ADA

SOMMAIRE

LES BASES DU ADA

Constantes littérales

- *Entier* : 7
- *Réel* : 7.0
- *Caractère* : 'A'
- *Chaîne* : "ABC"

Noms

Renommage[1]

On peut utiliser des alias de la manière suivante :

```
f' rename f ;
```

Cela ne génère pas de nouveau type, mais constitue un simple renommage de *f* en *f'*.

Surcharge des noms

Le mécanisme de surcharge permet à l'utilisateur d'utiliser le même nom pour des entités différentes, à condition que l'utilisation du nom ne soit pas ambiguë.

On donne ci-après un exemple raisonnable de l'utilisation de la surcharge pour des types énumératifs :

```
type type_de_capteur is (temperature, humidite, pression) ;
type alarme is (normal, temperature, intrusion) ;
```

Une mauvaise utilisation de ce mécanisme peut s'avérer incontrôlable, il convient donc de s'en servir avec précaution.

Opérateurs

Opérateurs standards

On donne une liste (non exhaustive) des opérateurs Ada globalement supportés par les types abstraits :

additives	`+ -`
appartenance	`in not in`
exponentiation	`**`
multiplicatives	`* / mod rem`
relationnelles	`< <= > >=`
unaire	`+ - abs`
logiques	`and or xor not`

[1] Cela peut s'avérer utile pour propager la visibilité d'un paquetage à un autre.

Attention : L'affectation := n'est pas un opérateur en soi dans la norme Ada, ni non plus le test d'égalité = et d'inégalité /= !!!

Ada dispose également d'opérateur d'évaluation partielle de condition, similaire au C : *and then* et *or else*.

Surcharge des opérateurs

Tous les opérateurs sont surchargeables. Il est donc possible de redéfinir un opérateur pour qu'il s'applique à un type abstrait.

Par exemple, pour un type abstrait *matrice* :

```
function "*" (x,y : matrice) return matrice is
begin
…
end "*" ;
```

Il faut veiller à ne pas utiliser l'opérateur que l'on surcharge dans le corps de fonction, ce qui induirait une définition récursive ! Cela est essentiellement problématique lors de la surcharge du test d'égalité, mais il y a toujours moyen de contourner le problème…

Instructions

Ada est un langage structuré, qui possède une série d'instructions puissantes permettant d'écrire des algorithmes. Il fournit des instructions *de contrôle séquentiel, itératives et conditionnelles*, sans compter *les instructions spéciales*, comme celles concernant les tâches et les exceptions pour lesquels des chapitres particuliers sont consacrés.

Contrôle séquentiel

- *Instruction d'affectation* : L'instruction d'affectation donne à une variable une valeur nouvellement calculée.

```
compteur := compteur + 1 ;
anniversaire.annee := 1995 ;
```

- *Appel de sous-programme* : On renvoie au chapitre suivant traitant en détail des sous-programmes.

Il existe d'autres instructions séquentielles (*null, return…*), mais nous ne les aborderons pas.

Contrôle conditionnel

Ada propose les instructions *if* et *case* pour le contrôle conditionnel.

- L'instruction *if* et la structure *if then else* :

```
if taille_tampon = taille_tampon_max then
      traiter_debordement ;
end if ;

if etat_vanne = ouvert then
      lire_debit(grandeur) ;
else
      lire_pression(valeur) ;
end if ;
```

- L'instruction *case* : Sélection à choix multiple.

```
case taille_tampon is
      when taille_tampon_max / 2    =>    emettre_avertissement ;
                                          prendre_nouvelle_valeur ;
      when taille_tampon_max        =>    traiter_debordement ;
      when others                   =>    prendre_nouvelle_valeur ;
end case ;

case couleur_pixel is
      when rouge | vert | bleu      =>    augmenter_saturation ;
      when cyan .. blanc            =>    mettre_au_noir ;
      when others                   =>    null ;
end case ;
```

Contrôle itératif

Les itérations en Ada s'obtiennent au moyen de l'une des formes d'instruction *loop* (boucle). Ada autorise la boucle de base, la boucle avec compteur, et la boucle tant-que. L'instruction *exit* à l'intérieur de la boucle permet d'abandonner l'itération.

- *Boucle de base* :

```
loop
      lire_modem(symbole) ;
      exit when fin_de_transmission ;
      afficher(symbole) ;
end loop ;
```

- *Boucle avec compteur* :

```
for i in liste'range
loop
      somme := somme + liste(i) ;
end loop ;
```

Le compteur *i* n'est pas déclaré, il est pris directement en compte par le compilateur (variable registre). La portée du compteur est local à la boucle.

- *Boucle tant-que* :

```
while donnee_disponible
loop
      read(mon_fichier, tampon_d_entree) ;
      traiter(tampon_d_entree) ;
end loop ;
```

LES TYPES ABSTRAITS

Un type de données abstrait caractérise :
- un ensemble de valeur,
- un ensemble d'opérations applicables aux objets du type.

Généralités

Ada dispose de plusieurs classes de types :
- les types de données scalaires (entiers, réels, caractères…),
- les types de données composites (tableaux et articles),
- les types de données accès (pointeur),
- les types de données privés,
- les sous-types et les types dérivés.

Types de données scalaires

On distingue les types *entiers*, *réels* et *énumératifs*. On donne ci-dessous un tableau récapitulatif (non exhaustif). Les types entiers et énumératifs représentent les types discrets.

entiers	`integer`	entiers relatifs
	`positive`	entiers strictement positifs
	`natural`	entiers positifs
réels	`float`	
énumératifs	`boolean`	`true` ou `false`
	`character`	les caractères

L'utilisateur peut définir de nouveaux types de la façon suivante :

```
type indice is range 1..50 ;

type mass is digit 10 ;                          -- type point flottant
type voltage is delta 0.1 range -5.0..+5.0 ;     -- type point fixe

type couleur is (bleu, rouge, jaune) ;
```

Les attributs

Les types abstraits possèdent des attributs qui les décrivent… et que l'on peut référencer selon la syntaxe suivante ; on donne l'exemple d'un tableau :

```
tableau : array (1..10) of integer ;

tableau'range
tableau'first
tableau'last
```

Les déclarations de variables et de constantes

Les types de données ne fournissent qu'un mécanisme pour décrire la structure de données. Les déclarations, quant à elles, crée des instances (des objets) d'un type donné. En Ada, les déclarations d'objets permettent de créer des variables et des constantes.

Voici par exemple des déclarations de variable :

```
compteur : integer ;
anniversaire : date ;
mon_tampon : piles.pile ;
```

Voici des déclarations d'objets constantes :

```
pi : constant := 3.14 ;
telephone : constant integer := 05_49_55_06_50 ;
```

Les conversions de types

Le langage Ada est fortement typé, ce qui signifie qu'on ne peut pas combiner directement des objets de types différents. Il faut procéder à des conversions de type explicites (avec la fonction de conversion appropriée), comme suit :

```
type chiffre is integer range 0..9 ;
c : chiffre ;
x : integer ;

c := 5 ;
x := integer(c) ;
```

Il est possible cependant d'inhiber la vérification de type au moyen de la procédure *unchecked_conversion* !

Types anonymes

Considérons les exemples suivants d'utilisation d'un type tableau anonyme :

```
a : array (1..100) of integer ;
b : array (1..100) of integer ;
```

Bien que *a* et *b* paraissent de même type, le compilateur n'est pas en mesure de le reconnaître ; ce qui signifie que l'on ne pourra pas écrire :

```
a := b ;
```

si l'envie nous en prenait ! Cependant il est toujours possible de convertir la variable *a*, comme suit :

```
type mes_tableaux is array (1..100) of integer ;
c : mes_tableaux ;
c := mes_tableau(a) ;
```

Les tableaux

Quelques exemples

- *tableau indexé par un type énumératif défini par l'utilisateur :*

```
type pixel is array (couleur) of float ;
```

- *type tableau à six éléments d'un type défini par l'utilisateur :*

```
type capteur is array (index range 5..10) of voltage ;
```

- *tableau à deux dimensions* :

```
type echiquier is array (1..8,1..8) of couleur ;
```

Deux types de déclaration pour les tableaux

- *Les tableaux contraints* : La taille du tableau est connue à la compilation. Voici la définition d'un tableau d'entier de 100 cases.

```
type indice is range (1..100) ;
type mes_tableaux is array(indice) of integer ;
```

- *Les tableaux non contraints* : La taille du tableau est seulement connue à l'exécution du programme.

```
type mes_tableaux is array(integer range < >) of character ;
v : mes_tableaux(1..100) ;
```

- *Les tableaux dynamiques* : Pointeurs…

Utilisation des agrégats pour l'initialisation des tableaux

On distingue trois catégories d'agrégats :

- *les agrégats définis par position* :

```
a(1..5) := (3,6,9,12,15) ; ???
```

- *les agrégats définis par nom* : ???

```
a(1..5) := (5 => 15, 4 => 12, 1 => 3, 3 => 9, 2 => 6) ;
lignes'(1|15 => '*', others => 'u') ;
m := (1..25 (1..25 => 1)) ; -- initialisation d'un tableau 25x25
```

- *les agrégats mixes* : On mélange les deux techniques précédentes…

```
type couleur is (blanc, bleu, rouge, jaune, noir) ;
type drapeau_tricolore is array(1..3) ;
drapeau_france : drapeau_tricolore(bleu, 3 => rouge, 2 => blanc) ;
```

Les chaînes de caractères

Le type chaîne de caractère est un type tableau non contraint prédéfini :

```
type string is array(positive range < >) of character ;
```

L'affectation de chaîne est complète, c'est à dire qu'il est interdit d'écrire :

```
chaine : string(1..20) ;
chaine := "toto" ;
```

On dispose de la fonction concaténation $ des chaînes de caractères dont les spécifications sont les suivantes :

- `character $ character → string`
- `string $ character → string`
- `character $ string → string`
- `string $ string → string`

Considérons les exemples suivants d'utilisation de cette fonction :

```
c : character ;
s : string(5..10) ;
t : string(15..20) ;
```

Bien observer le résultat des opérations qui suivent (le premier indice gagne) :

```
c $ s ; -- 1..6
s $ c ; -- 5..11
c $ s ; -- 5..16
```

Les articles

Quelques exemples

Considérons les exemples suivants :

- *article à trois composants* :

```
type date is record
      jour : integer range 1..31 ;
      mois : integer range 1..12 ;
      annee : natural ;
end record ;
```

- *article à cinq composants* :

```
type vanne is record
      nom : string(1..20) ;
      emplacement : string(1..30) ;
      ouverte : boolean ;
      debit : float range 0.0..30.0 ;
      verifiee : date ;
end record ;
```

Utilisation des agrégats pour l'initialisation des articles

De la même manière que pour les tableaux, on utilise les agrégats pour initialiser des articles :

```
      anniversaire : date'(12,10,1978) ;

      anniversaire : date'(jour => 12, mois => 10, annee => 78) ;
```

Les types de données accès

Lorsque des objets doivent être manipulées dynamiquement, il convient d'utiliser des valeurs *accès* qui pointent sur des objets.

La syntaxe est la suivante :

```
type pointeur_tampon is access tampon ;
```

Par défaut, les variables de type accès sont initialisée à *null*.

Utilisation de *new* pour la création d'un objet dynamique

L'instruction *new* permet de créer un nouvel objet qu'il est possible d'affecter à un pointeur (pointant sur le type de l'objet). L'exemple, qui suit, d'une liste chaînée illustre ce propos :

type cellule ; -- simple déclaration
type lien is access cellule ;

```
type cellule is record
      data : contenu ;
      suivant : lien ;
end record ;

l1, l2, l3 : lien ;

l1 := new cellule ;
l1.data := 20 ;
l1.suivant := null ;

l2 := new cellule'(30,l1) ;
l3 := l2 ;
```

La libération de mémoire est réalisé au moyen de la commande *free* :

```
l3 = null ;
free(l2) ;
free(l1) ;
```

En Ada, il faut veiller, à ce que le seul pointeur que l'on passe en paramètre pointe sur la cellule mémoire qui va être libéré, sans quoi une erreur est produite !

Les types dérivés

Les type dérivés permettent de créer des familles de types relatifs à un type de base. Les types dérivés sont incompatibles avec leur type père ; ils définissent un nouveau type à part entière.

N.B. : Pour la démonstration suivante, on suppose placer les déclarations dans la spécification d'un paquetage *derivation* par exemple.

La syntaxe est la suivante :

```
type my_integer is new integer ;
```

Lorsqu'un type comme *my_integer* dérive d'un type prédéfini comme *integer*, il hérite[2] alors de tous les attributs et de toutes les fonctions de ce type, entre autres de la fonction "+" et de l'affectation.

Les opérations suivantes seront donc valides :

```
x,y,z : my_integer ;
z := x + y ; -- "+" hérité de integer
```

Mécanisme d'héritage

Par la suite, on peut définir de nouvelles fonctions s'ajoutant aux fonctions prédéfinies de *my_integer* comme fait *my_plus*.

```
function my_plus(x,y : in my_integer) return my_integer ;
```

[2] Sauf si le type dérivé a préalablement été déclaré privé (cf. section suivante), auquel cas l'héritage est plus restreint voire même nul.

Après quoi, si l'on définit un nouveau type *your_integer* dérivé de *my_integer* :

```
type your_integer is new my_integer ;
```

Celui-ci héritera de toutes les fonctions relatives au type *my_integer* dont la définition est antérieure à celle-là même de *your_integer*, mais pas de celles dont la définition pourrait venir après.

```
function my_plus_bis(x,y : in my_integer) return my_integer ;
```

Par conséquent, cette dernière fonction relative à *my_integer* n'est pas transmise par héritage au type *your_integer* !

```
x,y,z : your_integer ;
z := my_plus(x,y) ;          -- possible
z := my_plus_bis(x,y) ;      -- impossible
```

On peut néanmoins forcer l'utilisation de *my_plus_bis* avec *your_integer*, grâce à des conversions.

```
z := your_integer(my_plus_bis(my_integer(x),my_integer(y))) ;
```

Surcharge de fonction héritée

On peut surcharger une fonction héritée ; dès lors, la nouvelle fonction masque la précédente.

Dans l'exemple suivant, *your_integer* a hérité de *my_integer* la fonction *my_plus* qu'il redéfinit pour lui-même.

```
function my_plus(x,y : in your_integer) return your_integer ;
```

Cependant, cela n'affecte en aucun cas les relation d'héritage précédentes, qui sont statiques.

Les types privés

Les types privées sont utilisés uniquement dans les paquetages. Nous les aborderons plus en détail dans cette partie.

L'ensemble des valeurs et la structure sont cachés à l'utilisateur. Ce type interdit le mécanisme d'héritage. ???

Type *private*

Pour les types *private*, l'affectation et le test d'égalité et d'inégalité sont disponibles en plus des opérations définies dans la spécification de paquetage correspondante.

On donne un exemple pour illustrer la syntaxe :

```
type pile is private ;
```

Type *limited private*

Pour les types *limited private*, seules les opérations définies dans la spécification de paquetage correspondante sont disponibles.

La syntaxe est sensiblement la même:

```
type pile is limited private ;
```

On peut cependant surcharger le test d'égalité, mais pas l'affectation ! Dans ce cas, il faut trouver un autre nom…

En conséquence, des fonctions qui retourneraient le type *pile* ou des procédures qui utiliseraient le des paramètres *out pile* ne sont en aucune façon être utilisable par le client !

Un type défini à partir d'un *limited* est *limited*. ???

Les sous-types

Le sous-type restreint l'univers du type père, en gardant les mêmes méthodes. Le sous-type est compatible avec le type père.

On donne un exemple pour illustrer la syntaxe :

```
subtype number_base is range 2..16 ;
```

UNITES DE PROGRAMMES

Le langage Ada est basé sur des unités de programmes que sont :
- *les sous-programmes* (procédure et fonction),
- *les tâches* (multi-threading),
- *les paquetages* (encapsulation).

Les unités de programmes Ada se divisent en deux parties :
- *une spécification*, qui identifie le nom du sous-programme avec ses paramètres et représente l'interface client ;
- *un corps*, qui contient l'implémentation de l'interface défiine.

Sous-programmes

En Ada, on distingue deux sortes de sous-programmes : *les procédures* et *les fonctions*.

Fonction

Une fonction retourne une valeur (mécanisme d'affectation caché). Par ailleurs, les fonctions ne peuvent avoir que des paramètres *in*.

```
function pgcd (alpha, beta : in nombres) return nombres is

      -- déclarations locales
      diviseur : nombres := abs(alpha) ;
      dividende : nombres := abs(beta) ;
      reste : nombres ;

begin
      -- algorithme d'Euclide pour le pgcd

      while diviseur /= 0 loop
            reste := dividende mod diviseur ;
            dividende := diviseur ;
            diviseur := reste ;
      end loop ;

      return dividende ;

end pgcd ;
```

Procédure

Une procédure peut n'avoir aucun paramètre. A la différence des fonctions, ses paramètres peuvent être des trois types (cf. section suivante).

```
procedure faire_tourner (points : in out coordonnees ;
   angle : in radians) is
-- déclarations locales
begin
-- suite d'instructions
end faire_tourner ;
```

Paramètres entrée / sortie

Les termes *in* (entrée), *out* (sortie) et *in out* (entrée-sortie) spécifient le mode ou la direction du flot de données par rapport au sous-programme. Le paramètre *in* est le paramètre par défaut, quand aucun paramètre n'est spécifié.

Les paramètres *in* sont des paramètres constants, qui sont dupliqués à l'entrée du sous-programme. Les paramètres *out* sont affectés une seule fois à la fin et ne sont pas lisibles dans le sous-programme. Les paramètres *in out*, quant à eux, des paramètres constants, qui sont dupliqués à l'entrée du sous-programme, puis affecté à la fin une seule fois.

Appel de sous-programme

L'appel de sous-programme fait correspondre les paramètres réels aux paramètres formels. à ce niveaux, deux techniques sont envisageables *l'association par position* ou *par nom*.

- *Association par position* :

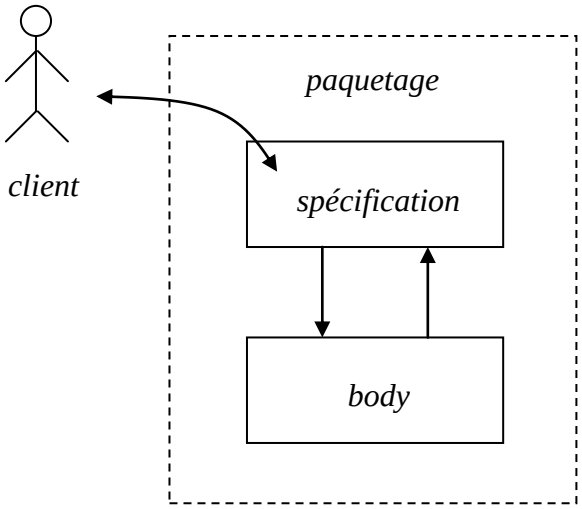
```
faire_tourner(points, 30.0) ;
```

- *Association par nom* : Exemple avec une association par nom.

```
faire_tourner(points, angle => 30.0) ;
```

Paquetages

Les paquetages permettent à l'utilisateur d'encapsuler un groupe d'entités logiquement reliées (un ensemble de ressource de calcul). En tant que tels, les paquetages supportent directement les principes d'abstraction de données et de dissimulation d'information.



Définition

Comme toute unité de programme, le paquetage consiste en une spécification et un corps. La spécification constitue le contrat du programmeur avec le client du paquetage. Le client n'a jamais besoin de voir le corps de paquetage, qui contient l'implémentation de la spécification, aussi ces deux parties de programmes sont-elles séparées !

Considérons l'exemple d'un paquetages *piles* définissant une pile abstraite. On donne tout d'abord sa spécification.

```
package piles is
```

```
        type pile is private ;
        procedure empiler(element : in integer ; sur in out pile) ;
        procedure depiler(element : out integer ; depuis in out pile) ;

private
        maximum_elements : constant integer := 100 ;
        type liste is array (1..maximum_elements) of integer ;
        type pile is record
              structure : liste ;
              sommet : integer range 1..maximum_elements := 1 ;
        end record ;

end piles ;
```

Le corps de ce paquetage a la forme suivante :

```
package body piles is

        procedure empiler(element : in integer ; sur in out pile) is
        ...
        begin
        ...
        end empiler ;

        procedure depiler(element : out integer ; depuis in out pile) is
        ...
        begin
        ...
        end depiler ;

end piles ;
```

Utilisation des types privés

On rappelle que les types privés s'utilisent uniquement dans les paquetages. Tout comme les autres types, les types privés définissent un ensemble de valeurs et d'opérations applicables. Mais contrairement aux autres types, les structures des types privés ne sont pas visibles pour le client. De plus, l'implémenteur peut définir certaines opérations pour des types privés, et celle-ci deviennent les seules que l'utilisateur peut employer. Les types privés fournissent donc un mécanisme permettant de faire respecter la dissimulation d'information et de créer de nouveaux types de données abstraits. Cf. l'exemple précédent.

Utilisation d'un paquetage

Si l'on veut faire appel à des sous-programmes, il convient de le préciser au moyen de la syntaxe suivante son paquetage d'origine (mécanisme de compilation séparée) :

```
with text_io ;
```

Ce faisant, on peut utiliser les fonctions issu de ce paquetage ; par exemple :

```
text_io.new_line ;
text_io.put_line("Hello World !") ;
```

Ou plus simplement, en utilisant un raccourci d'écriture (risque d'ambiguïté) pour ce paquetage :

```
use text_io ;
new_line ;
put_line("Hello World !") ;
```

Tâches

Les tâches constituent une autre classe d'unité de programme Ada. Une tâche peut être représentée comme des opérations simultanées indépendantes qui communiquent en se passant des messages, par mécanisme de rendez-vous.

Nous n'aborderons pas les tâches dans ce cours.

TRAITEMENT D'EXCEPTIONS

Ada permet un traitement d'exception structuré en blocs. Les exceptions, qui sont typiquement (mais pas nécessairement) des conditions d'erreurs, peuvent être prédéfinies (comme une erreur numérique dans le cas d'une division par zéro) ou définies par l'utilisateur (comme une condition de débordement de tampon). Les exceptions prédéfinies sont levées implicitement, les exceptions utilisateur le sont explicitement.

```
procedure traiter_temperature is
      temperature : float ;
      trop_chaud : exception ;
begin
      loop
            get(temperature) ;
            normaliser(temperature) ;
            if temperature > limite then
                  -- lancement d'une exception
                  raise trop_chaud ;
            end if ;
      end loop ;

-- traite-exception
exception
      when trop_chaud => ...

end traiter_temperature ;
```

Si une exception se produit, le traitement normal est suspendu et le contrôle passe au *traite-exception*. Si aucun traite-exception n'est présent dans le bloc courant, alors le contrôle passera au niveau lexical suivant (appelant), jusqu'à ce que l'on trouve un traite-exception ou que l'on atteigne le système d'exploitation (l'environnement).

UNITES DE PROGRAMME GENERIQUES

Ada permet de se servir des unités génériques comme mécanisme de construction de composants logiciels réutilisables. Considérons un algorithme qui s'appliquerait à divers types de données ; les unités génériques permettent de définir un modèle commun et paramétré.

Il faut noter que cette paramétrisation se produit à la compilation.

Une définition générique ne produit aucun code ; elle définit seulement un modèle d'algorithme. L'utilisateur doit instancier l'unité générique pour en créer une copie locale.

Les paramètres génériques

Types génériques

On donne une liste d'exemple pour la définition d'un type générique *element* plus ou moins contraint.

`type element is limited private ;`	définition la plus générale[3]
`type element is private ;`	
`type element is access ... ;`	
`type element is (< >) ;`	type énuméré
`type element is range < > ;`	
`type element is delta < > ;`	type pont fixe
`type element is digit < > ;`	type point flottant
`type element is array(index) of ... ;`	tableau contraint
`type element is array(index range < >) of ... ;`	tableau non contraint

Mode *in*, *in out*

Les paramètres génériques sont *in* ou *in out*, mais jamais *out* ! En l'absence de précision, les paramètres sont considérés *in*. Les paramètres *in out* fonctionne comme des variables globales modifiables localement, ce qui s'avère quelque peu dangereux[4] !

Paramètres par défaut

Par exemple, ont peut définir un paramètre générique recevant une valeur par défaut, au cas où le client ne prendrait pas le soin de l'initialiser lui-même.

```
generic
      ligne : in integer := 24 ;
      colonne : in integer := 80 ;

package terminal is ...
```

Dans cet exemple, l'utilisation du type *integer* est contraignante pour le client ; mieux vaut utiliser le type énuméré…

[3] Dans le sens où l'on restreint le moins possible le client.
[4] Vis-à-vis de la robustesse.

Instanciation par nom ou par position

L'instanciation par nom utilisant des paramètres avec des valeurs par défaut ne nécessite pas qu'on lui passe tous ces paramètres, ce qui se traduit par les instanciations suivantes valides :

```
package micro is new terminal(24,30) ;          -- par position
package micro is new terminal(colonne => 40) ; -- par nom
```

Sous-programme générique

Considérons l'exemple suivant :

```
generic
      type element is range < > ; -- type discret
      procedure permuter(gauche, droite : in out element) ;

procedure permuter(gauche, droite : in out element) is

tempo : element ;

begin
      tempo := droite ;
      droite := gauche ;
      gauche := tempo ;
end permuter ;
```

On peut ensuite instancier cette procédure pour divers types discrets :

```
procedure permuter_entier is new permuter(integer) ;
procedure permuter_entier is new permuter(my_integer) ; -- surcharge
```

Paramètre générique en sous-programme

- On donne un premier exemple :

```
generic
      ligne : in integer := 24 ;
      colonne : in integer := 80 ;
      with procedure envoyer(valeur : in character) ;
      with procedure recevoir(valeur : out character) ;

package terminal is ...
```

L'instanciation se réalise de la manière suivante :

```
procedure my_envoyer(valeur : in character) is ...
procedure my_recevoir(valeur : out character) is ...

with terminal ;
package my_terminal is new
      terminal(envoyer => my_envoyer, recevoir => my_recevoir) ;

with my_terminal ; use my_terminal ;
...
```

- On donne deux exemples mettant en évidence les problèmes de surcharge ; dans le premier cas, le client doit obligatoirement redéfinir la fonction ; dans le second, il peut choisir d'utiliser la définition visible (standard ou pas), si il en existe une.

```
with function "+" (a,b : in integer) ;

with function "+" (a,b : in integer) is < >;
```

- On donne un dernier exemple sans commentaire :

```
generic
      type t is (< >) ;
      with function next(x : t) return t is  t'succ ;
```

Paquetage générique

Reprenons l'exemple du paquetage piles étudié un peu en avant de ce cours et modifions le pour en faire une unité générique. On obtient :

```
generic
      limite : natural ;
      type donnees is private ;

package piles is

      type pile is private ;
      procedure empiler(element : in donnees ; sur in out pile) ;
      procedure depiler(element : out donnees ; depuis in out pile) ;

private
      type liste is array (1..limite) of donnees ;
      type pile is record
            structure : liste ;
            sommet : integer range 1..limite := 1 ;
      end record ;

end piles ;
```

Nous pouvons ainsi créer plusieurs sortes de piles :

```
package pile_d_entiers is new piles(100, integer) ;
package pile_de_float is new piles(limite => 300, donnees => float) ;
```

Dans le premier cas, la déclaration crée un paquetage logiquement équivalent à celui présenté précédemment dans ce chapitre. On a utilisé une association de paramètres par noms dans le deuxième exemple pour améliorer la lisibilité.