

Analyse et programmation langage ADA



Informatique 1^{ère} année

Attachement

- L'attachement n'est pas formellement défini. Intuitivement, on associe (ou attache) un attribut à un objet. Les attributs peuvent être des noms, types, valeurs.
- L'attachement peut se faire à différents moments. On considère normalement trois périodes:
 - À la compilation (en fait, à la traduction, parce que l'attachement se fait dans les compilateurs et les interpréteurs);
 - Au chargement (Lorsque le code objet est préparé pour l'exécution, quand le code nécessaire aux instructions prédéfinies du langage ou aux bibliothèques utilisées est obtenu);
 - À l'exécution (Lors de l'exécution du programme).

Attachement Statique et Dynamique

- On distingue aussi deux types d'attachement selon sa durée:
 - Attachement statique est permanent, pour la durée du programme;
 - Attachement dynamique effectif durant certaines parties de l'exécution seulement.

Assigner des propriétés aux variables

- variable → nom
 - À la compilation
 - ☞ Décrit dans les déclarations
- variable → adresse
 - Au chargement et à l'exécution (e.g. C),
 - à l'exécution (e.g. Smalltalk)
 - ☞ Habituellement fait implicitement
- variable → type
 - À la compilation (e.g. Java),
 - à l'exécution (e.g. Scheme)
 - ☞ décrit dans la déclaration

Assigner des propriétés aux variables suite

- variable \rightarrow valeur
 - à l'exécution,
 - Au chargement (initialisation)
 - ☞ Spécifié dans les instructions, surtout les assignations
- variable \rightarrow durée
 - à la compilation
 - ☞ Décrit dans la déclaration
- variable \rightarrow porté
 - à la compilation
 - ☞ Exprimé par l'emplacement des déclarations

Durée de vie

- L'attribution de la mémoire pour un objet se fait au chargement. Deux classes de variables peuvent être distinguées:
 - Variables statiques
 - Variables dynamiques

Variables statiques

- L'attribution se fait une seule fois, avant le début du programme.
- Fortran IV est un langage important où c'est la seule façon d'attribuer la mémoire aux objets.
- Ceci est désuet, trop rigide, mais conceptuellement simple et peu coûteux. Ceci rend la récursivité impossible.

Variables dynamiques

- L'attribution se fait après le début du programme, lors de son exécution.
- Deux possibilités d'attribution dynamique:
 - **Attribution et libération** Explicite, c'est à dire que le programmeur doit le faire. C'est ce qui est fait quand on utilise des pointeurs.
 - ☞ Par exemple, en ADA on attribue avec `new(p)`, et libère avec `dispose(p)`.
 - ☞ En C on utilise `malloc()` et `free()`.
 - Attribution implicite (quand on entre dans un bloc) et libération implicite (quand on en sort).

Blocs

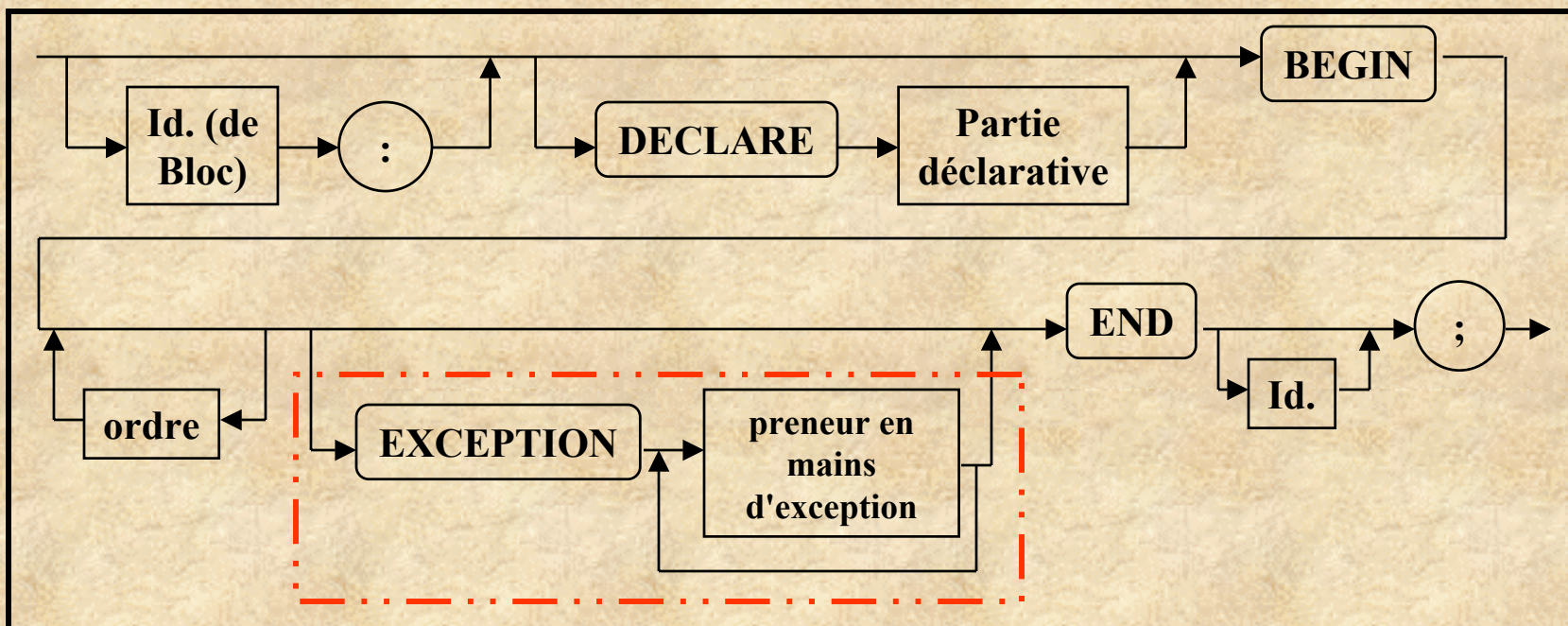
- On regroupe des déclarations et des instructions pour:
 - regrouper les étapes d'une même activité (e.g., les étapes d'un algorithme de tri),
 - assurer une interprétation correcte des noms.
- Les Noms sont attachés à divers éléments d'un programme. On fait référence à ces noms dans des instructions.
- La portée d'un nom N signifie l'ensemble des endroits dans un programme où N dénote le même objet.

Blocs suite

- Les blocs peuvent être imbriqués. Les noms introduits dans un bloc sont appelés attachements locaux. Un nom qui est utilisé, sans avoir été déclaré dans un même bloc, doit avoir été déclaré dans un bloc englobant ce bloc .
- L'imbrication des blocs est possible en Pascal, Ada et (sous certaines formes) en C. Elle n'est pas permise en Fortran.
- Un programme, une procédure ou une fonction consiste en une entête et un bloc nommé. Par rapport aux blocs anonymes disponibles en Algol 60, Algol 68, PL/I, Ada et C – mais pas en Pascal.

Diagramme syntaxique d'un 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 ;

Blocs anonymes

- Un bloc anonyme est comme une procédure définie (sans nom) et appelé immédiatement et une seule fois.
- De tels blocs sont utiles quand un calcul est nécessaire une seule fois, et des variables auxiliaires sont nécessaires pour ce calcul. On ne veut pas déclarer ces variables à l'extérieur du bloc (une procédure aurait pu être utilisée pour parvenir au même résultat)



Portée et visibilité des variables 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 dans des blocs imbriqués

```

declare
  I: Integer := 5;

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

  begin
    K := I;
    ...
  end
end
end
  
```

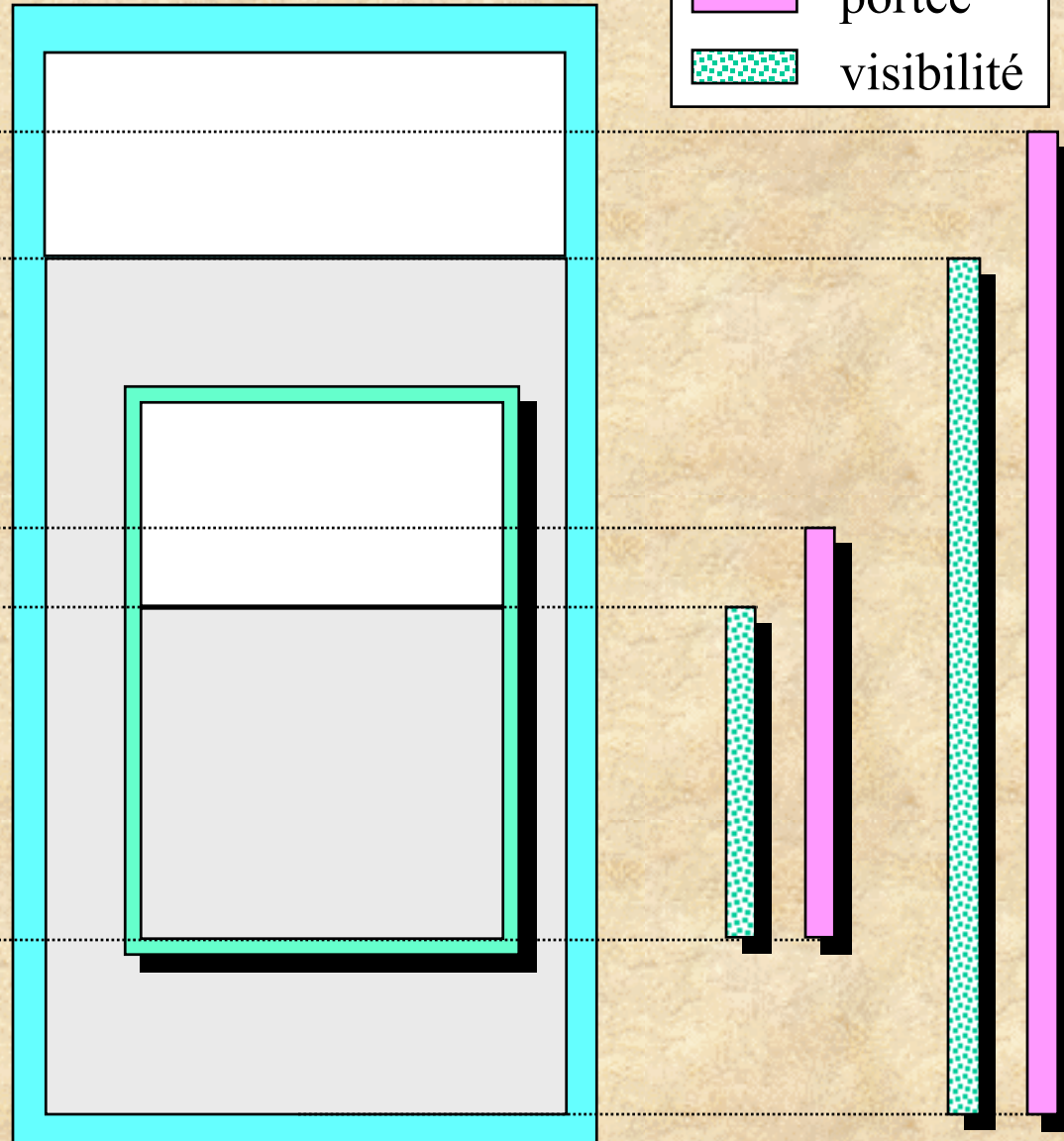
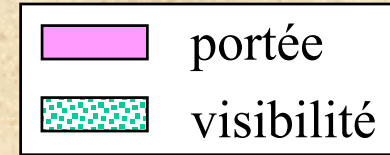
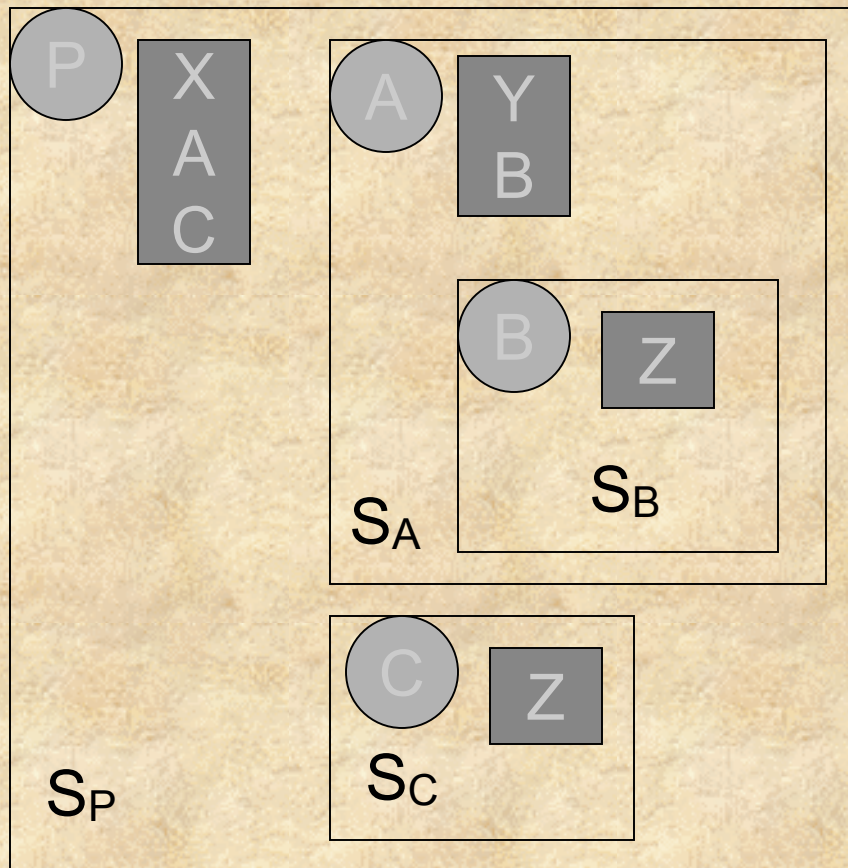


Diagramme d'imbrication de procédures



```

procedure P() is
  X : Integer;
  procedure A() is
    Y : Character;
    procedure B() is
      Z : Float;
      begin --B

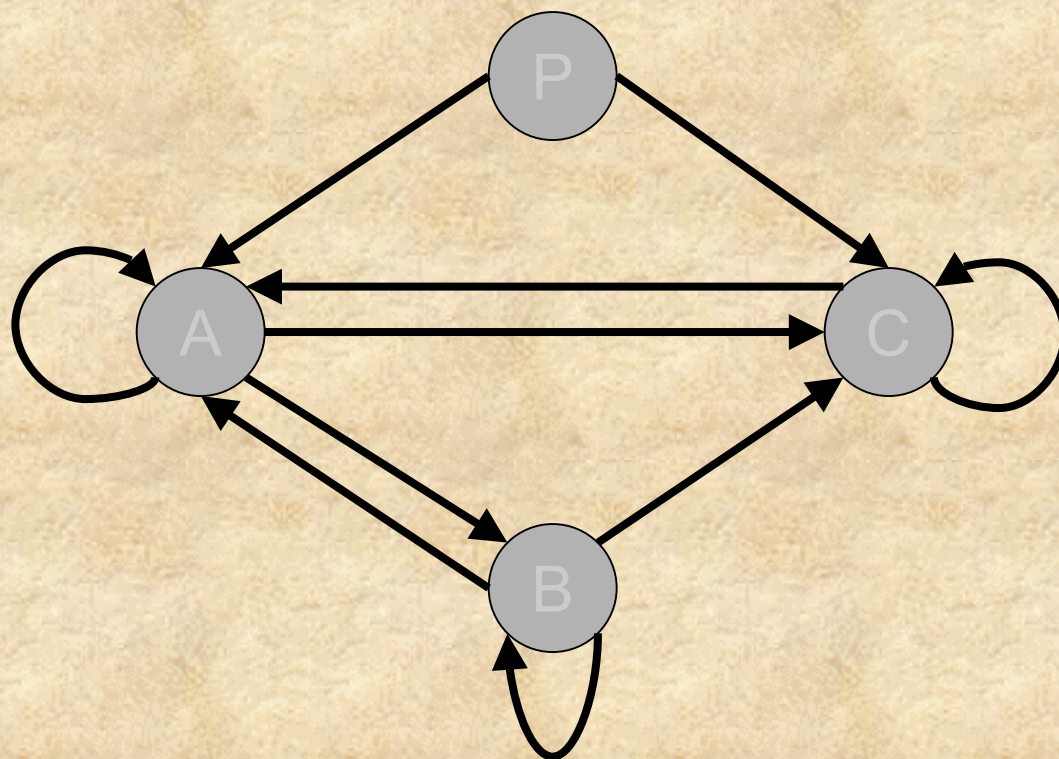
        end B;
      begin --A

        end A;
    procedure C() is
      Z : Integer;
      begin --C

        end C;
    begin --P

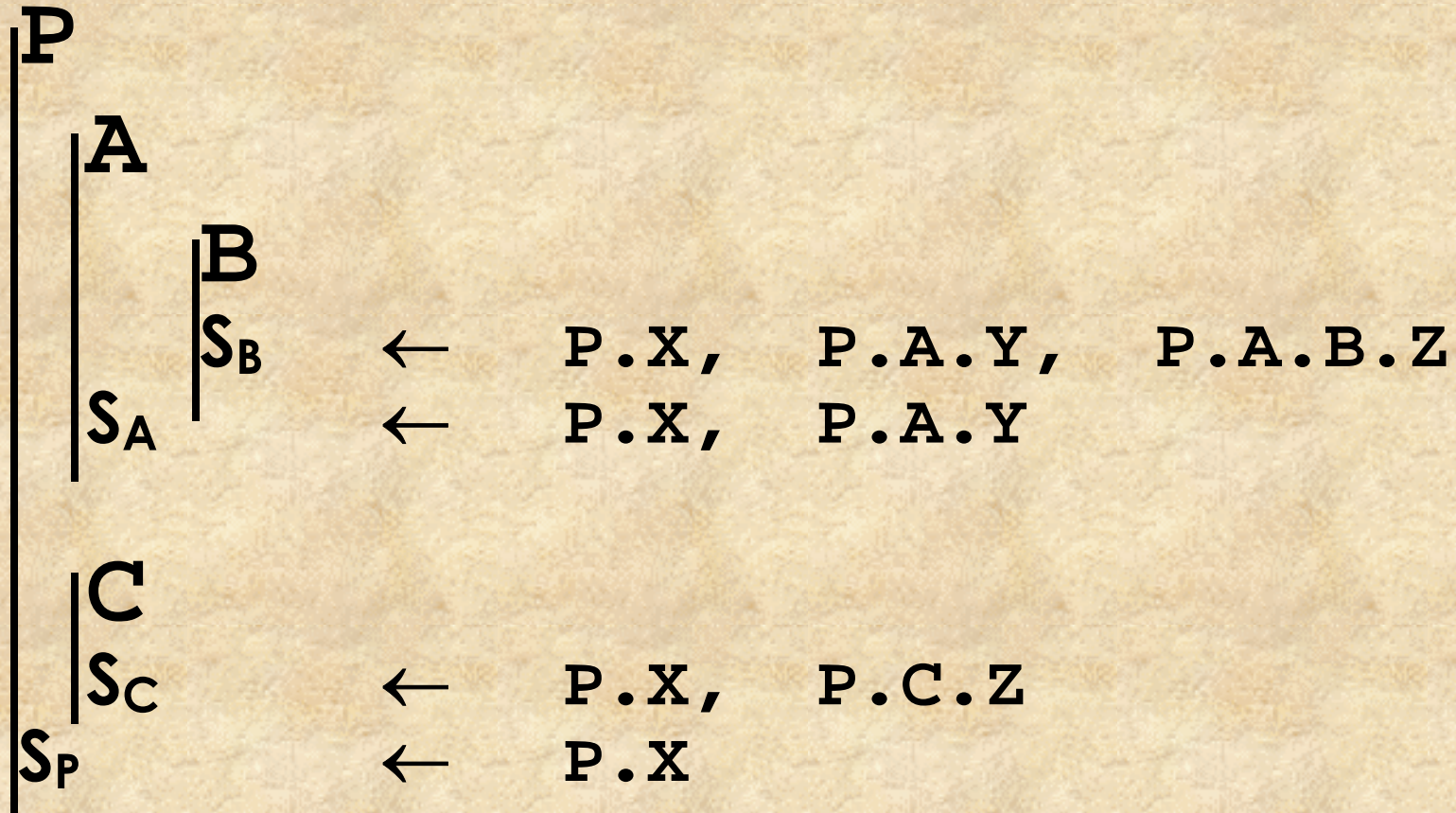
    end P;
  
```

Graphes d'appels



Montre quelles unités du programme peuvent appeler quelles autres unités. Les flèches pointant vers l'unité d'où elles proviennent montrent la possibilité de récursivité

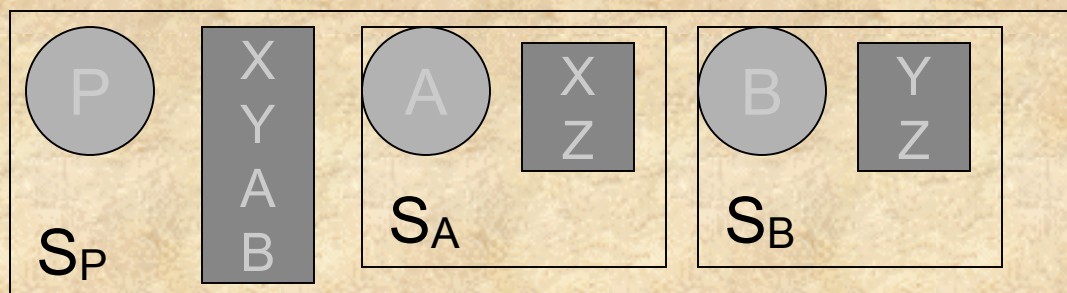
Visibilité, l'environnement de référence



P.X signifie "variable X déclarée dans P".

Visibilité

- Trous dans la portée (variables cachées): le même nom est utilisé dans un bloc et dans un bloc imbriqué à l'intérieur de celui-ci.



- Les variables visibles dans S_P : $P.X$, $P.Y$
- Les variables visibles dans S_A : $P.A.X$, $P.A.Z$, $P.Y$
- Les variables visibles dans S_B : $P.B.Y$, $P.B.Z$, $P.X$
- Donc, $P.X$ est visible partout dans P sauf dans S_A , où $P.A.X$ la cache en devenant elle même visible.
- Et, $P.Y$ est visible partout dans P sauf dans S_B , où $P.B.Y$ la cache en devenant elle même visible.
- Il n'y a pas de trous dans la portée de $P.A.Z$ ou $P.B.Z$ car elles ont des régions de visibilité disjointes.

Problème de la récursivité croisée

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 (...);
```

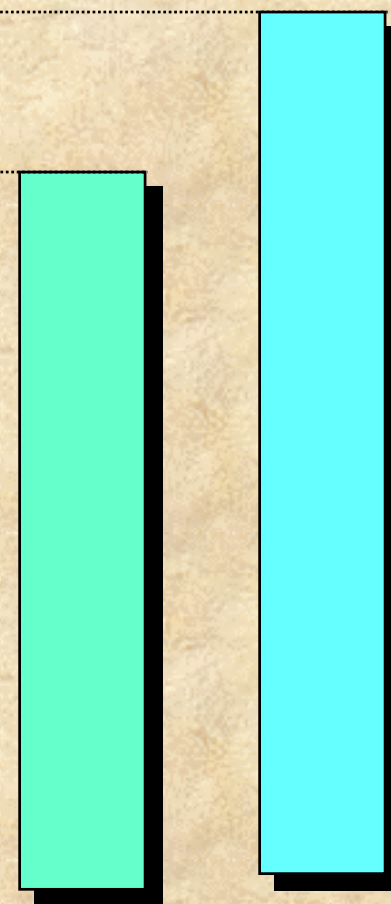
```
end G;
```

```
procedure F (...) is
```

```
begin -- F
```

```
    G (...);
```

```
end F;
```



Porté et visibilité dans les procédures

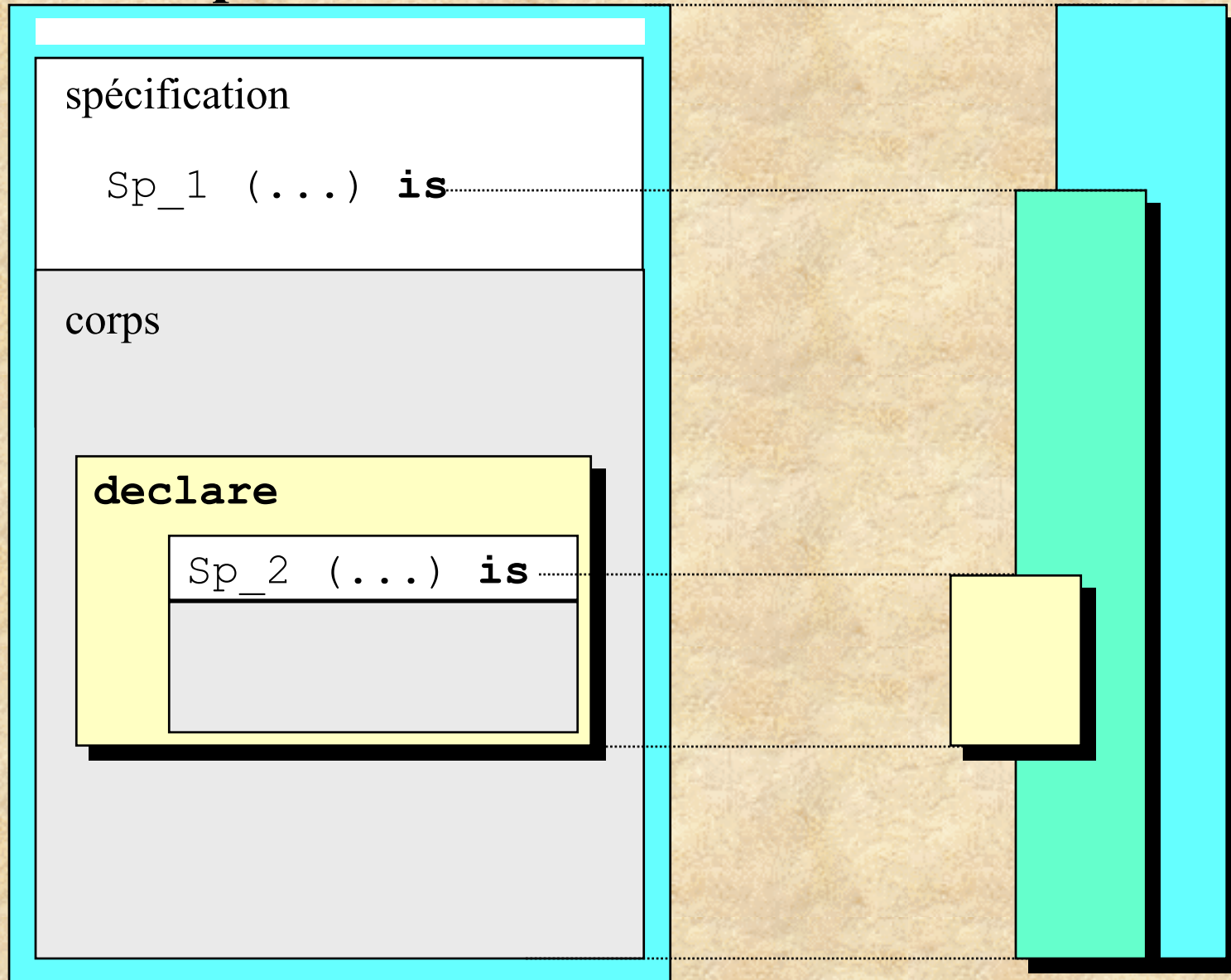
```

procedure P() is
  X : Integer;
  procedure A()is
  begin
    X := X + 1;
    put(X);
  end A;
  procedure B()is
    X : Integer;
  begin
    X := 17;
    A();
  end B;
begin
  X := 23;
  B();
end P;

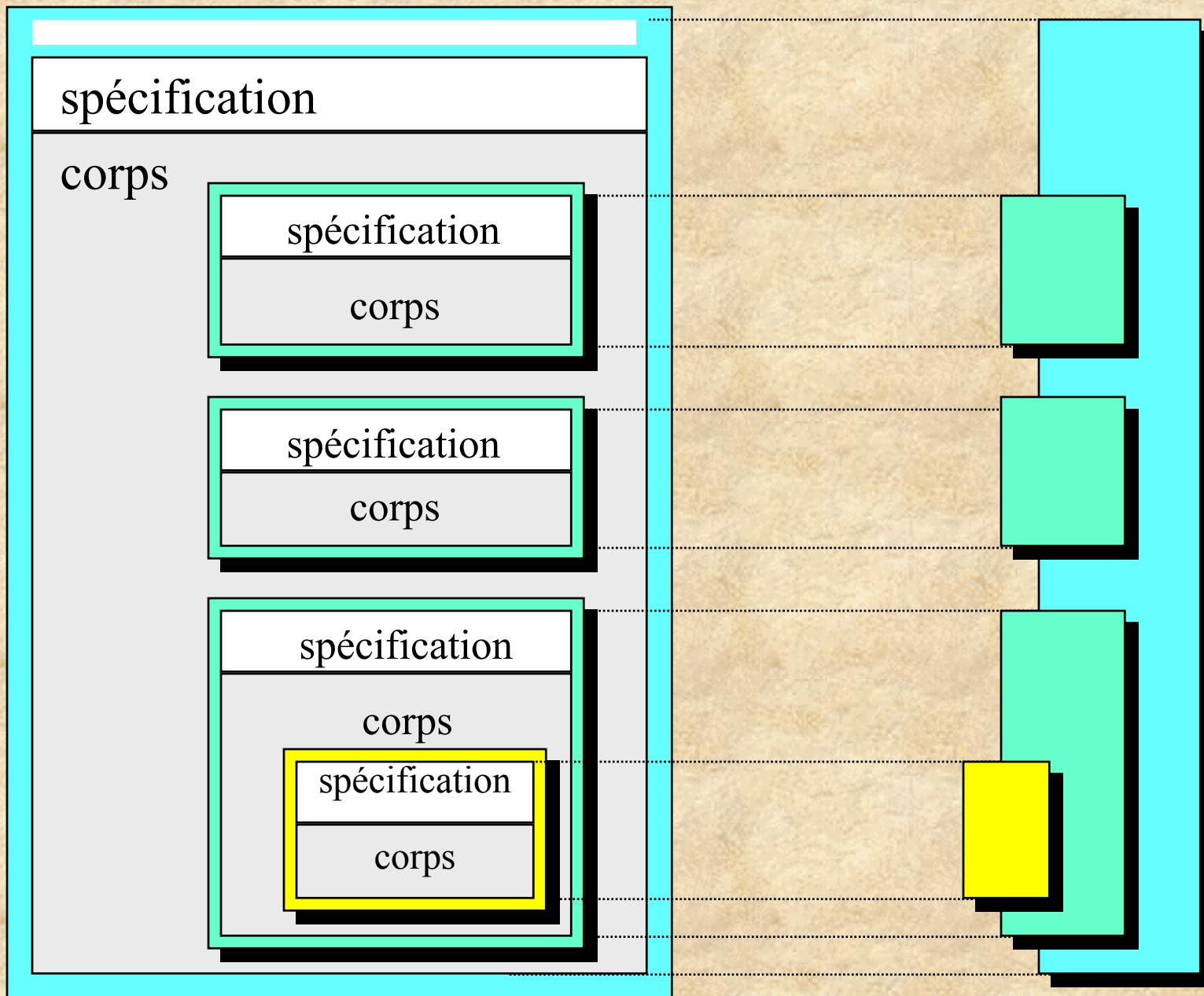
```

- Le programme appelle B, puis B appelle A. Quand A utilise X, Lequel des X est-ce?
- Avec la portée statique, c'est le X de P, le bloc dans lequel A est imbriqué. Le résultat est 24.
- Avec la portée dynamique (où la recherche est faite dans la chaîne d'appels), c'est le X de B, le bloc le plus récemment appelé contenant une déclaration de X. Le résultat est 18.

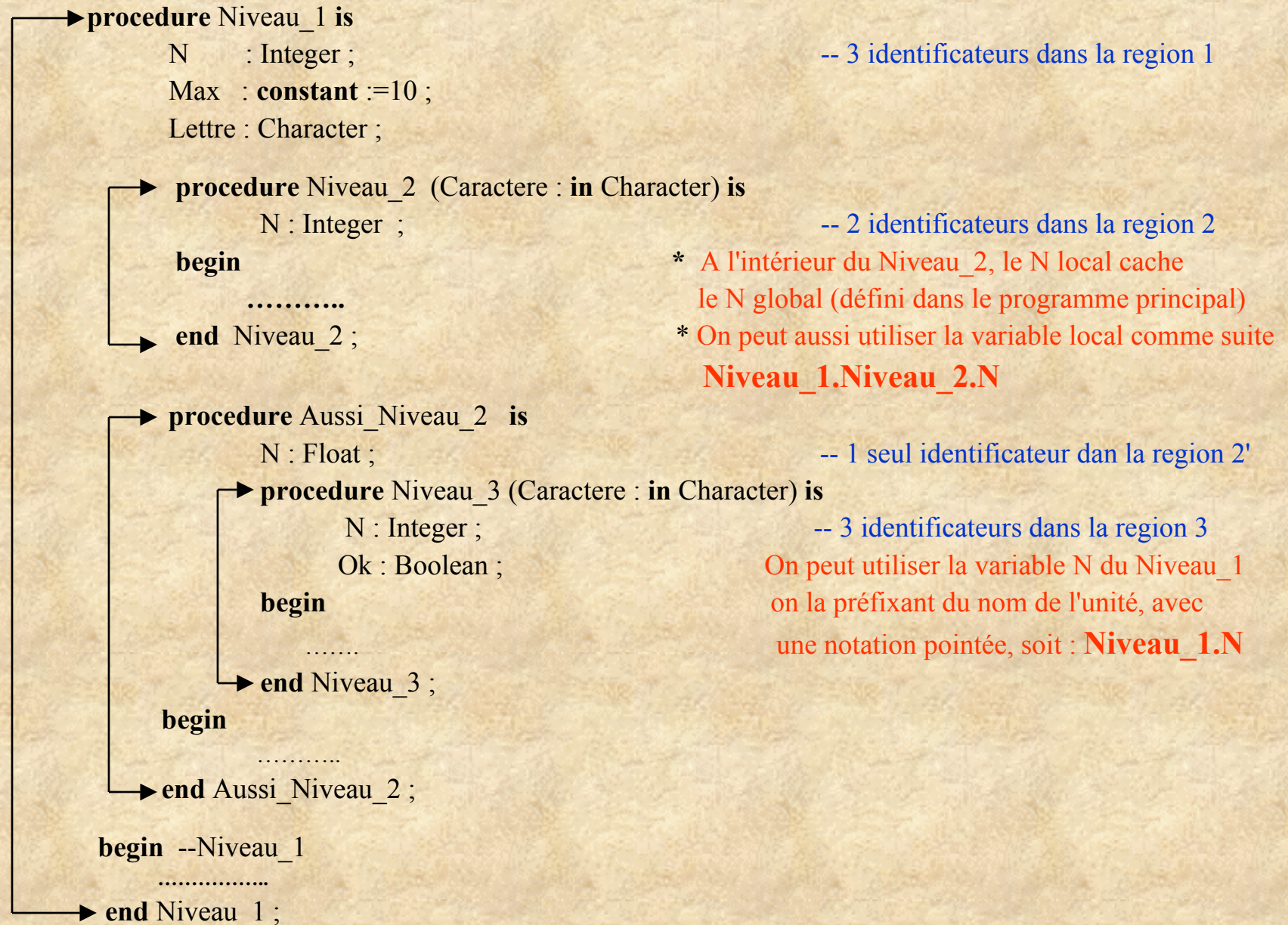
Portées et visibilité dans une procédure avec déclaration de bloc



Généralisation sur portée et visibilité



Portée et visibilité des paramètres (Global, Local)



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 := 1;
  Put ( F2 + F1) ; -- 13 + 13 = 26
end Exemple;
```

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

Remarque : $F1 + F2 \neq F2 + F1$

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 := 1;

 Put (F2 (A) + F1 (A)) ; -- 13 + 13 = 26

end Exemple;

Remarque : F1 + F2 \diamond F2 + F1 de même F2 + F2 \diamond 2 * F2