, on peut le faire comme suit (une possibilité) :

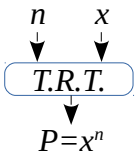
```
Procédure factoriel(n:entier; E/S f:entier);
Début
     Si ( (n=0) ou (n=1)) alors
        f ← 1; //Issue de secour
     Sinon
        factoriel(n-1, f); //Calculer (n-1)! dans f
        f ← n * f; // f reçoit n * (n-1)! = n!
     Fin-si;
Fin;
```

Le code, en langage C, se trouve sur le lien : https://onlinegdb.com/CpOBcj_cn

IV.8.3.2. Puissance

Calcule de la puissance x^n. On a :

$$x^n = \begin{cases} 1 & Si\ n=0 \\ x * x^{(n-1)} & Si\ n>0 \end{cases}$$

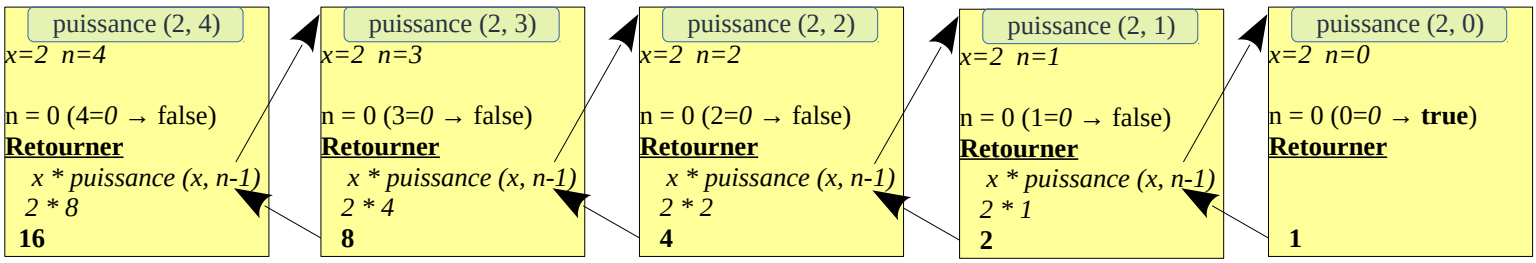



```
Fonction puissance (x:réel; n:entier):réel;
 variables
    p:réel;
Début
     si (n = 0) alors
          p := 1;
     sinon
          p := x * puissance(x, n-1);
      Fin-si;

     retourner p;
Fin;
```

Ce code peut être simplifié comme suit :

```
Fonction puissance (x:réel; n:entier):réel;
Début
     si (n = 0) alors
          retourner 1;

     retourner x * Puissance(x, n-1);
Fin;
```

Si on déroule la fonction puissance pour *n=4* et *x=2*, on aura :



Un programme en langage C, exploitant le code récursif de la fonction puissance ci-dessus, se trouve sur le lien suivant : https://onlinegdb.com/gvVdS31X-

Le code de la procédure récursive puissance, peut être écrit comme suit :

```
Procédure puissance (x:réel; n:entier ; E/S p:réel);
Début
     si (n = 0) alors
          p := 1;
     sinon
          puissance(x, n-1, p);
          p ← p*x;
      Fin-si;
Fin;
```

Le code du programme C avec la procédure *puissance* est sur le lien : https://onlinegdb.com/aOC3TgD-s