Licence 2 - Mention Mass

Algorithmique et Programmation en Ada

résumé succinct du cours de première année

Joëlle Cohen

2 octobre 2006

Table des matières

Introduction

Ces notes de cours ne prétendent pas être exhaustives ni se substituer en aucune manière aux ouvrages publiés notamment sur le langage de programmation choisi Ada 95 ("Programmer en Ada 95" de J.G.P. Barnes chez Addison-Wesley...). Ce document n'a d'autre but que de fournir aux étudiants de L2 un support de cours leur permettant de revoir les notions du cours de première année utilisant Ada. Les programmes ont été testés sur le compilateur GNAT (GNU Ada Translator) de diffusion libre et gratuite par ftp ://ftp.cs.nyu.edu/pub/gnat

(voir aussi le site http ://www.usafa.af.mil/dfcs/bios/mcc_html/adagide.html pour obtenir un éditeur adapté).

Chapitre 1

Types simples

1.1 Le type Entier

Le type **integer** représente l'ensemble des entiers relatifs $\cdots -2, -1, 0, 1, 2 \cdots$. Sur une machine 16 bits à complément à deux on a alors **integer'first** = -32768=-2^{15} et **integer'last** = 32767=$2^{15} - 1$.

Les valeurs entières peuvent être écrites de différentes façons

- signé ou non signé : +3 -4 52 -126
- dans des bases différentes de la base 10 depuis la base 2 jusqu'à la base 16 ; dans ce cas on écrit tout d'abord la base suivi de # suivi de la valeur dans cette base suivi de # éventuellement suivi de E et de la valeur de l'exposant : les six expressions suivantes valent 24 en base 10

  2#11#E3

  2#11000#

  10#24#

  24

  8#30#

  16#18#

Pour faciliter entre autre les entrées - sorties, par la suite on adoptera

- soit l'utilisation du type prédéfini **integer**
- soit la déclaration d'un **sous-type**[1]comme dans l'exemple suivant :
  SUBTYPE t_dividende IS **integer range** 0 .. 1000 ;

t_dividende représentera les entiers compris entre 0 et 1000 (bornes comprises).

On pourra déclarer plusieurs de ces sous-types et les utiliser pour déclarer des variables comme dans l'exemple suivant :

exemple :

SUBTYPE t_nat IS **integer range** 0 .. 100 ;

SUBTYPE t_pos IS **integer range** 1 .. 100 ;

SUBTYPE t_neg IS **integer range** -100 .. -1 ;

n, p : t_nat := 0 ; x, y : t_neg := -1 ; a, b : t_pos := 1 ;

[1]la notion de sous-type sera vue ultérieurement

n et p sont des variables dont les valeurs seront comprises entre 0 et 100, x et y sont des variables dont les valeurs seront comprises entre -100 et -1, et a et b sont des variables dont les valeurs seront comprises entre 1 et 100. n et p sont initialisés à 0, x et y sont initialisés à -1 et a et b sont initialisés à 1.

Ici les trois sous-types sont issus du même type **integer** et cela nous permet d'utiliser par exemple x et a dans la même expression. Ce n'aurait pas été possible si l'on avait déclaré des types au lieu de sous-types.

Les opérations suivantes prennent deux entiers du même type et renvoient un entier

+ est l'addition

− est la soustraction

* est la multiplication

/ est le quotient euclidien

REM est le reste de la division euclidienne ; il a le signe du dividende et sa valeur absolue est strictement inférieure à la valeur absolue du diviseur

MOD est le modulo ; il a le signe du deuxième opérande et sa valeur absolue est strictement inférieure à la valeur absolue du deuxième opérande

** est l'exponentiation : le premier opérande est mis à la puissance du deuxième opérande qui doit être positif

Les opérations suivantes prennent un entier et renvoie un entier

+ − permettent de signer une valeur entière

ABS est la valeur absolue

Les opérations suivantes renvoient une valeur de vérité (vrai ou faux)

= est le test d'égalité

/ = est test d'inégalité

< > sont les tests d'ordre strict

<= >= sont les tests d'ordre large

En l'absence de parenthèse, une expression sera évaluée de gauche à droite.

La procédure d'entrée depuis le clavier est réalisée par GET : l'instruction GET (a) attend qu'une valeur soit saisie au clavier – c'est-à-dire tapée et suivie d'un retour chariot – puis l'associe à la variable a. Dans le cas où la valeur saisie n'appartient pas au type de la variable a, une erreur d'exécution se produit. Plus exactement, une exception CONSTRAINT_ERROR est levée, mais l'utilisation d'une exception sera vue plus tard.

La procédure de sortie sur l'écran est réalisée par PUT : l'instruction PUT (b) affiche à l'écran la valeur de la variable b au moment de l'exécution de PUT (b).

Afin de pouvoir utiliser ces procédures, il faut introduire le nom de l'unité Ada qui la contient par le mot WITH et permettre l'accès à son contenu par le mot USE .

En ce qui concerne les entiers, puisque l'on a choisi de déclarer les variables directement de type standard **integer** ou bien déclarer un sous-type du type standard **integer** , on fera précéder notre programme des 2 clauses suivantes :

```
WITH Ada.Integer_Text_Io ;
USE Ada.Integer_Text_Io ;
```

1.2 Type booléen

Le type **boolean** est prédéfini et a deux valeurs : False et True.

exemple : f : **boolean** := True ; - - f est de type booléen et initialisé à True

Les opérations booléennes classiques sont

NOT est la négation,

AND est la conjonction $\wedge$,

OR est la disjonction $\vee$,

XOR est le ou exclusif,

AND THEN est la conjonction qui n'évalue pas le deuxième terme si le premier est faux – auquel cas l'expression est fausse globalement,

OR ELSE est la disjonction qui n'évalue pas le deuxième terme si le premier est vrai – auquel cas l'expression est vraie globalement.

NOT a priorité sur les autres opérateurs qui sont de même priorité. Par conséquent l'usage des parenthèses est indispensable pour l'utilisation conjointe de OR , AND , XOR , AND THEN , OR ELSE .

1.3 Type caractère

Le type **character** est l'ensemble des caractères imprimables ou bien formé d'un blanc unique : ce sont les caractères codés en ASCII[2] sur 8 bits dans l'ordre du codage depuis le code 0 jusqu'au code 255. Ces caractères seront encadrés d'apostrophes.

exemple : 'g' et 'G' sont deux valeurs distinctes du type **character** .

On peut utiliser le type **character** prédéfini ou bien le restreindre par une déclaration de sous-type

exemple :

rep : **character** := 'n' ; - - rep est une variable de type **character** initialisé à 'n'

SUBTYPE t_lettre IS **character range** 'A' .. 'z' ;

l : t_lettre := 'a' ; - - l est initialisé à 'a'

En accord avec le code ASCII, les valeurs du type **character** sont classées selon leur code et on peut donc les comparer et les opérations de comparaison renvoyant un booléen sont valides pour le type **character** : = /= < <= > >=.

exemple : '3' < 'B' sera évalué à True et 'Z'> 'a' sera évalué à False.

On a aussi la possibilité d'utiliser des *attributs fonctions* qui s'appliquent au type **character** et qui renvoient des résultats du même type ou bien de type **integer** .

first renvoie le premier caractère

last renvoie le dernier caractère

succ renvoie le caractère suivant dans l'ordre ASCII ou dans le type déclaré

[2] cf. annexe A

pred renvoie le caractère précédent dans l'ordre ASCII ou dans le type déclaré

pos renvoie le numéro de la position (depuis 0) dans l'ordre ASCII ou dans le type déclaré

val prend un numéro et renvoie le caractère dont la position est donnée par ce numéro

remarque : ces attributs sont en fait ceux de tout type *énumératif* qui sera vu ultérieurement.

Les entrées - sorties sont réalisées par les procédures GET et PUT de l'unité Text_Io. On fera donc précéder notre programme des 2 clauses suivantes :

```
WITH Text_Io ;
USE Text_Io ;
```

1.4 Type réel

On sait que les nombres réels que l'on peut représenter sont en fait des nombres décimaux et qu'un réel sera codé par une approximation : la précision de cette approximation dépendra du codage choisi (taille de l'exposant et de la mantisse).

En Ada 95, il y a trois possibilités d'utiliser un type réel : le type *point-flottant* dont la précision sera relative et le type *point-fixe* dont la précision sera absolue et choisie par l'utilisateur ; ce dernier type est accompagné soit d'une contrainte d'intervalle soit d'une contrainte sur le nombre de chiffres de la représentation en base 10.

Dans tous les cas, une valeur réelle devra s'écrire à l'anglo-saxonne soit avec un point au lieu d'une virgule.

exemple : 2.0 0.031 -15.98 +321.00 (bien que les valeurs 2 et 2.0 soient égales la première est du type entier et la deuxième du type réel)

le type point-flottant

Pour le type flottant, on indique un nombre minimum de chiffres décimaux significatifs pour la mantisse et cette précision sera garantie pour toutes les valeurs de ce type.

Il existe un type **float** universel, mais pour éviter de dépendre de la machine sur laquelle on travaille, on optera plutôt pour l'utilisation d'un sous-type du type **float** .

exemple :

```
TYPE t_reel IS digits 5 ; - - la précision sera d'au moins 10^-5
SUBTYPE t_float IS float digits 5 ;
```

Ici, t_float est un sous-type du type **float** de précision minimale 10^{-5}.

le type point-fixe binaire

exemple : `TYPE t_fixe IS delta 0.001 range -1.0 .. 1.0 ;`

Ici, t_fixe représente des nombres compris entre -1 et 1 avec une précision d'au moins 10^{-3}.

le type point-fixe décimal

exemple : `TYPE t_fixed IS delta 0.001 digits 4 ;`

Les valeurs du type t_fixed auront 4 chiffres significatifs avec une précision de 10^{-3} donc seront comprises entre -9.999 et 9.999.

Par la suite, on utilisera uniquement le type point-flottant déclaré comme sous-type du type **float** .

Les opérations

Sur tous ces types, les op'erations suivantes sont valides :

les comparaisons = /= < <= > >=

les signes + −

addition soustraction + −

multiplication division * /

exponentiation **

valeur absolue ABS

Les variables employées devront être du même type et le résultat sera de ce type.

Les entrées - sorties Compte-tenue de la déclaration

SUBTYPE t_float IS **float digits** 5 ;

dans laquelle on peut bien sûr remplacer 5 par une autre valeur entière positive, les entrées - sorties sont réalisées par les procédures GET et PUT de l'unité Ada.float_Text_Io. On fera donc précéder notre programme des 2 clauses suivantes :

```
WITH Ada.float_Text_Io ;
USE Ada.float_Text_Io ;
```

pour pouvoir les utiliser.

1.5 Les entrées - sorties

Comme on l'a vu, Ada impose de préciser les unités que l'on utilisera pour effectuer les entrées - sorties qui pour l'instant seront limitées aux entrées depuis le clavier et aux sorties sur le moniteur.

A ce stade, afin de faciliter l'écriture des premiers programmes, on adoptera le choix de déclarer des sous-types et d'utiliser les clauses

```
WITH Text_Io ;
WITH Ada.Integer_Text_Io ;
WITH Ada.Float_Text_Io ;
USE Text_Io ;
USE Ada.Integer_Text_Io ;
USE Ada.Float_Text_Io ;
```

WITH précise les unités qui seront utilisées et USE donne l'accès à toutes les fonctionnalités des unités introduites par WITH .

On peut préciser le format d'affichage

- des entiers : PUT (n,f) affiche la valeur de n à droite d'un champ d'affichage de f caractères
- des flottants : PUT (x,f1 , f2 , f3) affiche la valeur de x avec f1 chiffres avant la virgule, f2 chiffres après la virgule et f3 chiffres pour l'exposant (f3=0 pour un affichage décimal).

Chapitre 2

Décision et itération

2.1 La décision

En Ada, il y a deux structures pour la décision : l'instruction IF et l'instruction CASE .

if

IF *expression booléenne* B THEN *suite d'instructions* S ;

ELSE *suite d'instructions* S' ;

END IF ;

La sémantique est habituelle. La clause ELSE peut être omise.

case

CASE *expression* E IS

WHEN $choix_1$ => *suite d'instructions* S_1 ;

WHEN $choix_2$ => *suite d'instructions* S_2 ;

WHEN $choix_n$ => *suite d'instructions* S_n ;

WHEN OTHERS => *suite d'instructions* S ;

END CASE ;

L'expression E est évaluée et si elle correspond au
- $choix_1$: S_1 est exécutée
- $choix_2$: S_2 est exécutée
- ...
- $choix_n$: S_n est exécutée
- une valeur autre que celles exprimées par $choix_1 \ldots choix_n$: S est exécuté.

2.2 L'itération

on connaît les bornes

FOR *identificateur_compteur* IN *borne_inf* .. *borne_sup* LOOP

suite d'instructions ;

END LOOP ;

remarque :
- il est inutile de déclarer la variable *identificateur_compteur* qui est automatiquement associée au type des valeurs de *borne_inf* et *borne_sup*.
- ce type doit être **entier** ou **énumératif**

identificateur_compteur prendra les valeurs successives de l'intervalle défini par *borne_inf borne_sup* et pour chacune d'elle *suite d'instructions* sera exécutée.

remarque :
- la variable de contrôle *identificateur_compteur* est **locale** à la boucle et n'a pas d'existence en dehors de cette boucle
- à chaque itération, la valeur de cette variable est constante et ne peut pas être modifiée
- si la valeur de *borne_inf* est supérieure à la valeur de *borne_sup* alors aucune instruction comprise entre LOOP et END LOOP n'est exécutée
- on a la possibilité d'écrire une boucle qui *décrémente* la variable de contrôle depuis *borne_sup* jusqu'à *borne_inf* de la façon suivante :

  FOR *identificateur_compteur* IN REVERSE *borne_inf* .. *borne_sup* LOOP
    suite d'instructions ;
  END LOOP ;

on ne connaît pas les bornes

LOOP

suite d'instructions S_1 ;

EXIT WHEN *expression booléenne* B ;

suite d'instructions S_2 ;

END LOOP ;

remarque : S_1 ou S_2 peuvent être omises.

la suite d'instructions S_1 est exécutée puis *expression booléenne* B est évaluée : si elle est vraie alors l'instruction LOOP est terminée sinon la suite d'instructions S_2 est exécutée et on recommence depuis le début de la boucle LOOP .

remarque :
- à chaque itération, le test de validité de B peut se faire en premier en omettant S_1
- à chaque itération, le test de validité de B peut se faire en dernier en omettant S_2
- si B n'est jamais évaluée à vraie alors la boucle LOOP se poursuit indéfiniment ...
- il faut donc faire en sorte que la valeur de B soit modifiée en cours d'exécution de la boucle LOOP afin de terminer celle-ci.

Chapitre 3

Procédures et fonctions

3.1 Déclaration

Une procédure se déclare de la façon suivante :

PROCEDURE *identificateur_pr* (*liste_para*) IS

partie_déclarative ;

BEGIN

suite d'instructions ;

END *identificateur_pr* ;

- *identificateur_pr* est le nom de la procédure
- *liste_para* est une liste des paramètres formels dans laquelle est précisée
  - l'identificateur de chaque paramètre
  - le type de chaque paramètre
  - le mode de chaque paramètre qui peut être IN , OUT , IN OUT
- dans *partie_déclarative* sont déclarées toutes les variables nécessaires au fonctionnement de la procédure c'est-à-dire les variables que l'on utilisera dans *suite d'instructions*
- les instructions de *suite d'instructions* peuvent alors utiliser les variables déclarées dans *partie_déclarative* ainsi que les paramètres formels de *liste_para*.
- la partie entre IS et END *identificateur_pr* ; est le *corps* de la procédure.

Une fonction se déclare de la façon suivante :

FUNCTION *identificateur_fct* (*liste_para*) RETURN *type renvoyé* IS

partie_déclarative ;

BEGIN

suite d'instructions ;

END *identificateur_fct* ;

- *identificateur_fct* est le nom de la fonction
- *liste_para* est une liste des paramètres formels dans laquelle est précisée
  - l'identificateur de chaque paramètre
  - le type de chaque paramètre
- *type renvoyé* est le type du résultat de la fonction

- dans *partie_déclarative* sont déclarées toutes les variables nécessaires aux calculs effectués par la fonction c'est-à-dire les variables que l'on utilisera dans *suite d'instructions*
- les instructions de *suite d'instructions* peuvent alors utiliser les variables déclarées dans *partie_déclarative* ainsi que les paramètres formels de *liste_para*
- dans *suite d'instructions* doit **obligatoirement** figurer RETURN *valeur* ; où *valeur* sera de type *type renvoyé* ; **cette instruction marque la fin de l'exécution** de la fonction.
- *valeur* peut être une valeur constante, la valeur d'une variable, la valeur d'une expression ...
- la partie entre IS et END *identificateur_fct* ; est le *corps* de la fonction.

Un sous-programme est nécessairement déclaré dans la partie déclarative d'un programme.

Il existe trois modes de paramètres.

- IN Le paramètre formel est alors une constante et sa valeur est celle du paramètre effectif qui lui est associé au moment de l'appel. Un paramètre formel en mode IN ne peut donc figurer qu'à droite d'une affectation dans le corps de la procédure.
- OUT Le paramètre formel est alors une variable et sa valeur à la fin du sous-programme est transmise au paramètre effectif qui lui est associé.
- IN OUT Le paramètre formel est alors une variable et sa valeur initiale au moment de l'appel est celle du paramètre effectif qui lui est associé ; puis sa valeur à la fin du sous-programme est transmise au paramètre effectif qui lui est associé.

remarque : pour une fonction, le mode de chaque paramètre est toujours IN donc aucun de ses paramètres ne devra figurer à gauche d'une affectation dans le corps de la fonction.

3.2 Appel d'une procédure ou d'une fonction

L'appel d'une procédure dans un programme dont elle est sous-programme se fait par l'instruction suivante :

identificateur_pr (*liste_para_effectifs*) ;

où (*liste_para_effectifs*) est la liste des variables sur lesquelles on veut faire agir la procédure *identificateur_pr*.

- Cette liste doit être composée d'autant de paramètres effectifs que (*liste_para*) avait de paramètres formels, l'association se faisant dans l'ordre de la liste.
- Chaque paramètre effectif doit avoir le même type que le paramètre formel auquel il est associé.
- Lorsque le paramètre formel est de mode IN , le paramètre effectif peut être une valeur constante ou une expression de même type que le paramètre formel associé.
- Lorsque le paramètre formel est de mode OUT ou bien IN OUT , le paramètre effectif est une variable dont l'identité est déterminée au moment de l'appel et ne pourra pas être modifiée.

remarque : L'appel à une fonction ne peut se faire que dans l'utilisation de la valeur qu'elle renvoie : affectation du résultat de la fonction à une variable, affichage du résultat de la fonction à l'écran, utilisation résultat de la fonction dans une expression ....

3.3 Blocs

Un bloc est une suite d'instructions délimitée par BEGIN END et éventuellement précédée d'une partie déclarative introduite par DECLARE . On peut donner un nom à un bloc – c'est d'ailleurs conseillé – pour faciliter la lecture d'un programme.

Un bloc sera donc

identificateur_bloc : DECLARE

déclarations ;

BEGIN

suite d'instructions ;

END *identificateur_bloc* ;

On peut donc résumer les déclarations de sous-programmes ainsi

déclaration du sous-programme IS *bloc*

remarque : un bloc peut en contenir un autre qui sera alors interne au premier.

cette notion de bloc va nous permettre de cerner la *portée* des variables – c'est-à-dire les parties du programme dans lesquelles telle ou telle variable existe – et la *visibilié* des variables – c'est-à-dire les parties du programme dans lesquelles telle ou telle variable est utilisable au moyen de son identificateur.

Règles de portée

La portée d'une variable déclarée au début d'un bloc ou d'un sous-programme va de sa déclaration jusqu'à la fin du bloc ou du sous-programme. Mais elle n'est pas visible dans un bloc interne qui redéclare une variable en utilisant le même identificateur.

exemple Le programme suivant a pour but de lire un entier n qui sera pris pour calculer le terme de rang n d'une suite donnée, puis d'afficher ce terme. La suite est définie par $u_0 = x$ et $u_n = \frac{1}{2}(u_{n-1} + \frac{x}{u_{n-1}})$. x est un réel positif qui lui aussi sera lu par le programme.

```
WITH Text_IO ; WITH Ada.Integer_Text_Io ;
USE Text_IO ; USE Ada.Integer_Text_Io ;
USE Text_IO ; USE Ada.Float_Text_Io ;
PROCEDURE suite IS
   SUBTYPE t_nat IS integer range 0 .. integer'last ;
   SUBTYPE t_float IS float digits 5 ;
   n : t_nat ; x : t_float ;
   FUNCTION U(k : t_nat ; x : t_float) RETURN t_float IS
      v : t_float := x ;
   BEGIN
      FOR i IN 1 .. k LOOP
         v := 0.5 * (v + x/v) ;
      END LOOP ;
      RETURN v ;
   END U ;
BEGIN
   PUT ("donner la valeur du rang n : ") ; GET (n) ;NEW_LINE ; - - lecture de la valeur n
   PUT ("donner la valeur du terme de rang 0 : ") ; GET (x) ;NEW_LINE ; - - lecture de la valeur x
   PUT ("U vaut ") ; PUT (U(n,x)) ;
END suite ;
```

3.4 Paquetage : une première approche

Un paquetage permet de regrouper plusieurs sous-programmes ou bien de construire un type avec ses opérations ou encore de regrouper des constantes ou les variables globales d'un programme ...

Un paquetage est composé principalement de deux parties :
- une spécification qui décrit tout ce que le paquetage fournit à ses utilisateurs éventuels, cependant une partie de ces spécifications peut être *privée* et non accessible de l'extérieur ; la partie spécification est obligatoire,
- un corps qui réalise les fonctionnalités (si fonctionnalité il y a) du paquetage annoncées dans sa spécification ; le corps d'un paquetage peut être absent si la partie accessible de la spécification du paquetage ne contient que des types ou des variables.

La déclaration d'un paquetage suit la syntaxe

PACKAGE *ident_pack* IS - - début de la spécification

déclarations D ;

PRIVATE

déclarations DP ;

END *ident_pack* ; - - fin de la spécification

PACKAGE BODY *ident_pack* IS - - début du corps

déclarations DL ;

BEGIN

suite d'instructions ;

END *ident_pack* ; - - fin du corps

où
- *D* est l'ensemble des déclarations de ce que le paquetage *ident_pack* rend accessible depuis l'extérieur
- *DP* sont des déclarations connues du paquetage seul
- *DL* sont des déclarations utiles à la réalisation des procédures ou fonctions déclarées dans la partie spécification.
- *D* et *DP* sont visibles dans le corps du paquetage

Il est important de noter que la compilation de la spécification d'un paquetage doit précéder la compilation de son corps. Il est conseillé de séparer les deux parties spécification et corps dans deux fichiers différents qui porteront les noms respectifs de *ident_pack*.ads et *ident_pack*.adb ce qui permet éventuellement de revenir sur la réalisation d'une procédure ou fonction dans le corps sans avoir à modifier ni à recompiler la spécification.

On va illustrer cette première approche de la notion de paquetage par un exemple qui se révèlera très utile au moment de l'étude des tris.

Il s'agit de réaliser l'échange des valeurs de deux variables de type **integer** ou **float** .

Pour cela, dans un fichier echange.ads, on déclare

PACKAGE echange IS

SUBTYPE t_entier IS **integer range integer'first** .. **integer'last** ;

SUBTYPE t_float IS **float digits** 5 ;

PROCEDURE ech(a,b : IN OUT t_entier) ;

```
    PROCEDURE ech(a,b : IN OUT t_float) ;
  END echange ;
```

puis dans un fichier echange.adb, on définit

```
  PACKAGE BODY echange IS
    PROCEDURE ech(a,b : IN OUT t_entier) IS
      sauv : t_entier ;
    BEGIN
      sauv := a ;
      a := b ;
      b := sauv ;
    END ech ;
    PROCEDURE ech(a,b : IN OUT t_float) IS
      sauv : t_float ;
    BEGIN
      sauv := a ;
      a := b ;
      b := sauv ;
    END ech ;
  END echange ;
```

remarque : le nom de la procédure d'échange des deux entiers est le même que le nom de la procédure d'échange de deux flottants mais la distinction entre les deux est faite par le type des paramètres formels. On a utilisé la possibilité de surcharger des sous-programmes. Pour l'utilisateur, cela permettra simplement de connaître le nom ech et de savoir que cette procédure permet l'échange des valeurs de variables de type entier ou flottant.

exemple : (suite) on peut alors écrire le programme suivant

```
  WITH Text_IO ; WITH Ada.Integer_Text_Io ;WITH Ada.Float_Text_Io ;
  USE Text_IO ; USE Ada.Integer_Text_Io ;USE Ada.Float_Text_Io ;
  WITH echange ; USE echange ;
  PROCEDURE exemple_ech IS
    n,p : t_entier ; x,y : t_float ;
  BEGIN
    PUT ("donner la valeur de n : ") ; GET (n) ;NEW_LINE ;
    PUT ("donner la valeur de p : ") ; GET (p) ; NEW_LINE ;
    ech(n,p) ; - - échange de deux entiers
    PUT ("n vaut maintenant : ") ; PUT (n) ; NEW_LINE ;
    PUT ("p vaut maintenant : ") ; PUT (p) ; NEW_LINE ;
    PUT ("donner la valeur de x : ") ; GET (x) ; NEW_LINE ;
```

```
    PUT ("donner la valeur de y : "); GET (y); NEW_LINE;
    ech(x,y); - - échange de deux flottants
    PUT ("x vaut maintenant : "); PUT (x); NEW_LINE;
    PUT ("y vaut maintenant : "); PUT (y); NEW_LINE;
END exemple_ech;
```

Chapitre 4

Type structuré

4.1 Type énuméré

Cette notion permet *d'énumérer* les valeurs possibles prises par le type que l'on veut définir.

exemple : on veut définir les couleurs d'un jeu de cartes. On peut déclarer

TYPE t_couleur IS (Pique, Coeur, Carreau, Trefle) ;

On utilise alors ce type dans les déclarations et les valeurs dans les affectations comme dans l'exemple qui suit.

exemple : (suite)

coul_dom : **constant** t_couleur := Pique ;

On peut appliquer les opérations suivantes pour un type énuméré
- l'affectation :=
- les tests d'égalité = /=
- les tests de comparaisons < <= > >=

Un type énuméré a des attributs dont voici les principaux
- first t_couleur'first est Pique
- last t_couleur'last est Trefle
- succ t_couleur'succ(Coeur) est Carreau, mais Trefle n'a pas de successeur
- pred t_couleur'pred(Coeur) est Pique, mais Pique n'a pas de prédécesseur
- val t_couleur'val(0) est Pique
- pos t_couleur'pos(Carreau) est 2

4.2 Les tableaux

Un tableau est une collection finie d'éléments de même type qui seront accessibles par leur indice dans le tableau.

Le type des éléments d'un tableau peut être quelconque et le type des indices doit être discret (**integer** , énuméré, ...). On distinguera deux types de tableaux :
- les tableaux contraints : les bornes de variation des indices sont fixées et constantes
- les tableaux non contraints : les bornes de variation des indices ont un type précis mais les bornes peuvent variées et sont précisées au moment de la déclaration de la variable de type tableau.

Dans ce chapitre nous n'utiliserons que des tableaux contraints.

déclaration et utilisation

On peut déclarer une variable de type tableau (contraint) de deux façons : soit directement en décrivant le type au moment où l'on introduit l'identificateur de la variable, soit en déclarant un type puis en utilisant ce type pour déclarer la variable.

exemple : TYPE t_vect_bool IS ARRAY (2 .. 11) OF **boolean** ;

TYPE t_vect_ent IS ARRAY (0 .. 99) OF t_entier ;

TYPE t_matrice IS ARRAY (1 .. 4 , 1 .. 3)OF t_entier ;

t1,t2 : t_vect_ent ;

t3,t4 : t_vect_ent ;

m1,m2 : t_matrice ;

v : t_vect_bool ;

t,u : ARRAY (0..99) OF t_entier ;

Attention : t1, t2, t3, t4 sont de même type mais t n'est pas du même type qu'eux bien qu'il contienne, comme eux, 100 valeurs de types t_entier. De plus, t et u ne sont toujours pas de même type car leur déclaration multiple est considérée comme une abbréviation de deux déclarations distinctes. v contient 10 booléens et m1, m2 sont des tableaux de tableaux ; ils contiennent chacun quatre tableaux de trois entiers.

Chaque élément d'un tableau est accessible par son indice de la façon suivante :

v(3) est l'élément d'indice 3 de v et c'est un booléen. m1(1)(2) est l'élément d'indice 2 du tableau m1(1) et c'est un entier.

On peut utiliser les opérations suivantes

:= l'affectation ne sera valide que si les deux variables sont de même type tableau.

<, <=, >, >= les tests d'ordre sont valides pour les tableaux **unidimensionnels** dont les éléments sont de type **discrets** ; ils correspondent alors à l'ordre lexicographique basé sur l'odre des éléments du tableau.

& la concaténation est valide pour les tableaux **unidimensionnels** ; son utilisation sera plus évidente au moment des chaînes de caractère.

Attributs

Pour les tableaux unidimensionnels, ils sont au nombre de quatre :
– first : renvoie la borne inférieure des indices du tableau
– last : renvoie la borne supérieure des indices du tableau
– length : renvoie le nombre d'éléments du tableau
– range est une forme abrégée de first .. last

Affectation

Comme toute variable, après sa déclaration une variable de type tableau est indeterminée. Sans initialisation, l'accès à ses éléments conduit donc à une erreur.

Pour donner des valeurs aux éléments d'un tableau, on peut utiliser une itération ; par exemple :

FOR i IN v'range LOOP v(i) := false END LOOP ; - - tous les éléments de v vaudront false

FOR i IN t1'range LOOP t1(i) := 0 ; END LOOP ; - - tous les éléments de t1 vaudront 0

FOR i IN m1(1)'range LOOP

FOR j IN m1(2)'range LOOP m1(i,j) := i+j ; END LOOP ;

END LOOP ;

m1 vaut alors $\begin{pmatrix} 2 & 3 & 4 \\ 3 & 4 & 5 \\ 4 & 5 & 6 \\ 5 & 6 & 7 \end{pmatrix}$.

On peut aussi utiliser une affectation ; par exemple :

t2 := t1 ; - - t1 doit être initialisé d'abord

On peut utiliser des *agrégats* ; un agrégat est un ensemble de valeurs que l'on associera soit par nom soit par position aux éléments du tableau ; par exemple :

v1 := (2|11 => false , others => true) ;

v1 vaut alors (false, true,true,true,true,true,true,true,true,false)

t2 := (0 .. 49 => 0 , 50 .. 98 => 1 , 99 => -1) ;

m2 := ((1 .. 3 => 0) , (2,3,4) , (others => -1) , (1|2 => 1, 3 => -1)) ;

m2 vaut alors $\begin{pmatrix} 0 & 0 & 0 \\ 2 & 3 & 4 \\ -1 & -1 & -1 \\ 1 & 1 & -1 \end{pmatrix}$.

On peut utiliser des *tranches* ; une tranche est un sous-tableau ; par exemple :

t3(0 .. 49) := t2(50 .. 99)

4.3 Les articles

Les tableaux permettent de regrouper sous un seul nom des valeurs d'un même type qui sont alors accessibles par leur indice ; les *articles* représenteront des collections d'objet dont les types pourront être différents et qui seront chacun accessibles par l'*identificateur* (le nom) de leur *champ*.

Déclaration et utilisation

La syntaxe est la suivante

TYPE ident_type IS RECORD

ident_champ1 : type_champ1 ;

ident_champ2 : type_champ2 ;

ident_champ3 : type_champ3 ;

...

ident_champN : type_champN ;

END RECORD ;

Le type des champs peuvent être quelconques.

exemple :

```
TYPE t_sexe IS (m,f) ;
SUBTYPE t_age IS integer range 0 .. 120 ;
SUBTYPE t_taille IS integer range 40 .. 250 ;
TYPE t_personne IS RECORD
   sexe : t_sexe ;
   age : t_age ;
   taille : t_taille ;
END RECORD ;
toto : t_personne ; nana : t_personne ;
```

On pourra manipuler toto champ par champ selon le type de chacun d'eux.

```
toto.sexe := m ; nana.sexe := f ; nana.age := 10 ;
toto.age := nana.age + 1 ;
```

ou bien globalement pour une affectation comme

```
toto := (m , 11 , 145) ;
nana := (age => 10, taille => 140, sexe => f) ;
```

4.4 Les chaînes de caractères

Une chaîne de caractères est en fait un tableau unidimensionnel de caractères. Le type prédéfini STRING permet de déclarer une chaîne de caractères.

```
TYPE t_mot IS STRING (1..5) ;
mot : t_mot ; - - mot contient 5 caractères
```

Attributs

il existe deux attributs permettant de lier le type STRING aux types **integer** et **character** :
- Image : renvoie une chaîne de caractères
- Value : renvoie la valeur de la chaîne dans le type indiqué

exemple :

Integer'Value("2002") vaut 2002

Character'Value("'a"') vaut 'a'

Integer'Image(123) vaut "123"

Character'Image('t') vaut "'t"'

Opérations

Ce sont les mêmes que pour les tableaux et en particulier on peut concaténer deux chaînes.

exemple :

```
mot := "table" ;
mot := "chaise" ; - - non valide car 6 caratères
mot := "oui" ; - - non valide car 3 caratères
mot := "ta" & "ble" ; - - mot vaut alors "table"
```

Chapitre 5

Traitement des exceptions

Lors de la compilation ou de l'exécution d'un programme, certaines instructions ne peuvent être exécutées et la compilation ou l'exécution est stoppée accompagnée d'un message d'erreur, par exemple *static expression raises "constraint_error"*. On dit qu'*une exception est levée*.

exemple : on a déclaré

TYPE t_arc_en_ciel IS (violet,indigo,bleu,vert,jaune,orange,rouge) ;

couleur : t_arc_en_ciel := rouge ; nuance : t_arc_en_ciel ;

BEGIN nuance := t_arc_en_ciel'succ(couleur) ; END ;

L'appel à t_arc_en_ciel'succ(couleur) provoque une erreur puisque dans le type t_arc_en_ciel rouge n'a pas de successeur.

Il y a différentes exceptions selon les règles du langage qui sont violées et un programme peut envisager les erreurs possibles et ainsi prendre en compte ces éventuelles exceptions.

exemple : (suite)

bloc_couleur : BEGIN nuance := t_arc_en_ciel'succ(couleur) ;

EXCEPTION WHEN CONSTRAINT_ERROR => nuance := t_arc_en_ciel'first ;

END bloc_couleur ;

...

5.1 déclarer une exception

Les principales exceptions prédéfinies sont

- CONSTRAINT_ERROR : lorsqu'on sort des bornes ou bien lorsqu'on exécute une opération arithmétique non valide comme une division par zéro
- PROGRAM_ERROR : lorsqu'on ne respecte pas une structure de contrôle comme un appel à un sous-programme non encore élaboré
- STORAGE_ERROR : lorsqu'on manque d'espace mémoire
- DATA_ERROR : lorsqu'on a lu une valeur ne correspondant pas au type déclaré ; elle est contenue dans le paquetage Text_IO
- TASKING_ERROR : cela concerne les *tâches* qui sont hors de notre propos

On peut déclarer une exception par un identificateur dans un bloc et ainsi définir la portée de cette exception.

5.2 Traitement des exceptions

Traiter une exception c'est prendre en compte et gérer cette exception là où elle peut être levée.

On peut lever une exception par une instruction :

RAISE *ident_exception* ;

Lorsqu'une exception est levée dans une unité de programme, l'exécution de cette unité est interrompue. On répond à cette exception en ajoutant des séquences d'exception à la fin de l'unité de programme en question :

```
EXCEPTION
   WHEN ident_exception1 => instructions1 ;
   WHEN ident_exception2 => instructions2 ;
   ...
   WHEN OTHERS => instructions ;
```

cette dernière traite tous les cas d'exception autres que ceux désignés au-dessus.

L'exécution de l'unité est donc abandonnée là où est levée l'exception au profit de son traitement.

remarque : une exception peut être propagée si elle n'est pas correctement traitée dans l'unité où elle a été levée ; elle devra alors être traitée dans une unité appelant la précédente.

Si une exception qui est levée n'est pas traitée alors l'exécution du programme est arrêtée avec un message d'erreur.

5.3 exemple de protection d'une saisie au clavier

Dans de nombreux exemples, on a été amené à saisir au clavier une valeur entière ou bien un flottant. Si l'utilisateur commet une erreur et frappe sur une touche caractère, l'exécution du programme est stoppée et il faut le relancer ; on peut prévoir cette erreur de saisie comme dans l'exemple suivant : les trois instructions habituelles

```
PUT ("donner un entier ") ; GET (n) ; NEW_LINE ;
```

seront remplacées par la boucle suivante

```
LOOP
   bloc :BEGIN
      PUT ("donner un entier ") ; GET (n) ; NEW_LINE ;
      EXIT ;
   EXCEPTION WHEN DATA_ERROR =>
      SKIP_LINE ;
      PUT ("ce n'est pas un entier ; recommencez") ; NEW_LINE ;
   END bloc ;
END LOOP ;
```

Dans cette boucle, l'exception DATA_ERROR peut être levée au moment de l'exécution de GET (n) si a été tapé au clavier autre chose qu'un entier : alors l'exception DATA_ERROR est traitée et EXIT

n'est pas pris en compte ; bloc se termine alors par l'affichage du message "ce n'est pas un entier ; recommencez" puis l'instruction LOOP reprend ; puisque EXIT n'a pas été exécuté, bloc est de nouveau exécuté donc une nouvelle saisie GET est faite : si elle s'achève sans lever DATA_ERROR alors EXIT est exécuté et LOOP est finie.

remarque : l'instruction SKIP_LINE de l'unité Text_Io permet de vider le *buffer* d'entrée c'est-à-dire le fichier dans lequel sont momentanément stockées les valeurs saisies au clavier.

On peut aussi envisager de limiter les bornes de la valeur à saisir de la façon suivante :

```
LOOP
   DECLARE borne : exception ;
   BEGIN
      PUT ("donner un entier entre 1 et 10 ") ; GET (n) ; NEW_LINE ;
      IF NOT (1<=n AND n<=10) THEN RAISE borne ; END IF ;
      EXIT ;
      EXCEPTION
      WHEN DATA_ERROR =>
         SKIP_LINE ;
         PUT ("ce n'est pas un entier ; recommencez") ; NEW_LINE ;
      WHEN borne =>
         SKIP_LINE ;
         PUT ("entre 1 et 10 ! recommencez") ; NEW_LINE ;
   END ;
END LOOP ;
```

Chapitre 6

Annexe A : quelques programmes

Le programme suivant lit deux entiers a et b et calcule le produit $a \times b$ en n'utilisant que des additions.

```
WITH Text_IO ; WITH Ada.Integer_Text_Io ;
USE Text_IO ; USE Ada.Integer_Text_Io ;
PROCEDURE multi_add IS
   SUBTYPE t_multipl IS integer range 0 .. 1000 ;
   s : t_multipl := 0 ; a,b : t_multipl ;
BEGIN
   PUT ("donner un entier a : ") ; GET (a) ; - - lecture de la valeur a
   NEW_LINE ;
   PUT ("donner un entier b : ") ; GET (b) ; - - lecture de la valeur b
   NEW_LINE ;
   FOR i IN 1 .. b LOOP
      s := s+a ;
   END LOOP ;
   PUT (s) ; - - écriture de la valeur finale de s
END multi_add ;
```

♣♣♣♣♣♣♣

Le programme suivant lit deux entiers a et b et calcule le quotient de a par b.

```
WITH Text_IO ; WITH Ada.Integer_Text_Io ;
USE Text_IO ; USE Ada.Integer_Text_Io ;
PROCEDURE quotient_add IS
   SUBTYPE t_dividende IS integer range 0 .. 1000 ;
   SUBTYPE t_diviseur IS integer range 1 .. 1000 ;
   s, q : t_dividende := 0 ; a : t_dividende ; b : t_diviseur ;
```

```
BEGIN
   PUT ("donner un entier a : "); GET (a);
   NEW_LINE;
   PUT ("donner un entier b non nul : "); GET (b); NEW_LINE;
   LOOP
      s := s+b;
      EXIT WHEN s>a;
      q := q+1;
   END LOOP;
PUT ("le quotient de a par b est "); PUT (q);
END quotient_add;
```

♣♣♣♣♣♣♣

Le programme suivant a pour but de lire un entier n qui sera pris pour calculer le terme de rang n d'une suite donnée, puis d'afficher ce terme. La suite est définie par $u_0 = x$ et $u_n = \frac{1}{2}(u_{n-1} + \frac{x}{u_{n-1}})$. x est un réel positif qui lui aussi sera lu par le programme. Le calcul de u_n se fait par l'intermédiaire d'une fonction à 2 paramètres pour n et x.

```
WITH Text_IO; WITH Ada.Integer_Text_Io;
USE Text_IO; USE Ada.Integer_Text_Io;
USE Text_IO; USE Ada.Float_Text_Io;
PROCEDURE suite IS
   SUBTYPE t_nat IS integer range 0 .. integer'last;
   SUBTYPE t_float IS float digits 5;
   n : t_nat; x : t_float;
   FUNCTION U(k : t_nat; x : t_float) RETURN t_float IS
      v : t_float := x;
   BEGIN
      FOR i IN 1 .. k LOOP
         v := 0.5 * (v + x/v);
      END LOOP;
      RETURN v;
   END U;
BEGIN
   PUT ("donner la valeur du rang n : "); GET (n);NEW_LINE; - - lecture de la valeur n
   PUT ("donner la valeur du terme de rang 0 : "); GET (x);NEW_LINE; - - lecture de la valeur x
   PUT ("U vaut "); PUT (U(n,x));
END suite;
```

♣♣♣♣♣♣♣

Exemple d'utilisation du paquetage tableau_unidim : on veut calculer la moyenne des éléments d'un tableau d'entiers

```
WITH Text_IO ; WITH Ada.Integer_Text_Io ;WITH Ada.Float_Text_Io ;
USE Text_IO ; USE Ada.Integer_Text_Io ;USE Ada.Float_Text_Io ;
WITH tableau_unidim ; USE tableau_unidim ;
PROCEDURE exemple_moyenne IS
  tab : t_vect_ent ;
  PROCEDURE Moyenne(v : t_vect_ent) RETURN t_float IS
      s : t_ent := 0 ;
  BEGIN
    FOR i IN v'range LOOP
      s := s+v(i) ; END LOOP ;
    RETURN (t_float(s)/t_float(v'range )) ; - - s et v'range sont ainsi transformés en flottant
  END Moyenne ;
BEGIN
  init_vect(tab) ;
  PUT ("La moyenne des elements du tableau est : ") ;PUT (Moyenne(tab),5,2,0) ;
END exemple_moyenne ;
```

♣♣♣♣♣♣♣

Exemple d'utilisation du paquetage tableau_unidim : on veut rechercher la présence d'une valeur entière dans un tableau d'entiers.

```
WITH Text_IO ; WITH Ada.Integer_Text_Io ;
USE Text_IO ; USE Ada.Integer_Text_Io ;
WITH tableau_unidim ;USE tableau_unidim ;
PROCEDURE exemple_recherche IS
  tab : t_vect_ent ; x : t_ent ;
  PROCEDURE rech(v : t_vect_ent ; x :t_ent ) RETURN boolean IS
  BEGIN
    FOR i IN v'range LOOP
      IF v(i)=x THEN RETURN (true) ; END IF ;END LOOP ;
    RETURN (false) ;
  END rech ;
BEGIN
  init_vect(tab) ;
  PUT ("entrer l'entier recherché") ; GET (x) ;NEW_LINE ;
  IF rech(tab,x) THEN PUT ("oui") ; ELSE PUT ("non") ;END IF ;
END exemple_recherche ;
```

♣♣♣♣♣♣♣

Chapitre 7

Annexe B : paquetages utilisés

Les paquetages suivants pourront être utiles et vous pourrez les modifier afin d'y ajouter d'autres fonctionnalités.

Fichier type_perso.ads

```
PACKAGE type_perso IS
SUBTYPE t_ent IS integer range -100 .. 100 ; - - on peut changer les bornes
SUBTYPE t_float IS float digits 5 ; - - on peut changer la précision
END type_perso ;
```

Fichier echange.ads

```
WITH type_perso ;USE type_perso ;
PACKAGE echange IS
  PROCEDURE ech(a,b : IN OUT t_ent) ;
  PROCEDURE ech(a,b : IN OUT t_float) ;
END echange ;
```

Fichier echange.adb

```
PACKAGE BODY echange IS
  PROCEDURE ech(a,b : IN OUT t_ent) IS
    sauv : t_ent ;
  BEGIN
    sauv := a ;
    a := b ;
    b := sauv ;
  END ech ;
  PROCEDURE ech(a,b : IN OUT t_float) IS
    sauv : t_float ;
```

```
    BEGIN
      sauv := a ;
      a := b ;
      b := sauv ;
    END ech ;
  END echange ;
```

Fichier tableau_unidim.ads

```
  WITH type_perso ;
  USE type_perso ;
  PACKAGE tableau_unidim IS
  n_max : constant t_ent := 50 ;
  SUBTYPE t_indice IS integer range 1 .. n_max ;
  TYPE t_vect_ent IS ARRAY (t_indice) OF t_ent ;
  TYPE t_vect_float IS ARRAY (t_indice) OF t_float ;
    PROCEDURE init_vect(t : OUT t_vect_ent) ;
    PROCEDURE init_vect(t : OUT t_vect_float) ;
    PROCEDURE aff_vect(t : IN t_vect_ent) ;
    PROCEDURE aff_vect(t : IN t_vect_float) ;
  END tableau_unidim ;
```

Fichier tableau_unidim.adb

```
  WITH Text_IO ; WITH Ada.Integer_Text_Io ;WITH Ada.Float_Text_Io ;
  WITH Ada.Numerics.Discrete_Random ;WITH Ada.Numerics.Float_Random ;
  USE Text_IO ; USE Ada.Integer_Text_Io ;USE Ada.Float_Text_Io ;
  PACKAGE BODY tableau_unidim IS
    PROCEDURE init_vect(t : OUT t_vect_ent) IS
    PACKAGE le_hasard IS NEW Ada.Numerics.Discrete_Random (t_ent) ; use le_hasard ;
      G : Generator ;
    BEGIN
      Reset (G) ;
      FOR i IN t'range LOOP
        t(i) := Random (G) ; - - Random (G) est un nombre engendré au hasard
                              - - dans l'intervalle défini par le type t_ent
      END LOOP ;
    END init_vect ;
```

```
  PROCEDURE init_vect(t : OUT t_vect_float) IS
  USE Ada.Numerics.Float_Random ;
    G : Generator ;
  BEGIN
    Reset (G) ;
    FOR i IN t'range LOOP
      t(i) := Random (G) * 100.0 ; - - Random (G) est un nombre engendré au hasard
                                    - - dans l'intervalle ]0, 1[
    END LOOP ;
  END init_vect ;
  PROCEDURE aff_vect(t : IN t_vect_ent) ;
  BEGIN
    FOR i IN t'range LOOP
      PUT (t(i)) ; IF i REM 5 = 0 THEN NEW_LINE ;END IF ;
    END LOOP ;
  END aff_vect ;
  PROCEDURE aff_vect(t : IN t_vect_float) ;
  BEGIN
    FOR i IN t'range LOOP
      PUT (t(i),2,2,0) ;PUT (' ') ; IF i REM 5 = 0 THEN NEW_LINE ;END IF ;
    END LOOP ;
  END aff_vect ;
END tableau_unidim ;
```

Fichier pile.adb

```
WITH type_perso ;
USE type_perso ;
PACKAGE pile IS
longueur_pile_max : constant t_ent := 1000 ;
SUBTYPE t_inter_pile IS integer range 1 .. longueur_pile_max ;
SUBTYPE t_inter_sommet IS integer range 0 .. longueur_pile_max ;
TYPE t_tab_pile IS ARRAY (t_inter_pile) OF t_ent ;
TYPE t_pile IS RECORD
  tp : t_tab_pile ;
  sp : t_inter_sommet ;
```

```
    END RECORD;
      PROCEDURE init_pile(p : OUT t_pile);
      PROCEDURE empiler(p : IN OUT t_pile; x : t_ent);
      PROCEDURE depiler(p : IN OUT t_pile);
      FUNCTION sommet(p : t_pile) RETURN t_ent;
      FUNCTION est_pile_vide(p : t_pile) RETURN boolean;
      FUNCTION est_pile_pleine(p : t_pile) RETURN boolean;
      FUNCTION longueur_pile(p : t_pile) RETURN t_inter_sommet;
    END pile;
```

Fichier pile.adb

```
    PACKAGE BODY pile IS
      PROCEDURE init_pile(p : OUT t_pile) IS
      BEGIN p.sp := 0; END init_pile;
      PROCEDURE empiler(p : IN OUT t_pile; x : t_ent) IS
      BEGIN
        IF p.sp < p.tp'last THEN p.sp := p.sp + 1; p.tp(p.sp) := x;
        ELSE null; END IF;
      END empiler;
      PROCEDURE depiler(p : IN OUT t_pile) IS
      BEGIN IF p.sp > 0 THEN p.sp := p.sp - 1; ELSE null; END IF; END depiler;
      FUNCTION sommet(p : t_pile) RETURN t_ent IS
      BEGIN RETURN p.tp(p.sp); END sommet;
      FUNCTION est_pile_vide(p : t_pile) RETURN boolean IS
      BEGIN IF p.sp = 0 THEN RETURN true; ELSE RETURN false; END IF; END est_pile_vide;
      FUNCTION est_pile_pleine(p : t_pile) RETURN boolean IS
      BEGIN IF p.sp = p.tp'last THEN RETURN true; ELSE RETURN false; END IF; END est_pile_vide;
      FUNCTION longueur_pile(p : t_pile) RETURN t_inter_sommet IS
      BEGIN RETURN p.sp; END longueur_pile;
    END pile;
```

Fichier file.ads

```
    WITH type_perso;
    USE type_perso;
    PACKAGE file IS
    longueur_file_max : constant t_ent := 1000;
```

```
SUBTYPE t_inter_file IS integer range 1 .. longueur_file_max;
TYPE t_tab_file IS ARRAY (t_inter_file) OF t_ent;
TYPE t_file IS RECORD
   tf : t_tab_file;
   prem : t_inter_file;
   der : t_inter_file;
END RECORD;
   FUNCTION plusun(i :t_inter_file) RETURN t_inter_file;
   PROCEDURE init_file(f : OUT t_file);
   FUNCTION est_file_vide(f : t_file) RETURN boolean;
   FUNCTION est_file_pleine(f : t_file) RETURN boolean;
   PROCEDURE enfiler(f : IN OUT t_file; x : t_ent);
   PROCEDURE defiler(f : IN OUT t_file);
   FUNCTION premier(f : t_file) RETURN t_ent;
   FUNCTION longueur_file(f : t_file) RETURN t_ent;
END file;
```

Fichier file.adb

```
PACKAGE BODY file IS
   FUNCTION plusun(i :t_inter_file) RETURN t_inter_file IS
   BEGIN RETURN (i REM longueur_file_max)+1; END plusun;
   PROCEDURE init_file(f : OUT t_file) IS
   BEGIN f.prem := f.tf'first; f.der := f.tf'last; END init_file;
   FUNCTION est_file_pleine(f : t_file) RETURN boolean IS
   BEGIN IF plusun(plusun(f.der))=f.prem THEN RETURN true;
      ELSE RETURN false; END IF;
   END est_file_pleine;
   FUNCTION est_file_vide(f : t_file) RETURN boolean IS
   BEGIN IF plusun(f.der)=f.prem THEN RETURN true;
      ELSE RETURN false; END IF;
   END est_file_vide;
   PROCEDURE enfiler(f : IN OUT t_file; x : t_ent) IS
   BEGIN
      IF plusun(plusun(f.der))=f.prem THEN NULL;
      ELSE f.der := plusun(f.der); f.tf(f.der) := x;
      END IF;
   END enfiler;
   PROCEDURE defiler(f : IN OUT t_file) IS
```

```
    BEGIN
      IF plusun(f.der)=f.prem THEN NULL ;
      ELSE f.prem := plusun(f.prem) ; END IF ;
    END defiler ;
    FUNCTION premier(f : t_file) RETURN t_ent IS
    BEGIN RETURN f.tf(f.prem) ; END premier ;
    FUNCTION longueur_file(f : t_file) RETURN t_ent IS
    BEGIN RETURN (f.prem - f.der + 1)MOD f.tf'last ; END longueur_file ;
  END file ;
```

Chapitre 8

Annexe C : Mots réservés

Cette liste contient les mots qui ne peuvent pas être utilisés comme identificateurs

abort	abs	abstract	accept	access	aliased	all
and	array	at				
begin	body					
case	constant					
declare	delay	delta	digits	do		
else	elsif	end	entry	exception	exit	
for	function					
generic	goto					
if	in	is				
limited	loop					
mod						
new	not	null				
of	or	others	out			
package	pragma	private	procedure	protected		
raise	range	record	rem	rename	requeue	return
reverse						
select	separate	subtype				
tagged	task	terminate	then	type		
until	use					
when	while	with				
xor						