Applications Concurrentes : Conception, Outils de Validation

ACCOV_B

ANNEXE 1 :

On présente les éléments du langage Ada. On utilise Ada pour sa clarté, pour sa généralité et sa portabilité (il est normalisé ISO et le compilateur Gnat est un logiciel libre disponible et copiable au CNAM). C'est aussi un langage très sûr et très efficace, utilisé pour des applications devant être certifiées

ADA : PROGRAMMATION À PETIT GRAIN

ADA : PROGRAMMATION À GROS GRAIN

LA PROGRAMMATION OBJET EN ADA

PROGRAMMATION À PETIT GRAIN ("PROGRAMMING IN THE SMALL")

SOUS-PROGRAMMES (PROCÉDURES ET FONCTIONS)

```
declare
   function Minimum(X, Y: in Integer) return Integer is
   begin
      if X > Y then return Y; else return X; end if;
   end Minimum;
   procedure Quadrat(A, B, C: in Float; R: out Float;
         S: in out Float; OK: out Boolean) is
   Z: Float; --variable locale à la procédure
   begin
      Z := A * A + B * B + C * C; OK := Z /= 0;
      if OK then S := S + Z; R := SQRT(Z); end if;
   end Quadrat;
   T, Y: Integer; L1, L2, L3, M, N: Float; Correct: Boolean;
begin
   T := Minimum(7, Y); Quadrat(L1, L2, L3, M, N, Correct);
end;
```

MODE DES PARAMETRES FORMELS

in : **le paramètre formel est une constante et il n'est possible que de lire la valeur du paramètre effectif associé à l'appel. Passage par copie de valeur au moment de l'appel.**

out : **le paramètre formel est une variable et il n'est possible que de mettre à jour la valeur du paramètre effectif associé à l'appel. Passage par copie de résultat au retour du sous-programme.**

in out : **le paramètre formel est une variable et il est possible de lire et de mettre à jour la valeur du paramètre effectif associé à l'appel. Passage par copie de valeur au moment de l'appel et copie de résultat au retour du sous-programme.**

passage par defaut : in

pour une fonction, tous les paramètres obligatoirement en in.

PROGRAMMATION À PETIT GRAIN ("PROGRAMMING IN THE SMALL")

STRUCTURE DE BLOC

```
declare
   <déclarations locales au bloc>
begin
   <suite d'instructions du bloc>
exception
   <récupérateurs d'exceptions>
end;

-- déclarations obligatoires car langage fortement typé
```

TYPES PRÉDÉFINIS:

Boolean, Character, String, Integer, Natural, Duration, Float

TYPES CONSTRUITS

```
type Couleur is (Rouge, Jaune, Bleu, Orange, Vert, Violet);
type Acrylique is new Couleur; -- type dérivé  = nouveau type
subtype CouleurFine is Couleur range Rouge..Orange; -- sous-type
type Aquarelle is new CouleurFine;
type Primaire is (Bleu, Rouge, Jaune);
X: Couleur := Jaune;  -- déclaration avec initialisation
Y: CouleurFine; Z: Primaire;
U: Acrylique := Rouge; V: Aquarelle := Orange;
begin
   Y :=  X;
   Z := X ; --illégal car types différents
   Z :=  Primaire'Succ(Bleu);  --Z vaudra Rouge
   X := Couleur'Succ(Bleu);  --X vaudra Orange
   U := X; --illégal car types différents
   V := Y; --illégal car types différents
end;
```

TABLEAUX, CHAINES DE CARACTERES

```
Table: array (1.. Taille_Table) of Acrylique;
S:  String(1..7);
```

PROGRAMMATION À PETIT GRAIN ("PROGRAMMING IN THE SMALL")

ARTICLES

```
type Date_Jour is range 1..31;
type Date_Mois is range 1..12;
type Date_An is range 1_900..2_050; -- écriture pour 1 900
type Date is
record
   Jour: Date_Jour := 1;
   Mois: Date_Mois :=1;
   Annee:Date_An;
end record;
D1, D2, D3, D4: Date;
begin
   D1.Annee := 1991; D2 := (3, 2, 1953);
   D3 := (Annee => 1974, Jour =>4, Mois =>7);
   D4.Jour := D3.Jour;
   D4.Mois := D4.Jour; --illégal car types différents
end;
```

TYPE POINTEUR

```
type Cellule;
type  Lien is access Cellule;
type Cellule is
   record
      Valeur: Element;
      Suivante: Lien;
   end record;
Val: Element; Ajout: Lien;
begin
   Ajout := new Cellule;
   Ajout.Valeur := Val;
   Ajout.Suivante := null;
end;
```

PROGRAMMATION À PETIT GRAIN ("PROGRAMMING IN THE SMALL")

INSTRUCTIONS

<séquence> ;

<affectation> :=

<conditionnelle>

```
-- Nombre : Integer range 0..10_000;
if Nombre < 10 then Long := 1;
else if Nombre < 100 then Long := 2;
        else     if Nombre < 1000 then Long := 3;
                 else Long := 4;
                 end if;
        end if;
end if;

--ou mieux
if Nombre < 10 then Long := 1;
elsif Nombre < 100 then Long := 2;
elsif Nombre < 1000 then Long := 3;
else Long := 4;
end if;
```

EXPRESSIONS BOOLÉENNES

Dans le cours, on utilise beaucoup les expressions booléennes et il est parfois nécessaire d'imposer l'ordre d'évaluation
Par exemple, pour éviter l'erreur suivante :

```
if I/J > K then Action1; else Action2; end if; -- erreur si J = 0
```

on doit programmer

```
if J > 0 then if I/J > K then Action1; else Action2; end if; end if;
```

On dispose de la clause and then

```
if J > 0 and then if I/J > K then Action1; else Action2; end if;
```

-- and then **garantit que l'évaluation est faite de gauche à droite**

On a de même la clause or else

CHOIX

```
--Requete: Char;

case Requete is
   when 'A'|'a'    => Action1;   -- la donnée requete vaut a ou A
   when 't'        => Put('t'); Action2;
   when 'e'        => Put("requête e");  Action3;
   when 'x'..'z'   => Action1;
   when others     => null;  -- toutes les autres valeurs du caractère
end case;
```

PROGRAMMATION À PETIT GRAIN ("PROGRAMMING IN THE SMALL")

BOUCLES

```
        -- boucle DO classique
for I in 0..9 loop A(I) := I; Put(A(I)); end loop;

        -- recherche en table de la première valeur T(K) = Vrai
for K in 1..N loop
   exit when T(K);
end loop;

            -- boucle FOR  qui calcule Y = fact(X)
Y := X; while X > 2 loop X := X - 1; Y := Y * X; end loop;

      -- boucle d'attente jusqu'à ce qu'une action externe,
                -- ou d'un  autre processus, mette Signal à Vrai
Signal : Boolean := False;
while not Signal loop null; end loop;

                        -- boucle infinie avec sortie calculée
loop
   lire_l_article_suivant;
   exit when End_Of_File; -- sortie si Vrai
   jouer_l_article_lu;
end loop;

-- on l'utilise beaucoup pour le code des processus cycliques, ou démons, qui ne se terminent jamais
```

RETURN et EXIT

return

```
1) sortie d'une fonction
2) sortie d'une procédure, utile si longue alternative
procedure Toto is
begin if OK then S := S+Z; ... return: end if; S := S - Z ; end Toto;
procedure Toto is -- équivaut à
begin if OK then S := S+Z; .... else S := S - Z ; end if; end Toto;
```

exit

```
sortie de boucle
```

PROGRAMMATION À GROS GRAIN

("PROGRAMMING IN THE LARGE")

PAQUETAGE

<SPECIFICATION> interface vers le reste du programme
<CORPS> partie cachee inaccessible

COMPILATION SEPAREE

unités de bibliothèques :
sous-programmes (procédures et fonctions) et paquetages

CONTROLE DE VISIBILITE

accès à une unité de bibliothèque par WITH
sinon structure de bloc classique pour tout le texte sauf les corps de paquetage qui restent toujours cachés
surchage et renommage possibles

TYPES ABSTRAITS

types de données plus sous-programmes d'accès
paquetage avec type privé: PRIVATE ou LIMITED PRIVATE
nom du type connu hors du paquetage, mais pas d'accès

PROGRAMMATION PAR OBJETS

avec paquetage et classes
héritage simple
types taggés

REUTILISATION DU LOGICIEL

par unités de bibliothèque
par unités génériques: types en paramètres
par objets, classes de types (T'class), classes abstraites

PAQUETAGE

```
package Filentier is -- déclaration de l'interface (ads de Gnat)
   procedure Ajouter(X: In Integer);
   function Premier return Integer;
   procedure Oter;
   function EstVide return Boolean;
   Plein, Vide : exception;
end Filentier;

package body Filentier is -- déclaration du corps (adb de Gnat)
   Max: Constant := 100;
   T: array(1..Max) of Integer; --Tampon Circulaire
   Tete, Queue : Integer := Max / 2;
   Taille: Integer := 0;
   Procedure Ajouter(X: in Integer) is
   begin
      if Taille = Max then raise Plein; end if;
      T(Queue) := X; Taille := Taille + 1;
      Queue := Queue mod Max + 1;
   End Ajouter;
   Procedure Oter Is
   begin
      if Taille = 0 then raise Vide; end if;
      Tete := Tete mod Max + 1; Taille := Taille - 1;
   end Oter;
   Function Premier return Integer is
   begin
      if Taille = 0 Then raise Vide; end if;
      Return T(Tete);
   end Premier;
   Function EstVide return Boolean is
   begin return Taille = 0; end EstVide;
begin                  -- zone d'initialisation du module
   null;               -- operation nulle
exception
   when Plein => Put("Ajout impossible car file pleine");
   when Vide => Put("operation impossible car file vide");
end Filentier;
```

UTILISATION DU PAQUETAGE

```
procedure Essai is
   package Filentier is <specification> <corps> end Filentier;
   M, N: Integer;
begin ...
   Filentier.Ajouter(M); ...
   if not Filentier.EstVide then
      N := Filentier.Premier; Filentier.Oter;
   end if; ...
end Essai;
```

OU MIEUX

```
procedure Essai is
   package Filentier is <specification> <corps> end Filentier;
   M, N: Integer; use Filentier; -- factorisation du préfixe
begin ...
   Ajouter(M); ...
   if not EstVide then
      N := Premier; Oter;
   end if; ...
end Essai;
```

OU MIEUX ENCORE

```
package Filentier is <specification> <corps> end Filentier;
   -- mis en bibliothèque

with Filentier; use Filentier; -- importation par with
package Principal is
   M, N: Integer;
begin ...
   Ajouter(M); ...
   if not EstVide then
      N := Premier; Oter;
   end if; ...
end Principal;
```

PORTEE DES DECLARATIONS

selon structure de bloc classique
sauf corps du paquetage
+ extension de portée d'unité de bibliothèque par la clause with

REGLES DE VISIBILITE

selon structure de bloc classique
+ clause use **pour rendre visible directement**
les déclarations de la partie spécification d'un paquetage
sans avoir à les qualifier par le nom du paquetage

DECLARATION DE SURNOM, D'ALIAS

par la clause renames **ou par l'utilisation de** subtype

```
package Numero is new Numerotation(IdProc);
function NomDuProcessus return IdProc
                        renames Numero.Numerote;
```

SURCHARGE DE SOUS-PROGRAMME

notion de signature grâce au type des paramètres

Exemple: le paquetage SI_B

```
with Text_Io;
package SI_B is
   subtype Positive_Count is Text_Io.Positive_Count;
   procedure Put(Item: in Character) renames Text_Io.Put;
   procedure Put(Item: in String) renames Text_Io.Put;
   procedure New_Line(Spacing: in Positive_Count := 1)
           renames Text_Io.New_Line;
    ...
end SI_B;
```

GENERICITE

exemple de composant de gestion de files ou listes

```
generic                                          -- déclaration
   TailleFiles : Positive := 200;                -- des entités génériques
   type Element is private;
package LesFiles is
   type File is private;
   FileVide; PlusDePlace: exception;
   function Premier(LaFile: in File) return Element; -- copie
   function EstVide(LaFile: in File) return Boolean;
   procedure Oter(LaFile: in out File); --supprime le premier
   procedure Ajouter(LaFile: in out File; Val: in Element);
      --ajoute Val en queue de LaFile
private
   type Lien is 0.. Taille_Files ;
   type File is
      record
         Taille: Natural := 0;
         Tete, Queue: Lien := 0;
      end record;
         -- files avec pointeur de tete et de queue
end Les_Files ;

with Les_Files ; -- importation, utilisation du paquetage générique
procedure Essai_Les_Files  is
   --création des types à partir du paquetage générique
   package TypeListeI1 is new Les_Files (Element => Integer);
   package TypeListeI2 is new Les_Files (100, Integer);
   subtype Message is String(1..6);
   package TypeListeS1 is new Les_Files (Element => message);
   -- creation des files de chaque type
   La1, La2: TypeListeI1.File; --file d'entier, au plus 200 éléments
   Lb1, Lb2: TypeListeI2.File; --file d'entier, au plus 100 éléments
   Lc1, Lc2: TypeListeS1.File; --file de chaine, au plus 200
   use TypeListeI1, TypeListeI2, TypeListeS1;
   A,B : integer := 10; C: message := "abcdef"; D: message;
begin
   for I in 1..55 loop
      Ajouter(La1,I);
      Ajouter(Lb2, Premier(La1)); Ajouter(Lb2, A); Oter(Lb2);
      Ajouter(Lc1, C);
   end loop;
   B := Premier(Lb2); D := Premier(Lc1); -- B = 1 et D = "abcdef"
end Essai_Les_Files;
```

```
package body Les_Files is        -- schéma d'implantation  (adb)
   type Cellule is  record
         Valeur: Element;
         Suivante: Lien := 0;
      end record;
   Table: array(1..TailleFiles) of Cellule;
      -- files linéaires avec pointeur de tête et de queue
      -- les files sont implantées dans un tableau,
      -- les pointeurs sont des index dans le tableau
      -- la cellule du dernier élement d'une file a 0 comme suivant
   Cellibres: File := (Taille => TailleFiles,
                                 Tete =>1, Queue =>TailleFiles);
   procedure Couper_Coller(Ls, Ld: in out File) is
   -- la premiere cellule de Ls est mise en queue de Ld
   begin
      if Ls.Taille = 0 then raise FileVide; end if;
      if Ld.Taille = 0 then Ld.Tete := Ls.Tete; -- cas a part
      else Table(Ld.Queue).Suivante := Ls.Tete;
      end if;
      Ld.Queue := Ls.Tete;
      if Ls.Taille = 1 then Ls.Tete:= 0; Ls.Queue := 0; --cas a part
      else
         Ls.tete := Table(Ls.Tete).Suivante;
         Table(Ld.Queue).Suivante := 0;
      end if;
      Ld.Taille := Ld.Taille + 1;
      Ls.Taille := Ls.Taille - 1;
   end Couper_Coller;
   function Premier(LaFile: in File) return Element is
   begin
      if LaFile.Taille = 0 then raise FileVide; end if;
      return  Table(LaFile.Tete).Valeur;
   end Premier;
   function EstVide(LaFile: in File) return Boolean is
      begin return LaFile.Taille = 0; end EstVide;
   procedure Oter(LaFile: in out File) is
   begin couper_coller(LaFile, Cellibres); end Oter;
   procedure Ajouter(LaFile: in out File; Val: in Element) is
   begin
      couper_coller(Cellibres,  LaFile);
      table(LaFile.Queue).Valeur := Val;
   end Ajouter;
begin
   for I in 1..TaillesFiles - 1 loop
      Table(I).Suivante := (I + 1);
   end loop;  -- la dernière est déjà initialisée avec suivante = 0
end Les_Files;
```

STRUCTURATION DES TÂCHES

On exprime la notion de processus avec le type task

```
with SI_B; use SI_B;
procedure SYST_B is
   X,Y: item; --données globales pour toutes les taches
   package Filentier is ... end Filentier; --global

   task type Producteur; task type Consommateur;

   task body Producteur is
      Z: item; -- donnée propre, locale à chaque producteur
      procedure Deposer is
         A: item; --donnée propre à chaque appel de Deposer
      begin ... end Deposer;
   begin ... end Producteur;

   task body Consommateur is          end Consommateur;

begin
    X := ...; Y := ...; --initialisation des variables globales
   declare
      Prod : Producteur; -- déclare une tâche du type Producteur
      Cons: array(1..5) of Consommateur; -- tableau de t^âches
   begin null end;
end SYST_B;
```

Avec cette écriture, on est sûr que le programme principal (la tâche principale) initialise bien les variables globales avant de créer-activer la tâche Prod et les 5 tâches Cons(1), ..., Cons(5).

IMPORTANT : une tâche n'est pas un élément de bibliothèque donc une tâche ne peut être mise en bibliothèque qu'en appartenant à un sous-programme ou à un paquetage

AUTRES ASPECTS : pour la synchronisation entre tâches, le langage Ada propose le rendez-vous ou le partage d'objets protégés. Les sémaphores utilisés dans le pseudo langage du cours systèmes sont codés par des objets protégés (voir le chapitre 9 du cours SRI_B).

NOTION DE PROCESSUS EXPRIME PAR LE TYPE TÂCHE

```
-- déclaration d'un nouveau type tâche
task type Producteur;
task type Consommateur;

-- déclarations statiques de tâches de ces types

Prod: Producteur;  --déclare une tâche
Cons: array(1..5) of Consommateur;
                                   -déclare un tableau de 5 tâches

-- déclarations dynamiques de tâches, avec pointeurs

type A_Producteur is access Producteur; -- type pointeur
A_Prod: A_Producteur; -- un pointeur du type
--plus loin dans le corps du programme
A_Prod := new Producteur;  -création dynamique
```

CHACUNE DES TÂCHES DÉCLARÉES SE DÉROULE EN PARALLÈLE COMME SI IL Y AVAIT AUTANT DE PROCESSEURS QUE DE TÂCHES

```
-- définition du corps de chaque tâche ou type de tâche

task body Producteur is
<déclaration des variables de travail de chaque tâche du type producteur qui sera créée>
begin
   <programme d'un producteur>;
end Producteur;

task body Consommateur is
<déclaration des variables de travail, locales, privées, à chaque tâche du type consommateurqui sera créée >
begin
   <programme d'un consommateur>;
end Consommateur;
```

LA PROGRAMMATION OBJET EN ADA

Dans le cours on utilise des objets simples et on utilise le paquetage comme conteneur pour les composants et les méthodes de l'objet. Ada permet de définir simplement les méthodes exportées par la partie <déclaration du paquetage> et tout ce qui est dans la partie <corps du paquetage> reste caché. C'est pourquoi, l'initialisation des composants est faite à l'initialisation du corps.

Dans le cours on utilise aussi des types abstraits d'objets et la généricité permet de contruire des types abstraits qui ont un moule commun.

Mais la programmation objet en Ada utilise aussi la notion de classe et les types taggés, ce qui permet l'héritage de type avec :
- **l'extension de types,**
- **le polymorphisme,**
- **les liaisons dynamiques.**

La programmation en termes d'objets répartis est obtenue par la catégorisation des paquetages. Celle-ci permet de définir des types distants (remote_types) et des paquetages interfaces d'appels de procédures distante (remote_call_interface) ou de types distants(remote_access_types).

Un bon ouvrage pour aborder la programmation par objets en Ada est le livre de J.P. Rosen, Méthodes de génie logiciel avec Ada 95, InterEditions 1995.