Analyse et programmation langage ADA

Ingénieurs1ère année

Utilisation en mode diaporama
La page suivante donne le sommaire.
Pour accéder à un chapitre cliquer sur le lien correspondant
De n'importe quel transparent, on revient au sommaire en appuyant sur le logo EISTI

Sommaire

Sous programmes

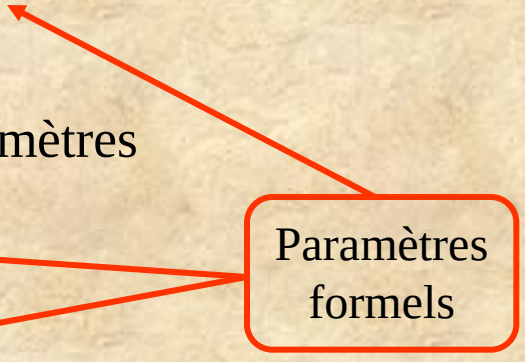
- Exemple de sous programme
- Déclaration de paramètres
- Traduction des paramètres formels
- Déclaration de sous programme
- Corps (body) d'un sous programme
- Appel de sous programme
- Arguments d'appel
- Portée des paramètres (Global, Local)
- Les effets de bords
- Surcharge des opérateurs

Exemple de sous programme

- Si une fonction ou une procédure a plusieurs paramètres formels, ils sont séparés par un "**;**"

- **procedure** Ma_Procedure(X : **in out** Integer **;** Y : **in out** Float) ;

- On peut mettre en facteur le type des paramètres
  procedure Permute(X, Y : **in out** Integer) ;



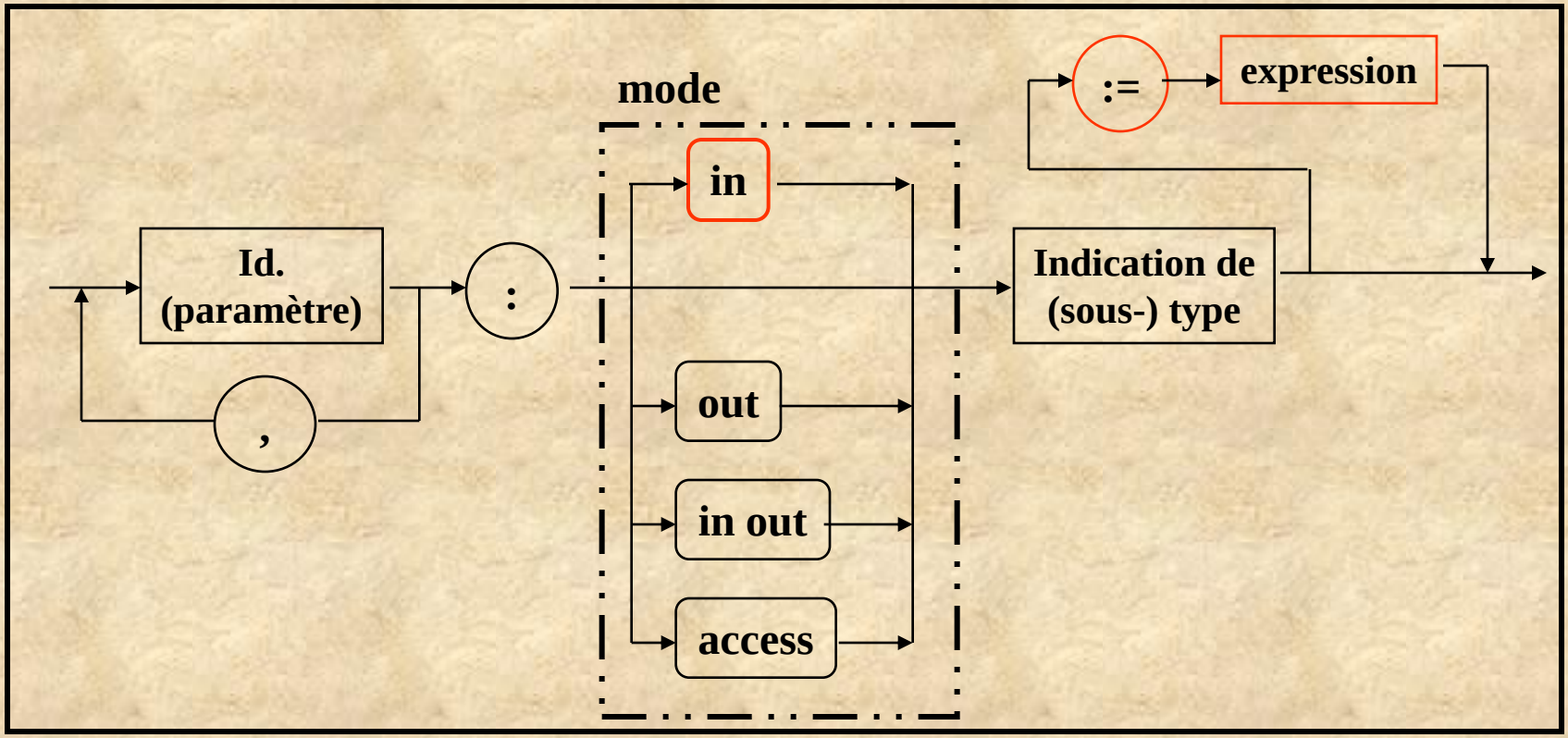
Exemple :

```
procedure Permute(X,Y : in out Integer) is
    Tmp : Integer;              -- garde temporairement X
begin  - - algorithme de Permute
    Tmp := X;
    X := Y;
    Y := Tmp;
end Permute;
```

Déclaration de paramètres dans sous-programme

- Une déclaration de sous-programme définit son nom et la manière dont on l'appelle, c'est à dire dont on fait exécuter les ordres qui sont dans son corps.

- La manière dont on appelle un sous-programme dépend essentiellement de la déclaration de ses paramètres éventuels.



Traduction des paramètres formels

Notation algorithmique	ADA
Consulté X : typeparam	X : **in** typeparam
Elaboré Y : typeparam	Y : **out** typeparam
Modifié Z : typeparam	Z : **in out** typeparam

- Le paramètre est consulté : on met **In** devant son type
- Le paramètre est élaboré : on met **Out** devant son type
- Le paramètre est modifié : on met **In Out** devant son type

Remarque : le langage Ada est très proche de l'algorithmique

Traduction des paramètres formels

- Pour chaque paramètre formel :
  - nom
  - mode
    - ✓ **in** : lecture seule
    - ✓ **in out** : lecture / mise à jour
    - ✓ **out** : écriture ou mise à jour seule
  - type

- Par défaut le mode est **in**
- Un paramètre de mode **in** est considéré comme une constante
- Les fonctions n'autorisent que le mode **in**.

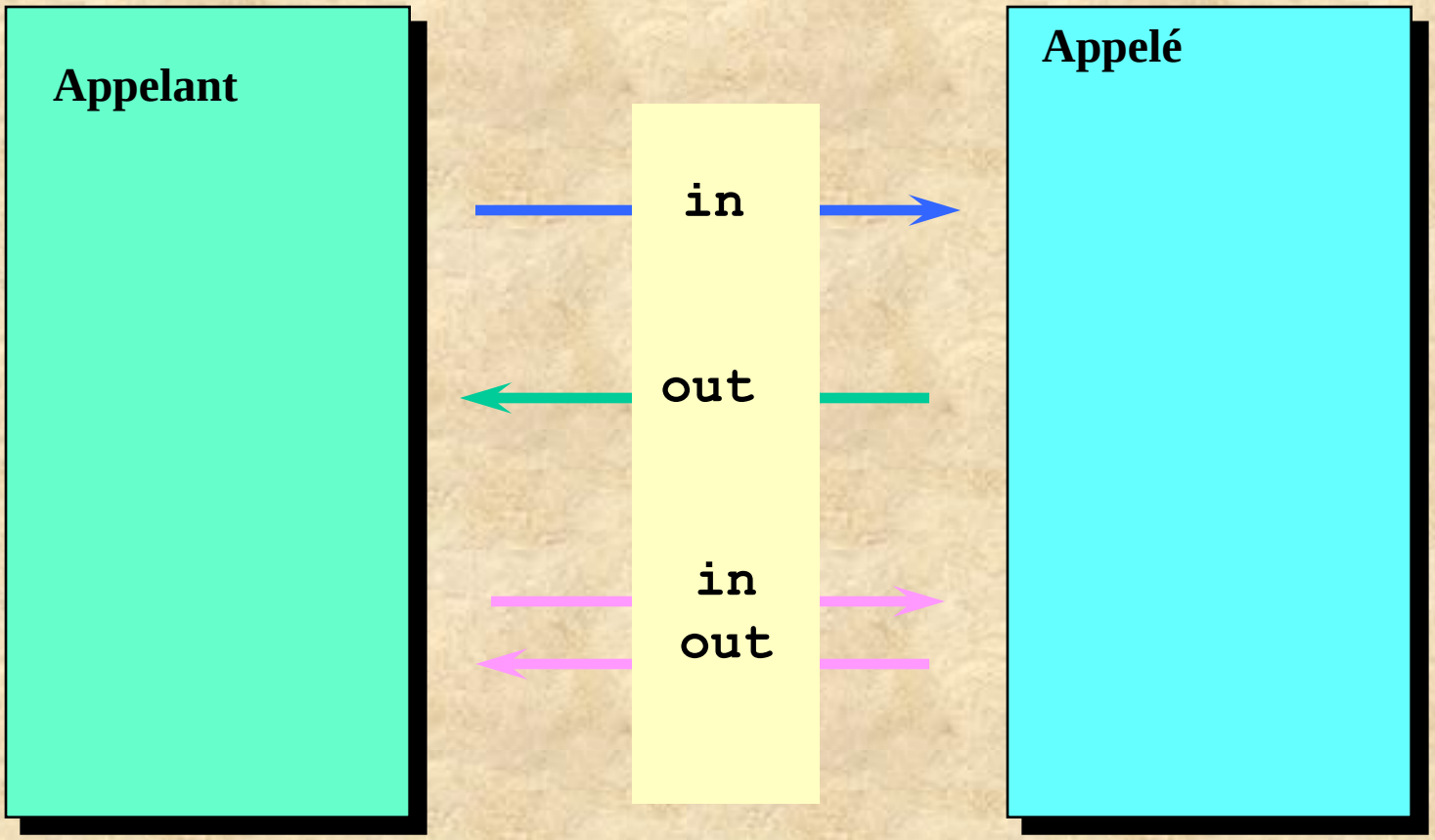
Exemple : on peut déclarer

```
X : in Float := 1.2;
Y : in  Float := 1.5;
Z : out  Float   -- ( valeur de retour)
```

Convention :

On commence par les parmètres de mode **in**, puis **in out** et enfin **out**

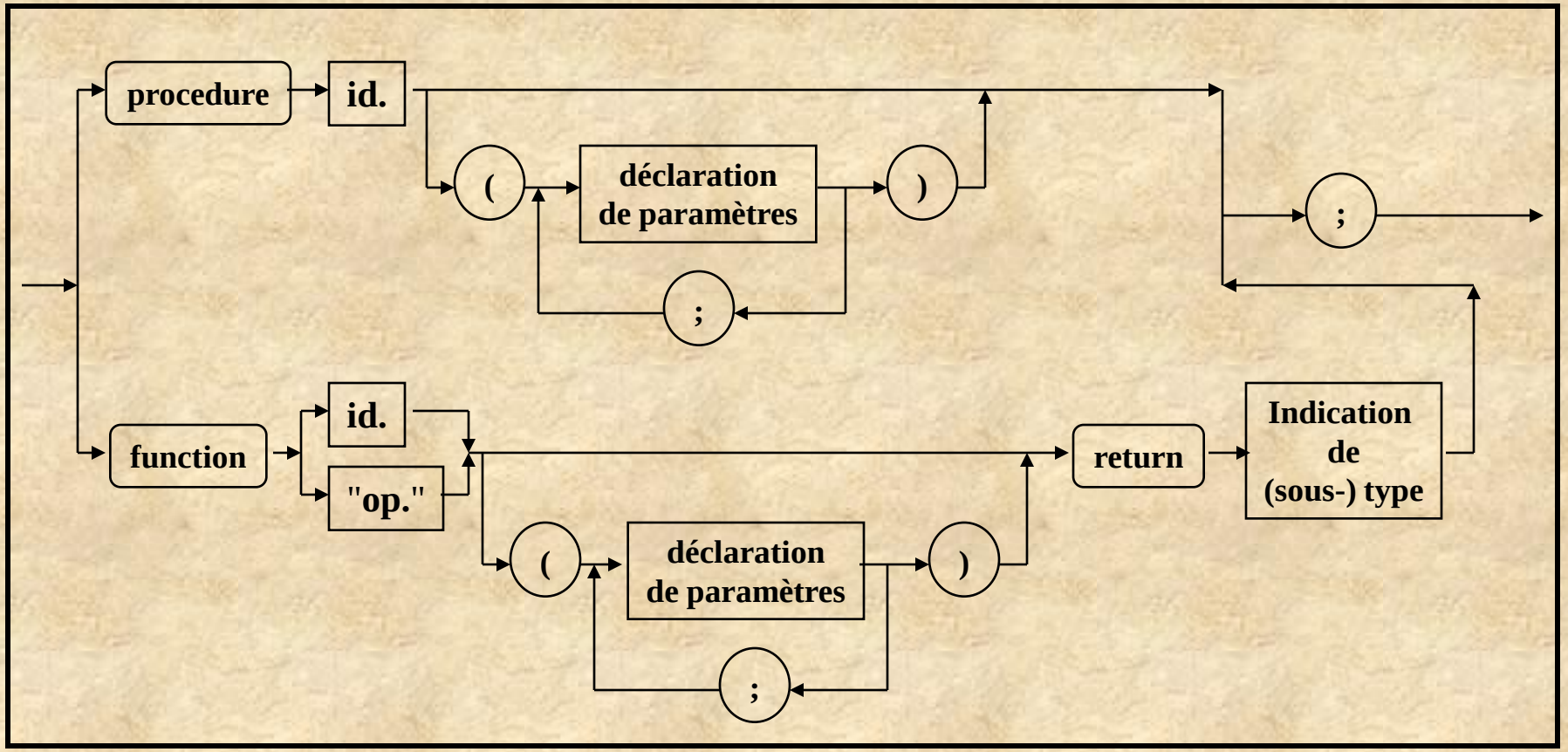
Appel de sous programme



Signature ≡ contrat entre l'appelant et l'appelé

Déclaration de sous programme

- La déclaration de ***sous programme*** est définie par le diagramme suivant :



Exemple de specifications

- Soit le type : **type** TDecimal **is range** 1 .. 10 ;

- Une *fonction* est un sous programme qui, lorqu'il est appelé, exécute ses ordres et retourne une valeur se substituant à l'appel. Le (sous-) type du *résultat* est donc indiqué après le mot **RETURN**.

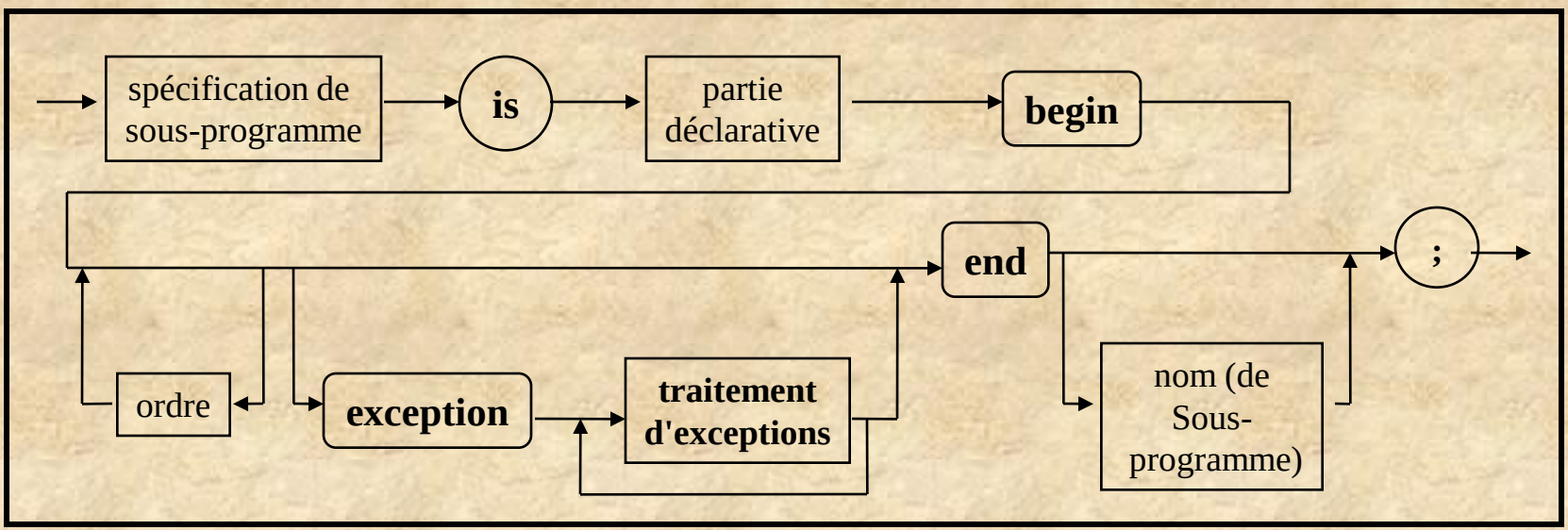
function Somme(X, Y : **in** TDecimal) **return** TDecimal ;

- Une *procedure* est un sous programme qui, lorqu'il est appelé, exécute ses ordres en communiquant des valeurs par ses paramètres. Elle est donc déclarée par les seules indications de son nom et de ses paramètres.

procedure Somme(X, Y : **in** TDecimal ; Z : **out** TDecimal);

Corps (body) de sous programme

- Le *corps* d'un sous-programme est défini par le diagramme suivant.



- La spécification de *sous-programme* présente au début du corps de sous-programme doit être identique à celle de la déclaration

- Cette déclaration peut être omise, sauf si le sous-programme est utilisé dans un progiciel, sa déclaration devant être dans la partie visible (et son corps dans la partie cachée), ou bien si le corps du sous programme est placé après celui d'une unité qui l'utilise.

Fonctions

- Déclaration

```
function Somme(X, Y : in TDecimal) return TDecimal ;
```

- Corps

```
function Somme(X, Y : in TDecimal) return TDecimal is
        Z : TDecimal ; -- Partie déclarative
begin
        Z:= X+Y ;   --  Partie instructions
        return Z ;   -- On peut utiliser return X+Y
end Somme;
```

Une fonction s'achève avec une instruction **return**.

Une fonction peut contenir plusieurs instructions **return**.

Procédures

- Déclaration

```
procedure Somme(X, Y : in TDecimal ; Z : out TDecimal);
```

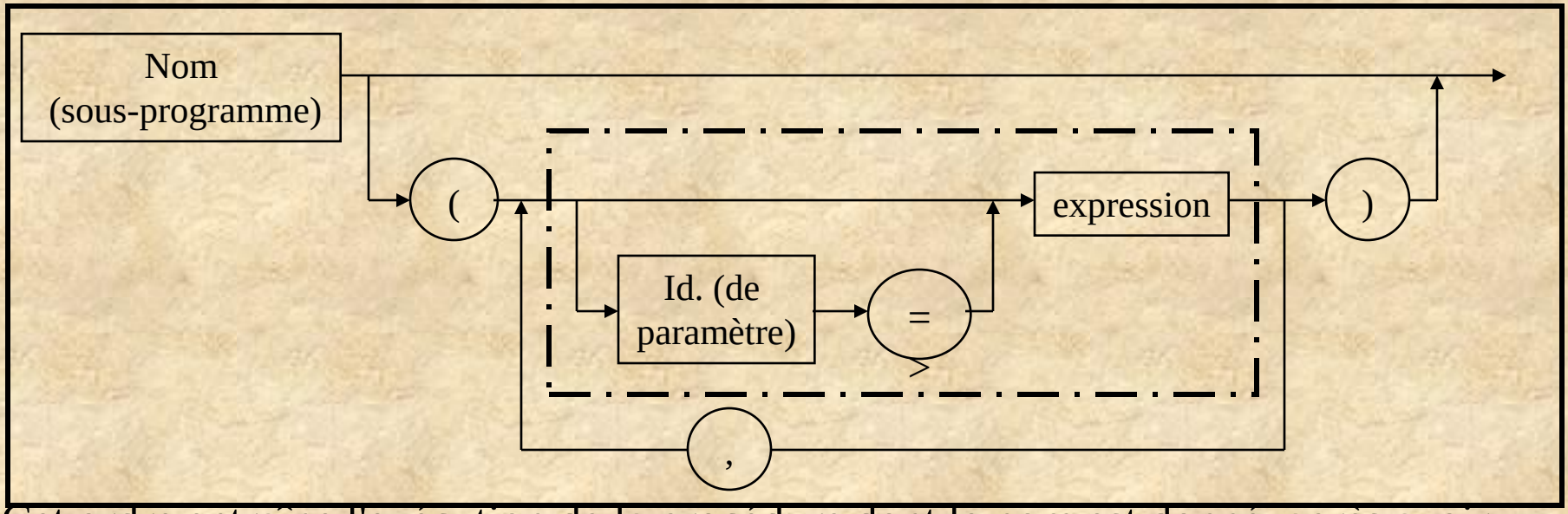
- Corps

```
procedure Somme(X, Y : in TDecimal ; Z : out TDecimal) is
        -- Partie déclarative ici aucun besoin
begin
        Z := X+Y; -- Partie instructions
end Somme;
```

Une procédure s'achève avec le **end** final.

Appel de sous programme

- L'appel de *procédure ou fonction*, est un *ordre* simple défini par le diagramme suivant



- Cet ordre entraîne l'exécution de la procédure dont le nom est donné, après avoir donner à chaque paramètre IN ou IN OUT une valeur, dites argument d'*appel*, sauf éventuellement si le paramètre correspondant à une valeur par défaut.

```
Get(A);                    New_Line(Spacing =>3);          Skip_Line;
Z := Sin(X+Y);             X := Random(Generator =>Generateur);
```

Résultat d'appel de sous programme

Remarque : La *procedure* **Somme** communique son résultat par le paramètre Z, alors que la *fonction* **Somme** retourne elle-même le résultat, et est donc directement utilisable dans une expression.

- Fonctions
  - appelées au sein d'expression
  - retourne une valeur
- Procédures
  - appelées comme des instructions,
  - modifie une valeur
- Déclaration et Corps
  - déclaration : convention d'appel (optionnelle)
  - corps : actions à effectuer (au moins une instruction "**null**")

Arguments d'appel

- La liste des arguments d'appel peut être :

-*positionnelle*, c'est à dire que les valeurs des arguments sont données respectivement et dans l'ordre aux paramètres (tel qu'ils sont déclarés dans la spécification de procédure) ;

-*nommée*, les valeurs étant données aux paramètres dont le nom les précède, séparé par " =>" L'ordre est alors quelconque ;

- *mixte*, la partie positionnelle précédant la partie nommée.

Exemple : la procédure "Agenda" imprime le calendrier d'un mois, de jour à jour, à partir d'une date donnée est déclarée ainsi :

procedure Agenda (An, Mois : **in** natural; Jour : **in** natural := 1) ;

```
Agenda(1999, 3, 8) ;                                  -- positionnelle
Agenda(Jour => 8, Mois => 3, An => 1999) ;            -- nommée
Agenda(1999, Jour => 8, Mois => 3) ;                  -- mixte
Agenda(Mois => 3, An => 1999) ;  ou Agenda(1999, 3) ; -- la valeur du Jour = 1
```

Concordance de type

En principe le type du paramètre effectif doit être identique à celui du paramètre formel. Toutefois, dans la mesure du raisonnable, on peut demander une conversion dans un sens au moment de l'appel et dans le sens inverse au retour.

- Imaginons l'en-tête de procédure:

  procedure Cube (I : **in out** Integer) **is**

  ...

- dans le module qui appelle Cube on a une variable J déclarée INTEGER on peut donc appeler simplement :

  Cube (J);

- Mais si l'on a aussi une variable X de type FLOAT, on peut également appeler:

  Cube (Integer(X));

- il y aura alors conversion FLOAT <==> INTEGER à l'entrée du sous-programme et conversion inverse au retour.

Exemple

```
procedure Trier is
        A, B, C : Integer;

        procedure Trier (X,Y : in out Integer);
        procedure Saisir(X : in out Integer);

        procedure Trier (X,Y : in out Integer) is
            Temp : Integer;                    -- garde temporairement X
         begin  - - Trier
            if ( X > Y) then
                Temp := X;
                X := Y;
                Y := Temp;
            end if ;
         end Trier;

        procedure Saisir(X : in out Integer)  is
        begin
                 Put(" Entrer La valeur : ");
                Get(X);
                Skip_Line;
        end;

begin -- Trier
        Saisir(A);    Saisir(B);     Saisir(C);
        Trier(A, B); Trier(B, C); Trier(A, B);
end Trier ;
```

On peut remplacez ce code par une procédure `Permuter`

IV.6.2 Portée et visibilité dans des blocs imbriqués

Portée

L’objet est potentiellement visible (l'objet existe, on parle aussi de durée de vie).

Visibilité.

On peut utiliser son identificateur pour y référer. Même si l'objet à une portée dans une région, il n'est pas obligatoirement visible. Il peut être masqué par un autre identificateur défini dans cette région

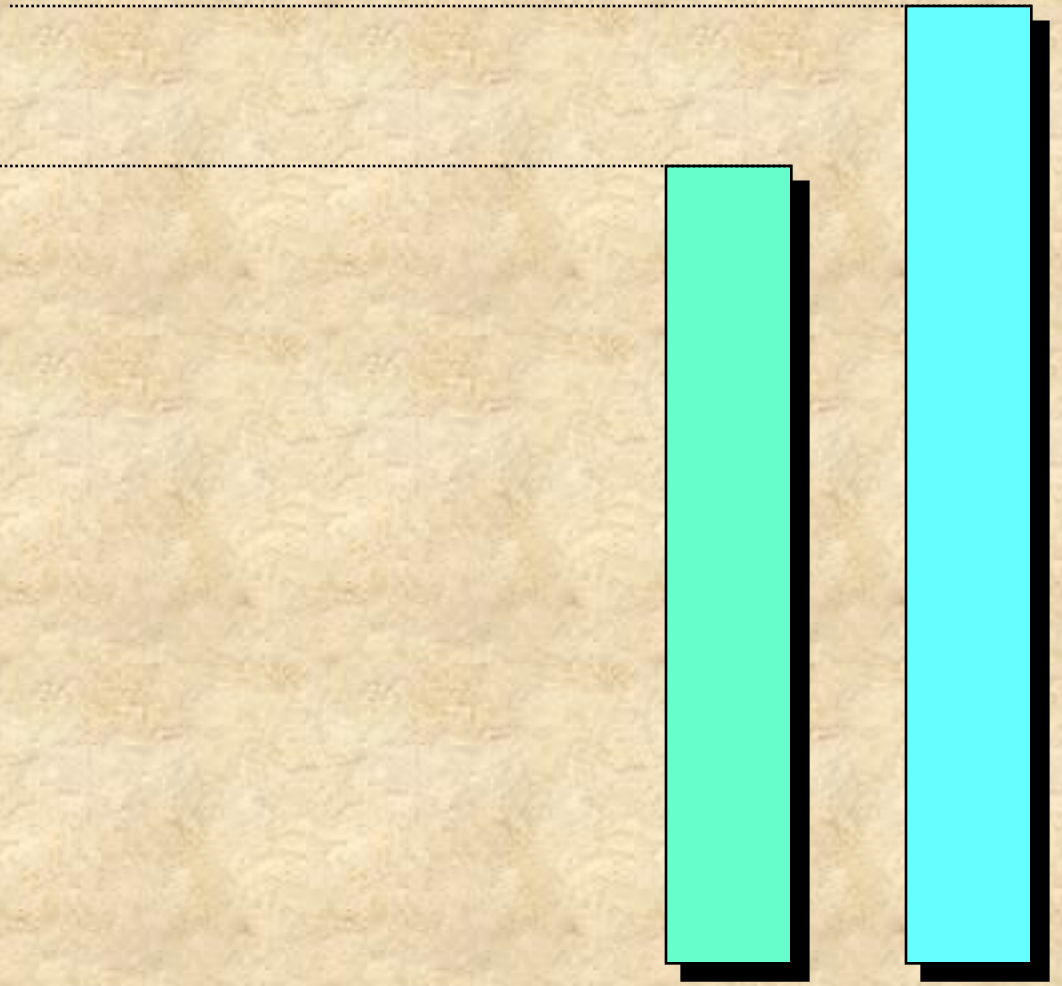
Portée et visibilité des paramètres (Global, Local)

```
procedure Niveau_1 is
        N       : Integer ;                          -- 3 identificateurs dans la region 1
        Max     : constant :=10 ;
        Lettre : Character ;

    procedure Niveau_2  (Caractere : in Character) is
            N : Integer  ;                           -- 2 identificateurs dans la region 2
     begin                                  *  A l'intérieur du Niveau_2, le N local cache
           ……..                                le N global (défini dans le programme principal)
     end  Niveau_2 ;                        * On peut aussi utiliser la variable local comme suite
                                              Niveau_1.Niveau_2.N

    procedure Aussi_Niveau_2   is
            N : Float ;                              -- 1 seul identificateur dan la region 2'
        procedure Niveau_3 (Caractere : in Character) is
                N : Integer ;                        -- 3 identificateurs dans la region 3
               Ok : Boolean ;                  On peut utiliser la variable N du Niveau_1
        begin                                  on la préfixant du nom de l'unité, avec
              …….                              une notation pointée, soit : Niveau_1.N
        end Niveau_3 ;
   begin
           ……..
   end Aussi_Niveau_2 ;

begin  --Niveau_1
     ……..
end Niveau_1 ;
```

Portées et visibilité

Si une procédure (ou fonction) en appelle une autre qui elle-même appelle à son tour la première, nous avons un petit problème au niveau des déclarations => **Problème de la récursivité croisée**

```
procedure F (...) ;

procedure G (...) is
begin -- G
        F (...);
end G;

procedure F (...) is
begin -- F
        G (...);
end F;
```



Les effets de bords

```
procedure Exemple is
    A : Integer := 1;
    function F1 return Integer is
    begin
        return A + 3;
    end F1;

    function F2 return Integer is
    begin
        A := A * 10 ;
        return A + 3;
    end F2;

begin
    Put ( F1 + F2) ;  -- 4 + 13 = 17
    A : Integer := 1;
    Put ( F2 + F1) ;  -- 13 + 13 = 26
end Exemple;
```

Il ne faut **jamais** utiliser de variables globales, mais les transmettre en paramètres.

Remarque : F1 + F2 <> F2 + F1

Les effets de bords

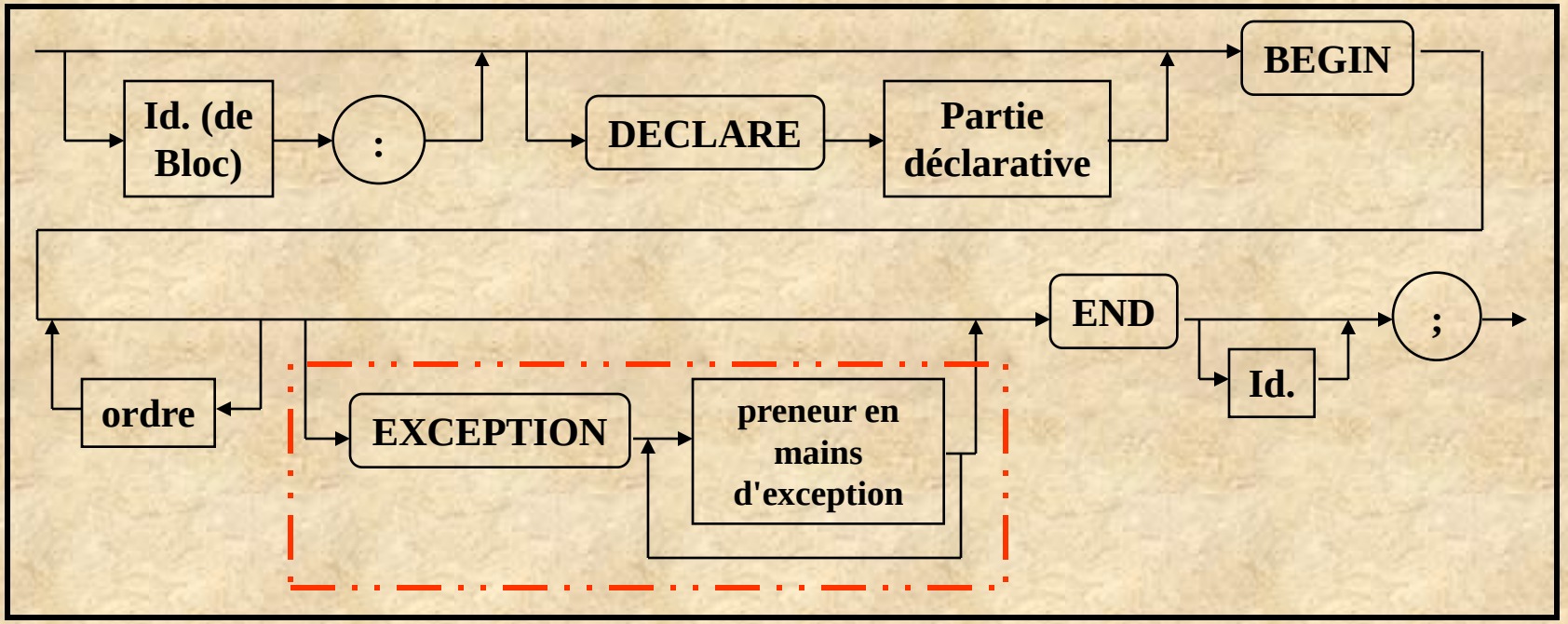
- Il en va de même si les fonctions avaient des paramètres de sortie:

```
procedure Exemple is
        A : Integer := 1;
       function F1 (A : in Integer ) return Integer is
       begin
              return A + 3;
       end F1;

       function F2 (A : "in out" Integer ) return Integer is
       begin
             A := A * 10 ;
             return A + 3;
       end F2;

begin
          Put ( F1 (A) + F2 (A) ) ;  -- 4 + 13 = 17
          A : Integer := 1;
          Put ( F2 (A) + F1 (A) ) ;  -- 13 + 13 = 26
end Exemple;
```

Remarque : F1 + F2 <> F2 + F1 de même F2 + F2 <> 2 * F2

Bloc

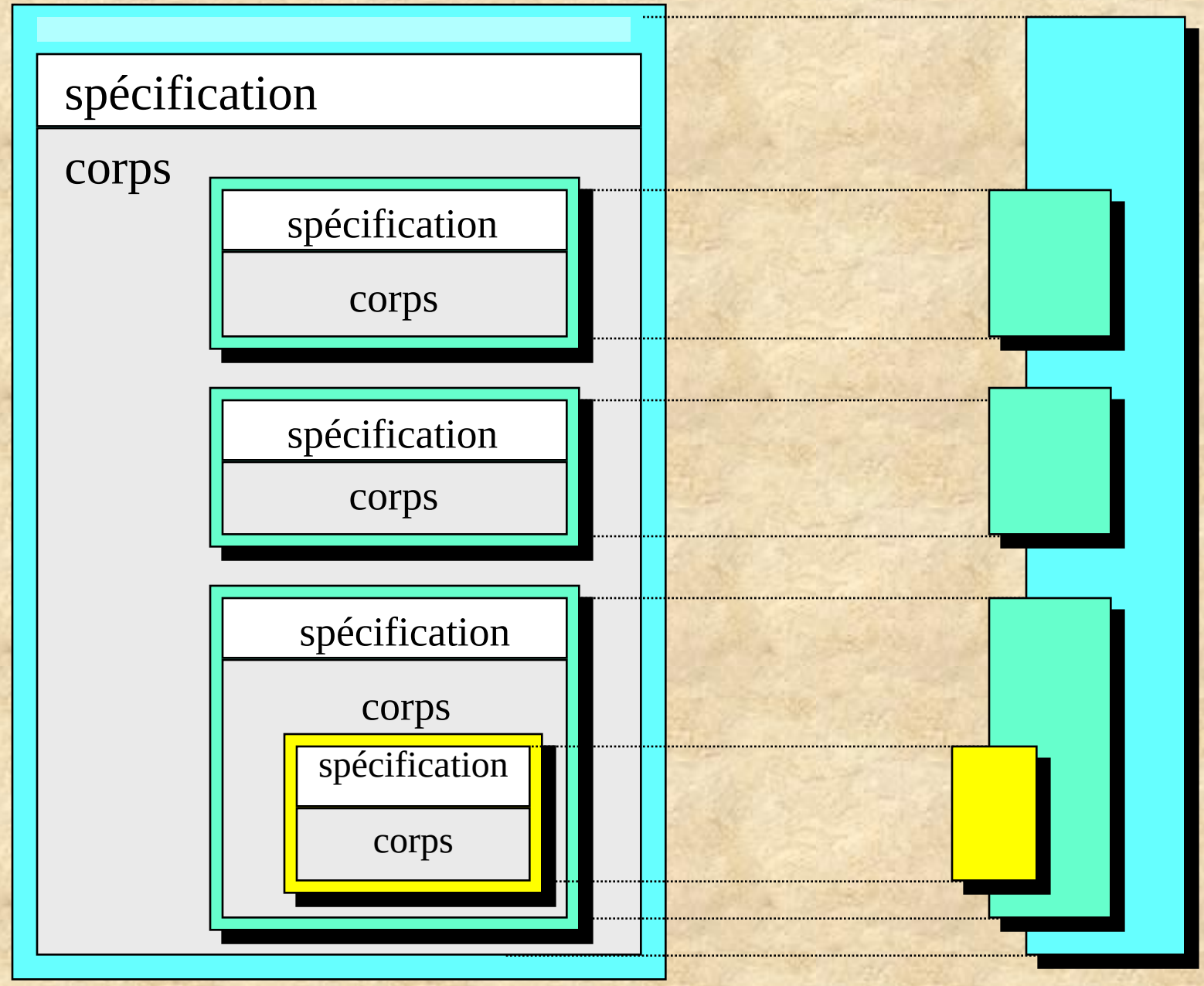
Un *bloc* est un ordre composé, défini par le diagramme syntaxique suivant :



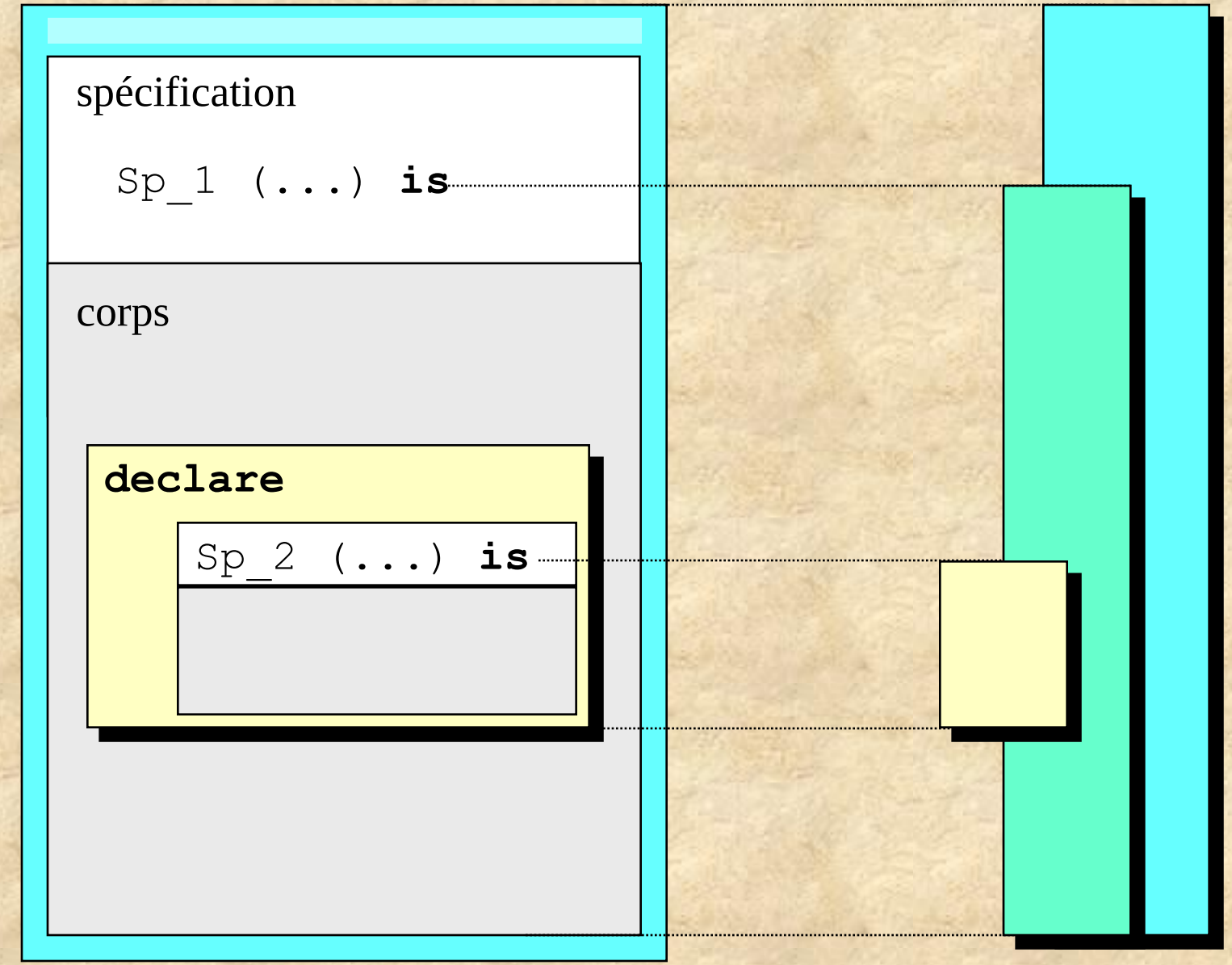
Exemple :

```
ECHANGE :
                declare
                        Val_1 : Integer ;    --Val_1 a une existence uniquement
                 begin                        -- à l'intérieur du bloc
                        Val_1 := 10 ;        - - ordre d'affectation
                        B       := Val_1;
end ECHANGE ;
```

Portée et visibilité dans des blocs

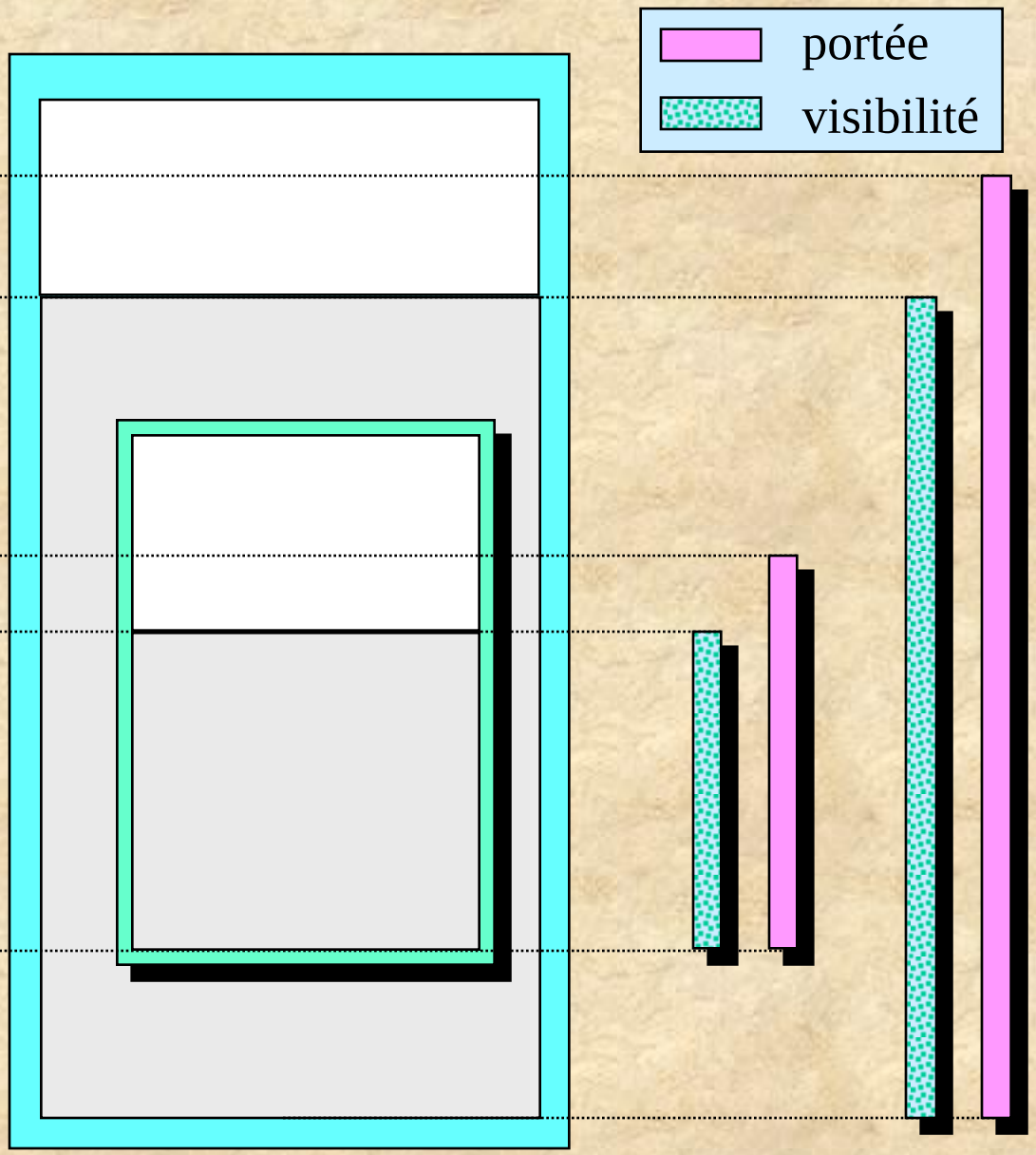


Portées et visibilité



Portée et visibilité

```
declare
   I:Integer:=5;

begin
    I:=I+1;
    declare
      K:Integer:=I;
      I:Integer:=0;

    begin
       K:=I;
       ...

    end


end
```



Surcharge de sous programmes ou d'opérateurs (polymorphisme statique)

- Le nom d'une fonction est soit un identificateur ordinaire, soit un opérateur prédéfini auquel on attribue un nouveau sens, et que l'on appelle surchargé. Il doit alors figurer en double apostrophe comme une chaîne de caractères littérale.

- Peuvent être surchargés les opérateurs
  - unaires : abs,+, -, not (logique)
  - binaires : **, *, /, +, -, mod, rem, &
    relation (>, <, <=, >=, =, /=),
    logique(and, or, xor)

Exemple : **type** V **is array** (1..10) **of** Float ; (voir cours tableau)
function "+"(X, Y : **in** V) **return** V ;

Le corps de la fonction définira l'addition de 2 vecteurs, et l'on pourra alors additionner deux objets A et B de type V par A + B.

Remarque :
Un opérateur surchargé ne peut avoir de paramètre avec valeur initiale par défaut.

Surcharge d'opérateurs (polymorphisme statique)

Définition de 2 sous-programmes ayant le même nom mais des signatures différentes.

function "+" (Left, Right : Integer) **return** Integer;
function "-" (Left, Right : Integer) **return** Integer;

function "+" (Left, Right : Float) **return** Float;
function "-" (Left, Right : Float) **return** Float;

function "+" (Left, Right : T_Complexe) **return** T_Complexe;
function "-" (Left, Right : T_Complexe) **return** T_Complexe;

Identification : à partir des paramètres d'appels

Appel de sous programme

- L'exécution des ordres du sous programme commence après appel de celui-ci et affectation des valeurs d'appel aux paramètres IN et IN OUT ; lorsqu'elle se termine les paramètres IN OUT et OUT reçoivent les valeurs de retour, et l'unité appelant poursuit le traitement

```
procedure Transformation is
   type T_Vect is array (1..3) of Float ;
   type T_Mat is array (1..3;1..3) of Float ;
   Mrot, Mtrans, Mhom : T_Mat := ((1.0,0.0,0.0),
                                  (0.0,1.0,0.0),
                                  (0.0,0.0,1.0));
   V, Vr : T_Vect := (others => 0.0);
begin
   Vr := "*" (Mrot, V);
   AfficherVect(Vr);

   Vr := Mhom * V;
   AfficherVect(Vr);

end Transformation ;
```

Paramètres formels

Partie déclarative

```
function "*"  (M  : in T_Mat ;
               Ve : in T_Vect) return T_Vect ;
```

```
function "*" (M  : in T_Mat ;
              Ve : in T_Vect) return T_Vect is
Vs :  T_Vect := (others => 0.0);
begin
      for i in M'range(1)  loop
         for j in M'range(2)  loop
            Vs(i) := Vs(i)+M(i,j)*Ve(j) ;
         end loop;
      end loop;
      return Vs;
end "*" ;
```

Paramètres effectifs

Partie non visible