Sommaire

Chapitre 1 : Notions fondamentales, représentation d'un algorithme 3
Notion d'algorithme et de programme 3
Résolution d'un problème 3
Les arbres programmatiques.. 3
Représentation graphique des répétitives et alternatives 3
Traitement des exceptions dans les arbres programmatiques.. 4
Conception des algorithmes en informatique 4
Environnement fonctionnel :.. 4
Caractéristiques des objets :.. 4
Conclusion 5
Chapitre 2 : Notion de paquetage en ADA 6
Introduction.. 6
Les paquetages en ADA.. 6
Forme d'un paquetage principal en ADA 6
Exemple de procédure/programme : 6
Utilisation du paquetage PFiPerSimp.. 6
Utilisation du paquetage PEntier 6
La notion de type.. 6
La notion de procédure 7
Notion de paramètre réel et formel - Mode de transmission des paramètres 7
Chapitre 3 : Glossaire et programme 8
Glossaire : 8
Arbre Programmatique : 8
Programme ADA : 8
Chapitre 4 : Etude du paquetage Pchaine, Complément sur les types 10
Type paramétré en ADA 10
Notion de type privé 10
Types paramétrés :... 10
Types fonctionnels ou abstraits... 10
Différence entre un type privé ou limité privé 10
Rappel sur les sous-types.. 10
Les types dérivés... 11
Le type character de PCaractere 12
Ensemble de valeur : 12
Opérations : 12
Chapitre 5 : PentGen, la généricité en ADA.. 13
Introduction... 13
Déclaration des constantes 13
Déclaration d'une unité générique 13
Chapitre 6 : Le paquetage PenumGen, les types énumérés, le paquetage PDecimal 15
Le type énumératif (ou type énuméré) 15
Définition 15
Exemples 15
Règles : 15
Opérations sur les types énumérés 15
Entrée/Sortie des objets d'un type énuméré 15
Le paquetage PDecimal 15
Chapitre 7 : Les sous-programmes, les paquetages en ADA 17
Les sous-programmes 17
Introduction 17
La spécification des sous-programmes 17
Le corps des sous-programmes 18
Exécution d'un sous-programme :.. 18
Glossaire d'un sous-programme :... 18

Les paquetages ... 22
Introduction ... 22
La spécification d'un paquetage ... 23
Le corps du paquetage ... 24
Chapitre 8 : Les fichiers Texte ... 25
Définition : ... 25
Chapitre 9 : Le paquetage PFiTex ... 26
Programme : ... 26
Chapitre 10 : Les tableaux ... 27
Notion de tableau ... 27
Types de tableau ... 27
Exemple 1 : ... 27
Exemple 2 : ... 27
Exemple 3 : ... 27
Exemple 4 : ... 27
Opération sur les tableaux : ... 27
Exemple 1 : ... 27
Les attributs : ... 27
Les tranches : ... 28
Remarque : ... 28
Les agrégats : ... 28
Initialisation d'un tableau ... 28
Instructions ... 28
La concaténation : ... 28
Utilisation de tableaux comme paramètres de procédures ou de fonctions : ... 29
L'instruction FOR ... 29
Exemple : ... 29
Remarques : ... 29
Reprise de la fonction Somme dans une boucle `for` : ... 29
Chapitre 11 : Le tri à bulle ... 30
Explications : ... 30
Arbre programmatique : ... 30
Programme ADA : ... 31
Chapitre 12 : Les fichiers de Records ... 32
Introduction ... 32
PFiSeqGen : ... 32
Exercice ... 32
L'accès direct ... 32
Exercice ... 32
L'instruction Case ... 33
Exercice : ... 33

Chapitre 1 : Notions fondamentales, représentation d'un algorithme

Notion d'algorithme et de programme

Programmer : Rédiger des ordres dans un langage assimilable par l'ordinateur, ce qui permettra une compilation(traduction par la machine)

Programme : Un ensemble fini et ordonné d'actions pour effectuer un travail. Les actions sont spécifiées dans un langage que comprend l'exécutant.

En informatique, l'exécutant est un ordinateur.

Les instructions sont données dans un langage dit « langage de programmation ».

Pour rédiger un programme, il va falloir en inventer le mécanisme (choix des différentes actions/traitements).

Le mécanisme dépend uniquement du problème que l'on a à traiter, et non pas du langage dans lequel il a été exprimé (notion d'algorithme).

Algorithme : Une suite finie d'opérations élémentaires constituant un schéma de calcul ou de résolution d'un problème.

Résolution d'un problème

- On a un problème à traiter.
- On construit l'algorithme.
- On valide l'algorithme en le testant au moyen de jeux d'essais.
- On traduit l'algorithme dans un langage de programmation.

Organigramme: Schéma représentant graphiquement un algorithme.

Les arbres programmatiques

Définition: Un arbre programmatique est un schéma qui met en évidence la structure d'un algorithme. Il est constitué d'une racine et de branches aboutissant à des feuilles rondes ou rectangulaires.

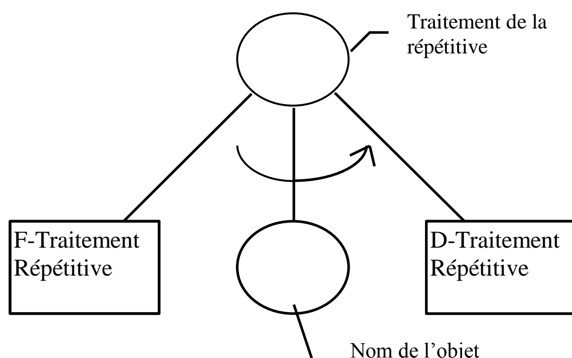
Une feuille ronde	->	Un noeud
Une feuille rectangulaire	->	Une feuille terminale

Le rond d'une répétitive doit porter le nom de l'objet traité dans la répétitive.

Quoi	->	Niveau 0 (pas de détails)
Comment	->	Niveau 1 à n (de plus en plus de détails)

Tout algorithme est construit à partir de séquences de répétitive et d'alternatives

Représentation graphique des répétitives et alternatives



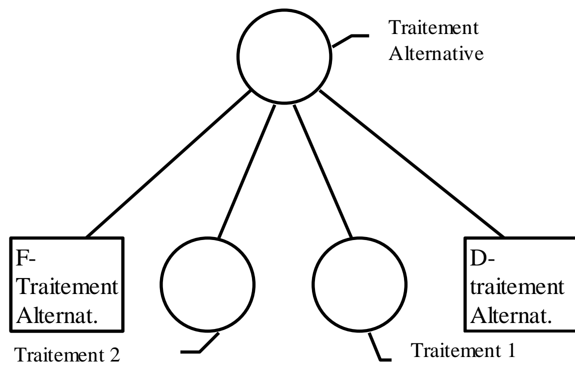
Une Répétitive

- On exécute une fois et une seule la feuille de droite (D-).
- On examine la condition C :

Si la condition C est vraie : On exécute une fois et une seule la feuille de gauche (F-)

Si la condition C est fausse: On exécute le contenu de « *Nom de l'objet* ».

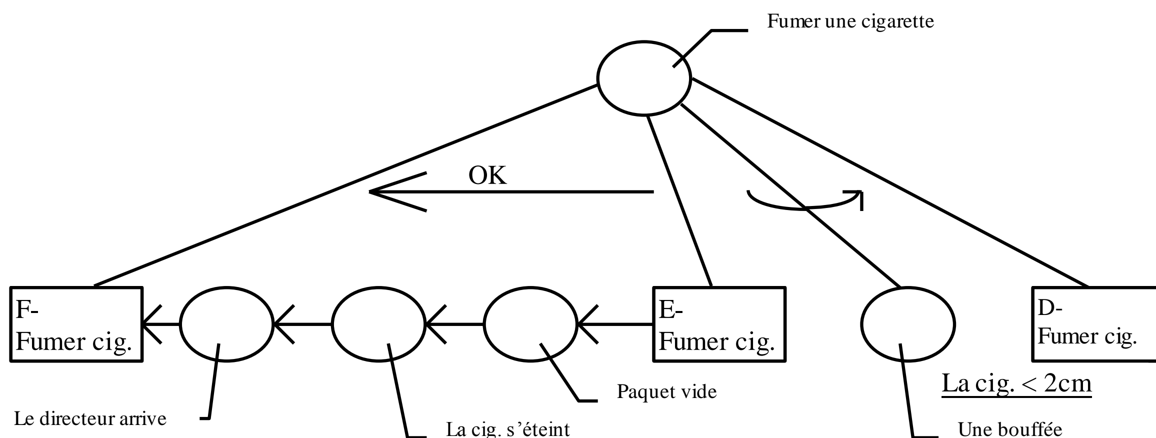
On réexamine la condition C.



Une Alternative

- On exécute une fois et une seule la feuille de droite (D-)
- On examine la condition C
  Si la condition C est vraie : On exécute « *Traitement 1* ».
  Si la condition C est fausse: On exécute « *Traitement 2* ».
- On exécute une fois et une seule la feuille de gauche (F-)

Traitement des exceptions dans les arbres programmatiques



On ne peut traiter des exceptions qu'au premier niveau de l'arbre (aussi compliqué que soit l'arbre).

Dés qu'une exception est levée, le traitement est interrompu après avoir exécuté ce qui est prévu dans le traitement de l'exception qui a été levée.

Conception des algorithmes en informatique

La description d'un algorithme comprend :

- La description des actions est représentée par un arbre programmatique
- La description de l'environnement fonctionnel (c'est-à-dire la description de l'ensemble des objets auxquels font références les actions).

Environnement fonctionnel :

On distingue 3 types d'objets :

- Les objets d'entrée (donnée du problème)
- Les objets de sortie (résultats produits par l'algorithme)
- Les objets internes ou de travail (objets de manoeuvres ou de résultats intermédiaires)

Glossaire : Définir les objets d'entrée, de sortie, intermédiaires.

1. Données du problème (Objets d'entrée)
2. Résultats recherchés (Objets de sortie)
3. Principe
4. Variables de travail (Objets de travail)

Ex : Résolution d'équations du 2ème degré.

Objets d'entrée :	A, B, C :	Coefficients de l'équation
Objets de sortie:	X1, X2 :	Racines de l'équation
Objets de travail :	Delta :	Discriminant

Caractéristiques des objets :

Pour définir un objet, il faut préciser :

- Son nom (son identificateur)
- Son utilisation (objet d 'entrée, objet de sortie, ...)
- Son sens (sa signification)
- Sa nature (« donnée », « valeur », « algorithme »)
- Son type (Caractérise les valeurs que peut prendre cet objet, et aussi les actions et opérations qu'on peut effectuer sur cet objet).

Une *donnée* peut être une *constante* ou une *variable*.

- Une *constante* est une valeur compatible avec le type de la constante. Cette valeur est définie une fois pour toutes et ne peut plus être modifiée.
- Une *variable* peut prendre n'importe quelle valeur compatible avec son type. Cette valeur peut changer pendant l'exécution de l'algorithme ou selon le moment d'exploitation de l'algorithme.

Un *algorithme* peut être soit une procédure, soit une fonction.

- Une *procédure* est un algorithme qui, à partir d'objets d'entrée (en général), produit un ou plusieurs objets de sortie. Une procédure **fait** quelque chose.
- Une *fonction* est un algorithme qui, à partir d'objets d'entrée, produit un seul résultat. Une fonction **vaut** quelque chose.

Conclusion

Quand on a à résoudre un problème :

- On recherche de quelles données on dispose.
- Quel résultat cherche-t-on à obtenir ?

Chapitre 2 : Notion de paquetage en ADA

Introduction

Les outils que l'on va utiliser en ADA sont fournis dans des paquetages.

Les paquetages en ADA

- Dans un paquetage, il y a des outils mis à la disposition du programmeur.
- Un paquetage est composé de 2 parties :

La spécification : Quoi
Le corps (body) : Comment

- Le paquetage est une unité fondamentale de compilation.
- La spécification contient uniquement les informations nécessaires communes à l'utilisateur et au créateur du paquetage (C'est la partie visible du paquetage : ce que l'utilisateur doit savoir pour l'utiliser).

Forme d'un paquetage principal en ADA

Un programme principal est une procédure (en ADA). Toute procédure sans paramètres et de niveau 0 est susceptible d'être un programme principal.

Exemple de procédure/programme :

```
procedure PgQuiNeFaitRien is
begin
  null;
end PgQuiNeFaitRien;
```

Utilisation du paquetage PFiPerSimp

Enoncé : Rédiger un programme qui permet de créer un fichier de personne et qui en affiche le contenu à l'écran.

```
with PFiPerSimp;        -- On a le droit de travailler
                        -- avec le paquetage PfiPerSimp
procedure CreAff is
begin
PFiPerSimp.CreerFichier;
  PFiPerSimp.LireFichier;
end CreAff;
```

On dit que le « with » sert à l'importation d'une unité de bibliothèque dans une unité de compilation. Une unité de bibliothèque peut contenir plusieurs unités de compilation.

On utilise une notation pointée pour indiquer qu'on utilise la procédure CreerFichier du paquetage PFiPerSimp. Le fait qu'une unité soit importée ne suffit pas à assurer que les éléments en sont directement visibles. Il faut rendre visible les éléments des unités importées, d'où la notation pointée qui assure la visibilité par rapport à la sélection.

Utilisation du paquetage PEntier

La notion de type

ADA est un langage fortement « typé ». C'est à dire que pour chaque variable manipulée, il faut spécifier les propriétés, les opérations, le domaine de variation que peuvent prendre ces valeurs. En outre, une fois typée, une variable ne peut plus changer de type. Hormis par un transtypage. Cette opération de transtypage sera faite avant le corps du programme.

En ADA, on distingue la définition d'un type et la déclaration d'un type.

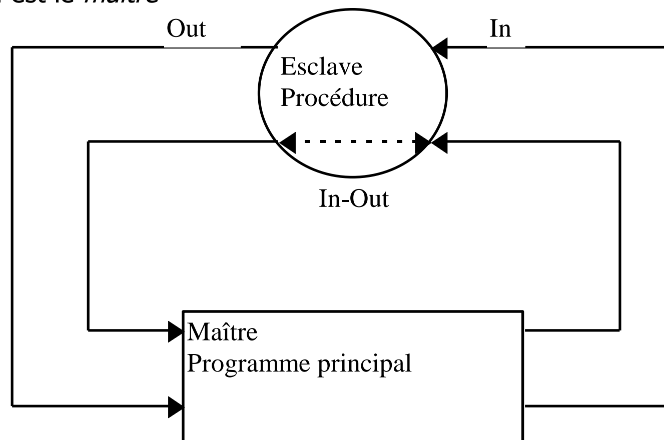
- Définir un type, c'est le décrire. C'est à dire expliquer comment il est fait.
- Déclarer un type, c'est lui donner un nom. En ADA, c'est le nom du type qui définit un type, et non sa définition.

Un type qui n'a pas de nom est un type « anonyme ».

ADA est un langage fortement typé en ce sens que 2 objets sont du même type si et seulement s'ils sont déclarés avec le même nom de type.

La notion de procédure

Une procédure est un outil (*esclave*)
Un programme principal est le *maître*



Notion de paramètre réel et formel - Mode de transmission des paramètres

Une procédure a 3 sortes de paramètres :

- Entrant
- Sortant
- Entrant-Sortant

Il y a donc en ADA, 3 modes de transmission des paramètres : In, Out, In-Out.

Chapitre 3 : Glossaire et programme

Enoncé : Rédiger en ADA un programme qui permet de saisir une suite de nombres entiers. On s'arrête à la frappe du 0, et qui, pour chacun d'eux affiche son double, son prédécesseur si le nombre est inférieur à 0 ou son successeur si le nombre est supérieur à 0.

Glossaire :

Données du problème :

LeNb est une variable numérique de type PEntier.TEntier qui recevra l'une des valeurs saisies au clavier par l'utilisateur.

Résultats recherchés :

SonDouble est une variable numérique de type PEntier.TEntier qui recevra le double de LeNb

NbAvant, *NbApres* : sont des variables numériques de type PEntier.TEntier qui recevront respectivement le nombre qui précède LeNb si celui-ci est inférieur à 0 ou qui succède LeNb si celui-ci est supérieur à 0.

Principe :

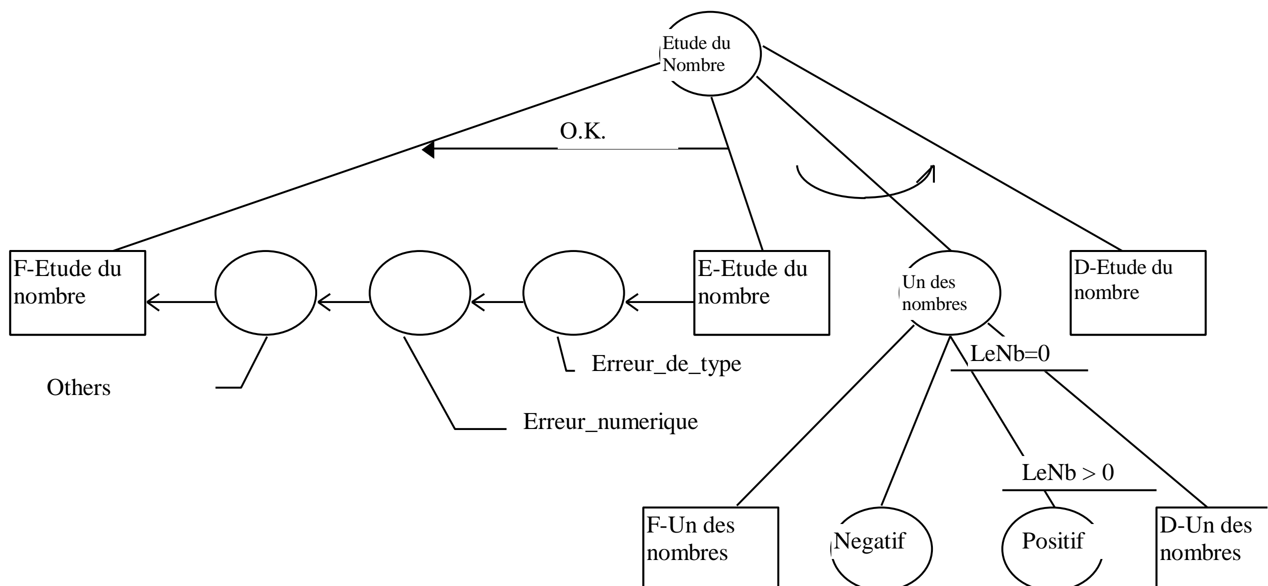
On saisit une suite de nombre jusqu'à la frappe du nombre 0.

Pour chacun des nombres saisis, on évalue son double, puis on regardera si le nombre et supérieur à 0 ou inférieur à 0. Suivant le cas, on affichera le successeur ou le prédécesseur.

Variables de travail :

Néant

Arbre Programmatique :



Programme ADA :

```
with PEntier;
with PChaine;
procedure EtudeDeNombre is
  LeNb : PEntier.TEntier;
  SonDouble : PEntier.TEntier;
  NbAvant : PEntier.TEntier;
  NbApres : PEntier.TEntier;
-- renommage des fonctions +,-,*,>,=
function "=" (OpG, OpD : in PEntier.TEntier)
  return boolean
  renames PEntier."=";
-- Attention, l'opérateur /= est visible quand l'opérateur = est visible
```

```
function "+" (OpG, OpD : in PEntier.TEntier)
  return PEntier.TEntier
  renames PEntier."+";

begin
  PChaine.put("Tapez un entier (0 pour finir)  : );
  PEntier.get_line(LEntier=>LeNb);
  while not(LeNb=0)
  loop
    SonDouble := LeNb * 2;
    PChaine.put("LeNombre ");
    PEntier.put(Lentier => LeNb);
    PChaine.put_line(" a : ");
    PChaine.put(" - Pour double : ");
    PEntier.put_line(Lentier => SonDouble);
    if LeNb>0
    then
      NbApres := LeNb+1;
      -- NbApres := PEntier.Succ (LEntier=>LeNb);
      -- pas de renommage du +
      PChaine.put("- Pour successeur : ");
      PEntier.put_line(Lentier => NbApres);
    else
      NbAvant := LeNb-1;
      -- NbAvant := PEntier.Pred(LEntier=>LeNb);
      -- pas de renommage du -
      PChaine.put("- pour prédécesseur : ");
      PEntier.put_line(LEntier=>NbAvant);
    end if;
    PChaine.put("Nombre suivant ? (0 pour finir) : ");
    PEntier.get_line(LEntier=>LeNb);
  end loop;

exception
-- Traitement des exceptions
  -- (1)
  when PEntier.ERREUR_DE_TYPE =>
    PChaine.put_line("Nombre en dehors de l'intervalle ou mauvaise saisie");
  -- (2)
  when PEntier.ERREUR_NUMERIQUE =>
    PChaine.put_line("Le calcul fait sortir de l'intervalle");
  -- (3)
  when others =>
    PChaine.put_line("Caracteres interdits en ADA ");

end EtudeDeNombre;
```

Chapitre 4 : Etude du paquetage Pchaine, Complément sur les types

Type paramétré en ADA

Notion de type privé

Un type implique des valeurs admissibles pour les variables déclarées de ce type.(Opérations permises sur ces variables).

Types paramétrés :

Un tableau est un type structuré.
Types structurés : Tableau, enregistrement, par opposition aux types simples (scalaires) : types discrets et numériques.
Types discrets : type dont on peut énumérer les valeurs.
<u>Ex :</u> Le type boolean
<u>Contre ex :</u> les réels

Dans les *types structurés*, on peut oublier des limites à ces types (Ex : tableau sans bornes). Les limites laissées libres sont nommées des "paramètres". D'où la notion de types paramétrés.

<u>Conclusion :</u> Un type paramétré est un modèle de type permettant de déclarer d'autres types. Si on veut utiliser une variable de type paramètré, il va falloir tout d'abord donner une valeur au paramètre. C'est à dire qu'il va falloir contraindre le type paramètré. Cette opération s'effectue en déclarant un sous-type contraint du type paramètré.
En ADA, les types paramètrés sont généralement appelés des types non-contraints.

Types fonctionnels ou abstraits

On parle d'un type fonctionnel lorsque sa description n'indique que les opérations admissibles pour ce type et laisse de côté sa représentation réelle.
On interdit donc toutes les opérations où interviennent explicitement la connaissance de la structure réelle du type.
En ADA, un type fonctionnel ou abstrait s'appelle un *type privé* (private) ou *type limité privé* (limited private).
On dit qu'un type est instanciable lorsqu'on peut déclarer une variable de ce type.

- Un type privé ou limité privé est instanciable.
- Un type non-contraint n'est pas instanciable.
- Un type contraint d'un type paramétré est instanciable.

Différence entre un type privé ou limité privé

type privé	**type limité privé**
• Opérations définies dans le paquetage • Opérations d'affectations := • Opérations de comparaisons =, /=	• Opérations définies dans le paquetage

- Les objets de types privés peuvent être déclarés, passés en paramètre et affectés ou comparés.
- Les objets de types limités privés ne peuvent être que déclarés ou transmis en paramètres.

<u>Les types dérivés</u>

<u>Ex :</u> TNaturel, TPositif

Rappel sur les sous-types

Un sous-type caractérise un ensemble de valeurs qui sont seulement un sous-ensemble d'un type connu sous le nom de "type de base".
Pour définir un sous-type, il faut :

- un type
- une contrainte.

Il n'y a aucun moyen de restreindre l'ensemble des opérations du type de base. (Les restrictions ne s'appliquent qu'aux valeurs)
<u>Ex :</u>

```
Subtype TCentimes is PEntier.TEntier range 0..99;
```

- Une déclaration de sous-type ne crée pas de nouveau type.
- Un sous-type hérite de toutes les opérations d'Entrée/Sortie de son type de base.

Ex :

```
Subtype TNumeroDeJour in PEntier.TEntier range 1..31;
NumJour : TNumeroDeJour;
InterJour : PEntier.Tentier ;
```

La frappe du nombre 50 provoque la levée d'une exception ⇒ CONSTRAINT_ERROR (prédéfinie par ADA).

Les exceptions du paquetage PEntier n'héritent pas de contraintes sur les valeurs imposées par le sous-type.

`NumJour := InterJour;` → OUI, si InterJour $\subset[1,31]$

`InterJour := NumJour;` → OUI, toujours

On peut dire qu'un sous-type n'introduit pas un nouveau type mais est plutôt une façon commode de désigner un type existant muni d'une contrainte.

L'utilisation d'un sous-type peut garantir que des erreurs soient détectées plus tôt en empêchant les affectations de valeurs inappropriées à des valeurs.

Les types dérivés

Ex :

```
type TNumCar is new PEntier.TEntier;
```

On va utiliser des types de données logiquement différentes :

```
type TMasse is new PEntier.TEntier range 0..100;
type TLongueur is new PEntier.TEntier range 0..100;

subtype TMasse is PEntier.TEntier range 0..100;
subtype TLongueur is PEntier.TEntier range 0..100;
```

Un type dérivé hérite des littéraux (constante) et des opérations déjà définis pour son type parent. Ces littéraux et opérations sont surchargés et leur identité doit être déduite du contexte. Mais les opérandes des types de base et dérivés ne peuvent pas être mélangés.

subtype TNumeroDeJour is PEntier.TEntier range 1..31;
type TNum is new PEntier.TEntier range 1..31;

```
NumJour: TNum;
Jour1, Jour2, JourDif : TNum;
InterJour : PEntier.TEntier;
Dif : PEntier.TEntier;
DerNum : TNum;
```

Ex :

```
Dif := NumJour - InterJour;
```

Il faut effectuer un transtypage

```
Dif := PEntier.TEntier(NumJour) - InterJour;
```

Correct à condition d'avoir renommé l'opérateur "-" de PEntier.TEntier

Ex :

```
DerNum := NumJour - InterJour;
TNum      TNum      type incompatible
```

Il faut effectuer un transtypage

```
DerNum := NumJour - TNum (InterJour)
```

Ex :

```
PChaine.put("Donnez le jour de l'année : ");
get_line(LEntier => Jour1);
```

Pas besoin de pointer par PEntier (get_line directement visible)

Le type character de PCaractere

Ensemble de valeur :

Le caractère a ⇒ `'a'`
Le caractère A ⇒ `'A'`
Le caractère . ⇒ `'.'`
Le caractère espace ⇒ `' '`

Opérations :

- Affectation ':='

Ex :
```
Carac : character;
...
Carac := '$';
```

- Comparaison =, <, <=, >, >=, /=

```
' '<'0'<'1'<'2'<..<'9'<..<'A'<'B'<'C'<..<'Z'<'a'<'b'<'c'<..<'z'
```

Type discret (fini et ordonné)

- fonction Succ, Pred

Ex :
```
Carac := 'D';
Carac := Succ(Carac);
```

- procedure get_line

```
procedure get_line(LeCar : out character);
PCaractere.get_line(LeCar => Carac);
```

Chapitre 5 : PentGen, la généricité en ADA

Introduction

- *Générique :* Du même genre, de la même famille.
- PEntGen permet de traiter des objets de la même famille que PEntier.
- Programmer générique, c'est programmer « en blanc » en pensant à la réutilisabilité. En effet, il est fréquent que dans plusieurs programmes on ait des morceaux de programmes identiques mais s'appliquant à des types différents.

Exemple 1 : *Echange de 2 valeurs, parties identiques pour PEntier, PDecimal, boolean. Il est donc inutile d'écrire 3 procédures différentes.*

Exemple 2 : *Classer des noms par ordre alphabétique. Classer des nombres entiers par ordre croissant. Classer des nombres décimaux par ordre croissant. La logique de l'algorithme est indépendante du type de données à classer (Algorithme générique)*

D'où la création de procédures et de paquetages génériques.

On ne peut pas utiliser directement une procédure ou un paquetage générique. Par contre, à partir de la procédure générique ou du paquetage générique, on va créer une véritable procédure ou un véritable paquetage en instanciant la procédure ou le paquetage générique.

Déclaration des constantes

Ex :
```
DOUZE : constant PEntier.TEntier := 12;
```

Déclaration d'une unité générique

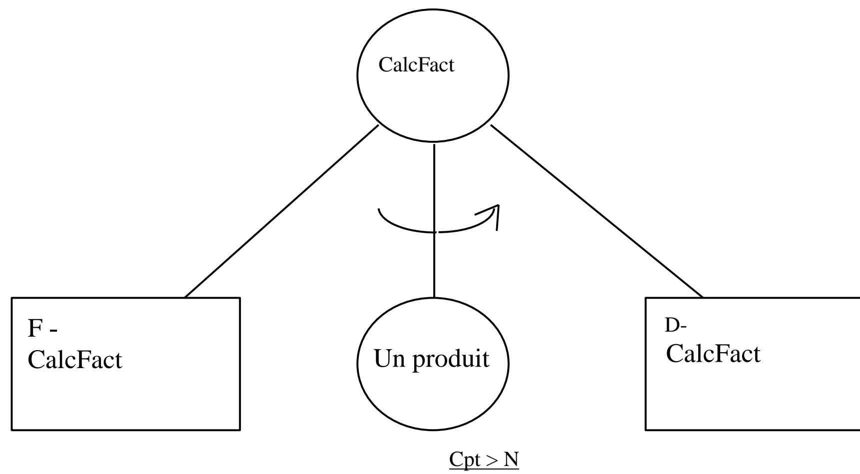
Une unité générique est composée de 3 parties :

1. La déclaration de paramètres génériques, qui est introduite par le mot-clé generic, suivi des paramètres de la généricité :
   - des types
   - des constantes ou variables
   - des sous-programmes
2. La spécification de l'unité générique, qui est semblable à celle d'une unité non-générique.
3. Le corps de l'unité générique .

Instancier le paquetage PEntGen revient à donner une valeur au paramètre générique. Ce qui donne un paquetage utilisable avec tous les outils.

Exemple d'instanciation du paquetage PEntGen :
```
package PEnt5 is new PEntGen(NbDeChiffres =>5);
```

Ex : Calcul de factorielle n pour $1 \leq n \leq 10$




```
with PChaine;
with PEntier;
with PEntGen;
procedure CalcFact is
  N : PEntier.TPositif;
  -- on instancie le paquetage PEntGen
  package PEnt7 is new PEntGen(NbDeNombres =>7);
  Fact : PEnt7.TPositif;
  Cpt : PEntier.TPositif:
  -- renommage des opérateurs * de PEnt7, > de PEntier
function "*"(OpG, OpD : in  PEnt7.TEntier)
  return PEnt7.TEntier
  renames PEnt7."*";
function ">"(OpG, OpD : in  PEntier.TEntier)
  return boolean
  renames PEntier.">";

begin
  PChaine.put("Tapez un entier entre 1 et 10 : ");
  PEntier.get_line(Lentier => N);
  Fact := 1;
  Cpt := 2;
  while not(Cpt>N)
  loop
    Fact := Fact * PEnt7.TPositif(Cpt);
    Cpt := Cpt+1;
  end loop;
  PEntier.put(Lentier => N);
  PChaine.put("! = ");
  PEnt7.put_line(Lentier => Fact);
end CalcFact;
```

Chapitre 6 : Le paquetage PenumGen, les types énumérés, le paquetage PDecimal

Le type énumératif (ou type énuméré)

Définition

Un type énumératif est déclaré en indiquant la liste des variables symboliques que peuvent prendre les objets de ce type. Dans un type énumératif, on peut trouver des identificateurs ou des littéraux.

Exemples

type TJour is (LUNDI, MARDI, MERCREDI, JEUDI, VENDREDI, SAMEDI, DIMANCHE);
type THexa is ('0','1',...,'9',A,B,C,...,F);
type TOperateur is ('+' , '-' , '*' , '/');
type TCouleur is (bleu, rouge, vert, noir, blanc, orange, marron);
type TFeux is (vert, orange, rouge);

```
Milieu : TJour;
ChiffreEnHexa : THexa;
LOp : TOperateur;
Demain := Mercredi;
Milieu := JEUDI; -- <=> Milieu := jeudi;
ChiffreEnHexa := A;
LOp := '+';
```

Règles :

- L'opérateur d'affectation est légal pour un type énuméré.
- Un même symbole peut appartenir à plusieurs types énumérés.
- Quand un opérateur à plusieurs sens, il y a surcharge du symbole.

Pour distingué de quel opérateur il s'agit : `Le_Type'(Le_Nom)`

Ex :
```
TCouleur'(rouge)
TFeux'(rouge)
```

Il résulte de la définition d'un type énuméré l'existence d'un ordre entre les valeurs du type.
Sur un type énuméré, on a droit aux opérateurs de comparaison.
Ex :
```
if rouge<vert -- ambiguïté)
then
```

Opérations sur les types énumérés

L'ensemble des opérations applicables à un type énuméré est constitué de :

- L'affectation `:=`
- L'égalité `=`, `/=`
- La comparaison `<, <=, >, >=`

On peut en plus appliquer les attributs `'last, 'first, 'succ, 'pred, 'pos, 'val` à un type énuméré
Ex :
```
Nom_du_type 'succ (...)
```

Entrée/Sortie des objets d'un type énuméré

Lors de la saisie ou de l'affichage, on doit instancier le paquetage générique PEnumGen.

Le paquetage PDecimal

La déclaration du type TDecimal comprend :

- l'indication de l'erreur absolue tolérable dans la représentation (donnée par le Delta).
- la contrainte de l'intervalle obligatoire.

Le Delta est du type point fixe, et s'écrit : ± (partie_entière).(partie_décimale). Cette représentation est dite à virgule fixe. Toute variable du type TDecimal a une précision d'au moins ± 0.001 comprise entre -100000 et 100000.

Chapitre 7 : Les sous-programmes, les paquetages en ADA

Les sous-programmes

Introduction

Quand on analyse un problème et qu'on construit un arbre programmatique, on construit le QUOI en vue du COMMENT. La construction du COMMENT se fait en allant du général vers le détail (méthode de l'analyse descendante).

Un problème résolu suivant la méthode de l'analyse descendante est décomposé en sous-problèmes. En ADA, la décomposition d'un programme se fera en associant un sous-programme à un sous-problème.

On structure l'application en sous-programmes. Cette structuration offre 2 avantages :

- Le sous-programme correspond à un sous-problème bien défini. Donc, on pourra l'écrire séparément, et au besoin, le diviser lui-même en sous-programmes.
- Ce que réalise un sous-programme peut être utilisé en différents points d'une même application. Il suffit de demander l'exécution du sous-programme en appelant les sous-programmes.

Un sous-programme peut être :

- Compilable en ADA
- Utilisé dans d'autres programmes
- Regroupé avec d'autres sous-programmes pour faire un paquetage.

En ADA, il existe 2 sortes de sous-programmes :

- Les procédures (Fait)
- Les fonctions (Vaut)

Une procédure définie une instruction de haut niveau et peut avoir des paramètres qui permettent de communiquer avec l'extérieur. Une instruction spécifique permet d'appeler une procédure.

Ex : `Nom_du_paquetage.Nom_de_la_procédure`

Ceci avec le remplacement des paramètres formels par les paramètres effectifs.

Une fonction vaut, donc retourne une valeur. Elle permet de calculer une valeur du type de la fonction. Une fonction utilise des paramètres qui ne peuvent pas être modifiés (transmis en mode `in`). L'appel d'une fonction ne constitue pas une instruction et à lieu dans l'évaluation d'une instruction.

Un sous-programme, comme toute entité du langage, est introduit par une déclaration qui apparaît de la partie déclarative de l'environnement d'appel.

La déclaration d'un sous-programme peut se faire en 2 étapes :

- La déclaration de la spécification du sous-programme (c'est à dire de son entête)
- La déclaration du corps du sous-programme

La spécification des sous-programmes

Une spécification de sous-programme constitue l'interface entre le sous-programme et les unités utilisatrices de ce sous-programme.

Une spécification de sous-programme précise ce qu'il est nécessaire de savoir pour appeler le sous-programme. Mais elle n'est pas toujours suffisante car elle ne dit rien ou presque rien sur ce que fait le sous-programme. Il est utile de la compléter par un commentaire.

Le nom d'un sous-programme étant introduit par sa spécification, ce sous-programme est maintenant connu par son nom et il peut être utilisé. Le nom de la fonction peut être celui d'un opérateur surchargeable du langage (+,-,*,/,>,>=,...)

La spécification d'un sous-programme contient le plus souvent une liste de déclarations de paramètres formels dont la séparation est le ':' .

Pour chaque paramètre, on indique : `Nom : Mode_de_passage Type [:= valeur_par_défaut]`

Règle :

- Un paramètre en mode *IN* doit être considéré comme une constante à l'intérieur du sous-programme.

Le paramètre formel transmis en mode *IN* se contente de recevoir la valeur du paramètre réel. On doit donc le considérer comme une constante. Par conséquent, il est interdit de lui modifier sa valeur et donc il est interdit de mettre dans le corps du sous-programme ce paramètre à gauche d'un " := ", pas plus que comme paramètre effectif correspondant au paramètre de DansLaChaine lors de l'appel de PChaine.Copier.

- Un paramètre en mode *OUT* correspond à un <u>résultat</u> qui va être restitué au programme appelant. Un paramètre en mode *OUT* ne peut que recevoir une valeur. On ne peut rien faire d'autre avec lui. Il est donc impossible de le trouver à droite d'un ":=", pas plus que dans l'expression d'une condition, ou comme paramètre correspondant au paramètre LaChaine de PChaine.Copier.
- Un paramètre en mode *IN-OUT* peut être utilisé comme une <u>variable</u> ordinaire. Il arrive avec une valeur et va être modifié par l'algorithme du sous-programme.

<u>Remarque :</u>
La spécification d'un sous-programme *fonction* doit obligatoirement indiquer le type de la valeur restituée lors de l'appel de la fonction. Le type de la valeur se met après le mot "`return`" de la fonction.

Le corps des sous-programmes

<u>Ex :</u>
Rédiger une procédure SaisirPhrase qui permet de saisir une chaîne de caractères LaPhrase d'au plus 100 caractères.
Rédiger une procédure AfficherPhrase qui permet d'afficher une chaîne de caractères LaPhrase d'au plus 100 caractères.
Rédiger une fonction NbDeBlancs qui fournie le nombre d'espaces figurant dans une chaîne de caractères LaPhrase d'au plus 100 caractères.
Rédiger une procédure CompterLesBlancs qui permet de compter le nombre d'espaces dans une chaîne donnée LaPhrase d'au plus 100 caractères.

```
Subtype TCh100 is PChaine.TChaine(NbMaxCar =>100);
procedure SaisirPhrase(LaPhrase : out TCh100);
procedure AfficherPhrase (LaPhrase : in TCh100);
function NombreDeBlancs (LaPhrase : in TCh100)
  return PEntier.TNaturel;
procedure CompterLesEspaces(LaPhrase : in TCh100;
                            NbBlancs : out PEntier.TNaturel);
```

C'est le corps du sous-programme qui traduit l'algorithme du sous-programme. Il se compose de 2 parties :

- Une partie déclarative.
- Une partie impérative.

La partie déclarative correspond à la déclaration de toutes les entités nécessaires à la mise en oeuvre de l'algorithme. C'est à dire de tout ce qui est nécessaire à la réalisation de l'algorithme à l'exception des paramètres formels (déclarés dans la spécification)
La partie impérative correspond à la traduction de l'algorithme en instruction (à partir d'un arbre programmatique). Il s'agit d'une suite plus ou moins importante d'instructions mise entres 2 mots : "`begin`" et "`end`"

<u>Règle :</u>

- Le corps d'une fonction doit avoir au moins une instruction `return 'expression';`
- Le corps d'une procédure doit avoir une instruction qui donne une valeur à chacun des paramètres en mode OUT.

Exécution d'un sous-programme :

A l'appel du sous-programme, il y a passage de la valeur des paramètres effectifs dans les paramètres formels correspondants transmis en mode IN ou en mode IN-OUT. Les paramètres en mode OUT ne reçoivent aucune valeur.
Pendant l'exécution : déroulement de l'algorithme.
A la fin de l'exécution : la fin de l'exécution d'un sous-programme fonction à lieu lors de l'exécution return.
Pour un sous-programme procédure, l'exécution se termine lorsqu'on rencontre l'arrêt d'un sous-programme et du programme.

Glossaire d'un sous-programme :

<u>Fonction :</u>
<u>1. Paramètre en entrée</u>
Définition des variables qui ont des valeurs à l'appel de la fonction.

2. Types de résultats
C'est le type ...
Le résultat sera retourné par la variable de travail définie dans le paragraphe ci-dessous.

3. Variables de travail
...

4. Principe
...

Procédure :
1. Paramètre en entrée
Définition de variables qui recevront à l'appel de la procédure la valeur des paramètres formels correspondants.

2. Paramètre en sortie
Définition des variables qui vont recevoir les résultats à restituer au programme d'appel.

3. Paramètres en entrée - sortie
...

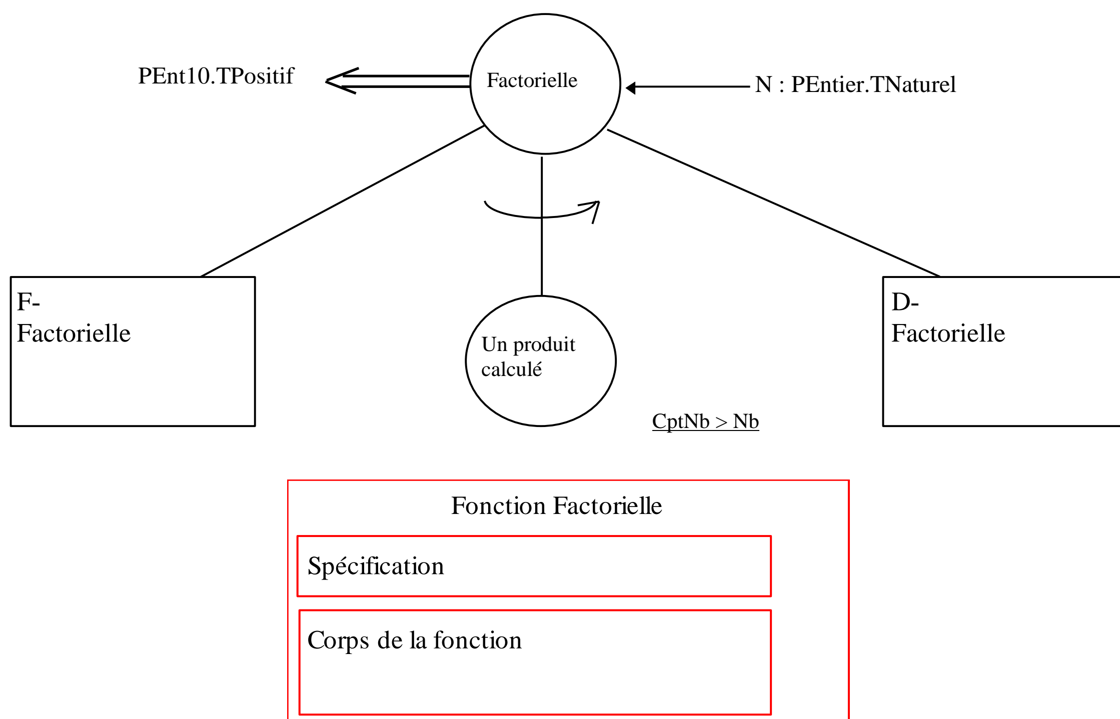
4. Variables de travail
...

5. Principe
...

Ex :
Rédiger une fonction Factorielle qui permet de calculer n! pour $n \in N^*$
1. Paramètres en entrée
N : est une variable de type PEntier.TNaturel qui contient le nombre pour lequel on veut calculer la factorielle.

2. Type de résultat
C'est le type PEnt10.TPositif où PEnt10 est l'instanciation du paquetage PEntGen faite de la façon suivante :

```
package PEnt10 is new PEntGen(NbDeChiffres =>10);
```



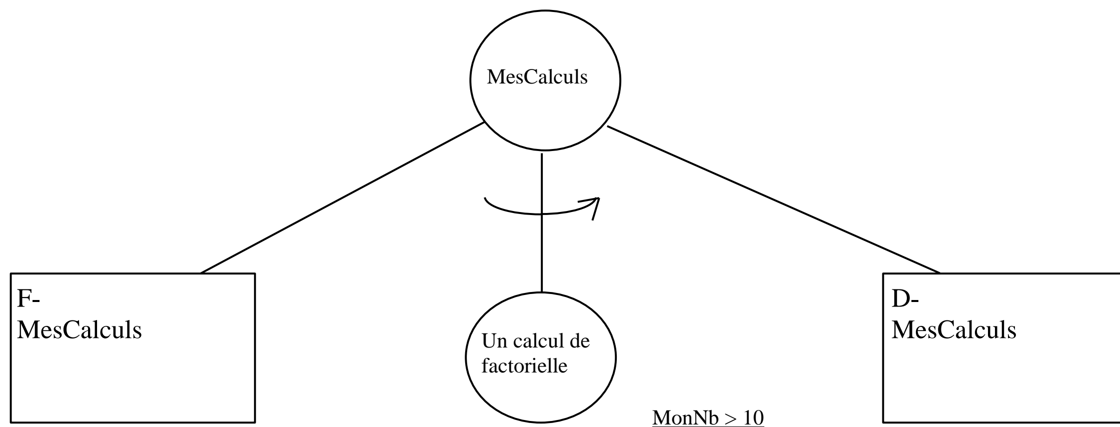
Spécification :

```
function Factorielle(N : in PEntier.TNaturel)
  return PEnt10.TPositif;
```

Corps de la fonction :

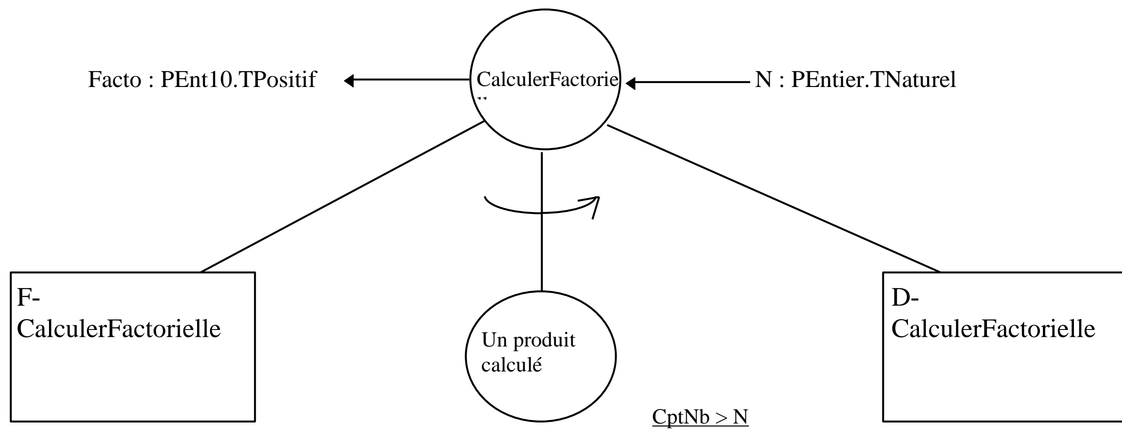
```
function Factorielle (N : in    PEntier.TNaturel)
  return PEnt10.TPositif is
CptNb : PEntier.TPositif;
Facto : PEntier.TPositif;
-- renommage des fonctions >, + de PEntier
-- renommage de la fonction * de PEnt10

begin
  CptNb := 2;
  Facto := 1;
  while not (CptNb>N)
  loop
    Facto := Facto * PEnt10.TEntier(CptNb);
    CptNb := CptNb + 1;
  end loop;
  return Facto;
end Factorielle;
```

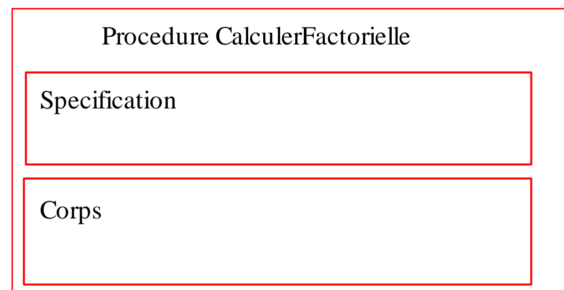
Ex :
Calcul des factorielles de 1 à 10




```
with PChaine;
with PEntier;
procedure MesCalculs is
  package PEnt10 is new PEntGen(NbDeChiffres =>10);
  MonNb : PEntier.TPositif;
begin
  MonNb := 1;
  while not (MonNb>10)
  loop
    PEnt10.put_line(LEntier => Factorielle(N => MonNb));
    MonNb := MonNb + 1;
  end loop;
end MesCalculs;
```

Ex :
Rédiger une procédure CalculerFactorielle qui permet de calculer la factorielle Facto d'un nombre N donné.





Spécification :

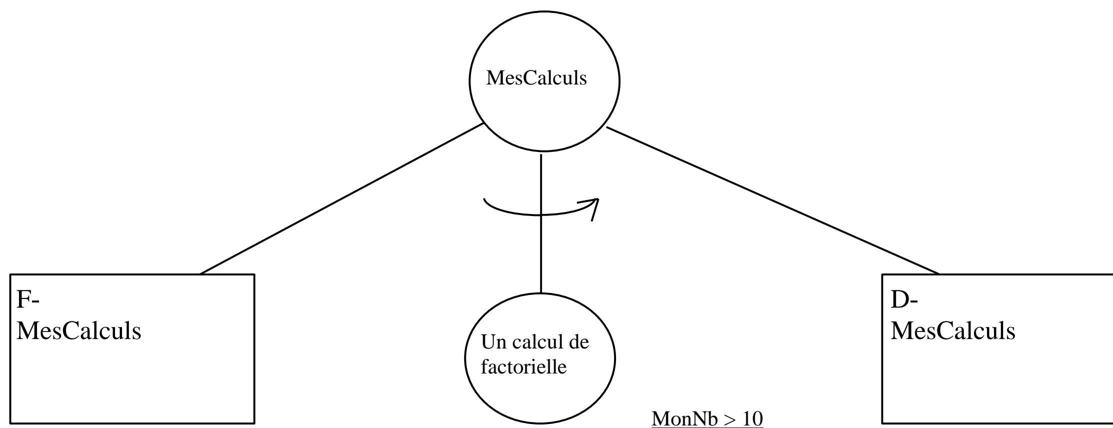
```
procedure CalculerFactorielle (N : in PEntier.TNaturel;
                               Facto : out PEnt10.TPositif);
```

Corps de la procedure :

```
procedure CalculerFactorielle(N : in PEntier.TNaturel;
                              Facto : out PEnt10.TPositif) is
  CptNb : PEntier.TPositif;
  WFacto : PEnt10.TPositif;
  -- renommage du * de PEnt10, +, >, de PEntier
begin
  CptNb := 2;
  Wfacto := 1;
  while not(CptNb>N)
  loop
    WFacto := WFacto * CptNb;
    CptNb := CptNb + 1;
  end loop;
  Facto := WFacto;
end CalculerFactorielle;
```

Programme d'appel

```
with PEntGen;
with PChaine;
procedure MesCalculs is
  paquetage, déclarations, renommages
begin
  MonNb := 1;
  while not (MonNb>10)
  loop
    CalculerFactorielle(N => MonNb;
                        Facto => FactDeMonNombre);
    PEnt10.put_line(Lentier => FactDeMonNombre);
    MonNb := MonNb + 1;
  end loop;
...
end MesCalculs;
```



Exemple de procedure :

```
with PChaine;
with PEntier;
subtype TCh200 is PChaine.TChaine(NbMaxCar => 200);
subtype TCh30 is PChaine.TChaine(NbMaxCar => 30);
procedure ChercherPresence(LeMot : in TCh30;
                           DansLaPhrase : in TCh200;
                           EstPresent : out boolean);

-- corps de la procedure
procedure ChercherPresence(LeMot : in TCh30;
                           DansLaPhrase : in TCh200;
                           EstPresent : out boolean) is
...
...
end ChercherPresence;
```

Les paquetages

Introduction

Définition d'un paquetage : Un paquetage est une collection d'entités et de ressources logicielles que l'on dit "encapsulées". Il permet de regrouper dans une même unité de programmation des déclarations :

- de constantes
- de types
- d'objets
- de sous-programmes(procédures, fonctions)

et le code d'initialisation de ces objets.

Un paquetage permet aussi de séparer la spécification d'un type (ensemble de valeurs et d'opérations sur ces valeurs) de sa mise en oeuvre.

C'est la notion de type abstrait qui est à la base de la programmation orientée objet. Comme les sous-programmes, le paquetage comprend 2 parties :

- La spécification
- Le corps

La spécification du paquetage : Elle constitue l'interface entre l'unité et l'environnement. Elle peut être vue comme un contrat entre le réalisateur du paquetage et les utilisateurs de celui-ci.

Comme le montre le paquetage PChaine, la spécification peut comporter 2 parties :

- La réalisation des objets exportables.
- Une partie dite cachée dont l'existence ne se justifie que par des contraintes de compilation du paquetage.

Le corps du paquetage : Le corps réalise la mise en oeuvre des ressources définies dans la partie spécification. Le corps comporte une partie déclaration et éventuellement une partie instruction qui a pour rôle d'effectuer l'initialisation du paquetage. Le corps contient obligatoirement les corps de tous les sous-programmes qui ont été spécifiés dans la spécification du paquetage.

La spécification d'un paquetage

Elle a pour forme générale :

```
with ...;
with ...;
package Nom_Du_Paquetage is
  ...
private
  ...
end Nom_Du_Paquetage;
```

La spécification peut ne comporter qu'une partie visible (on peut supprimer `private`). Tous les objets déclarés dans la partie visible ont pour portée celle du paquetage : ils ne sont directement visibles que de l'intérieur du paquetage. A l'extérieur du paquetage, on utilise la notation pointée.

Ex :

Rédiger la spécification d'un paquetage permettant de travailler sur des chaînes d'au plus 200 caractères et d'au plus 30 caractères et offrant les sous-programmes suivants :

- Une procedure SaisirPhrase permettant de saisir une phrase LaPhrase d'au plus 200 caractères.
- Une procedure AfficherPhrase permettant d'afficher une phrase LaPhrase d'au plus 200 caractères.
- Une procedure SaisirMot permettant de saisir un mot LeMot d'au plus 30 caractères.
- Une procedure AfficherMot permettant d'afficher un mot LeMot d'au plus 30 caractères.
- Une fonction CEstBienUnMot qui a la valeur VRAIE si une chaîne donnée d'au plus 30 caractères peut être considérée comme un mot (uniquement composée de lettres).
- Une procedure ChercherPresence

Spécification du paquetage :

```
package PPhrase is
  subtype TCh200 is PChaine.TChaine(NbMaxCar =>200);
  subtype TCh30 is PChaine.TChaine(NbMaxCar =>30);
  procedure SaisirPhrase(LaPhrase : out TCh200);
  procedure AfficherPhrase(LaPhrase : in TCh200);
  procedure SaisirMot(LeMot : out TCh30);
  procedure AfficherMot(LeMot : in TCh30);
  function CEstBienUnMot(LeMot : in TCh30)
    return boolean;
  procedure ChercherPresence(LeMot : in TCh30;
                             DansLaPhrase : in TCh200;
                             EstPresent : out boolean);
end PPhrase;
```

Utilisation du paquetage :

```
with PChaine;
with PPhrase;
procedure Essai is
  MaPhrase : PPhrase.TCh200;
  MonMot : PPhrase.TCh30;
  ResPresence : boolean;
begin
  PChaine.put("Tapez une phrase : ");
  PPhrase.SaisirPhrase(LaPhrase => MaPhrase);
  PChaine.put_line("Quel mot voulez-vous chercher ?");
  PPhrase.SaisirMot(LeMot => MonMot);
  if PPhrase.CEstBienUnMot(LeMot => MonMot);
  then
    PPhrase.ChercherPresent(LeMot => MonMot,
                            DansLaPhrase => MaPhrase,
                            EstPresent => ResPresence);
    if ResPresence
    then
      PChaine.put_line("Il y est");
    else
      PChaine.put_line("Il n'y est pas");
    end if;
  else
    PChaine.put_line("Vous avez tapé n'importe quoi");
```

```
    PChaine.put_line("Arrêt du travail");
  end if;
end Essai;
```

Le corps du paquetage

Il a la forme générale :

```
package body Nom_De_Paquetage is
  -- Partie déclarative
begin
  -- Suite d'instructions
exception
  -- traitements d'exceptions
end Nom_De_Paquetage;
```

La partie déclarative contient entre autres les corps de tous les sous-programmes qui ont été spécifiés dans la spécification du paquetage. Elle peut également contenir d'autres sous-programmes qui ne figurent pas dans la partie spécification du paquetage, mais qui se sont avérés nécessaires pour la réalisation de certains corps de sous-programmes.

L'élaboration du corps de paquetage comprend :

- L'élaboration de la partie déclarative. (Les objets locaux sont installés)
- L'exécution des instructions du corps. (Les objets locaux sont initialisés)

Corps du paquetage :

```
package body PPhrase is
  procedure SaisirPhrase(LaPhrase : out TCh200) is
  begin
    PChaine.get_line(LaChaine =>LaPhrase);
  end SaisirPhrase;

  procedure AfficherPhrase(LaPhrase : in TCh200) is
  begin
    PChaine.put_line(LaChaine =>LaPhrase);
  end AfficherPhrase;

  function CEstBienUnMot(LeMot : in TCh30)
    return boolean is
  begin
    ...
  end CEstBienUnMot;
...
end PPhrase;
```

Les paquetages permettent :

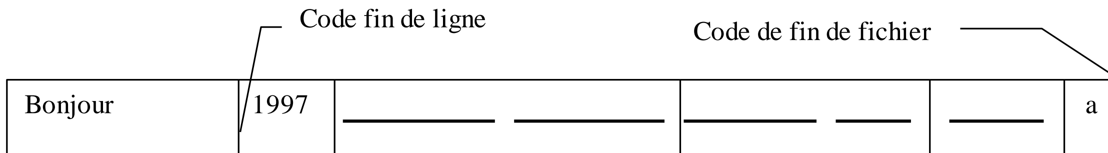
- De regrouper une collection de déclarations
- De former des bibliothèques d'unités de programmation
- De définir des types abstraits

Chapitre 8 : Les fichiers Texte

Définition :

Un fichier texte est défini comme une suite de lignes. Chaque ligne est d'un certain type (Ex : Chaîne de 200 caractères, 12, 'a'), mais d'un seul type.

Exemple de fichier texte :

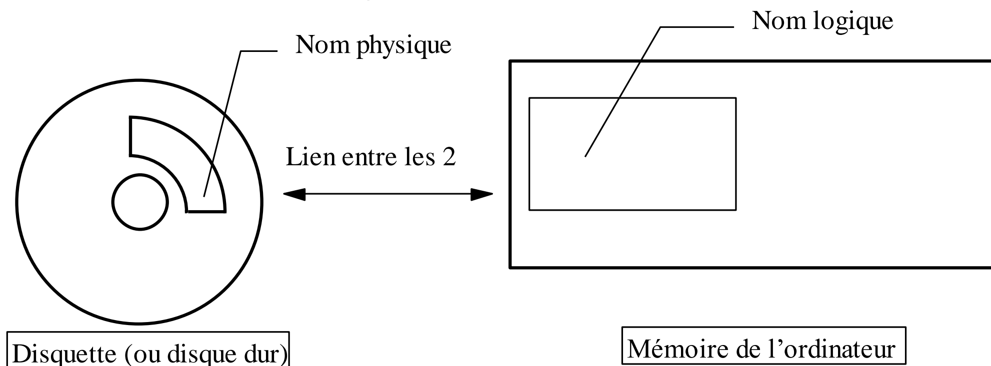


Un fichier texte est identifié par deux noms :

- Un nom *physique* (le nom du fichier sur le disque dur ou sur la disquette)
- Un nom *logique* (qui indique un endroit où sont stockées dans la mémoire les informations sur le fichier)

Quand un fichier a été ouvert, on ne travaille plus qu'avec son nom logique.

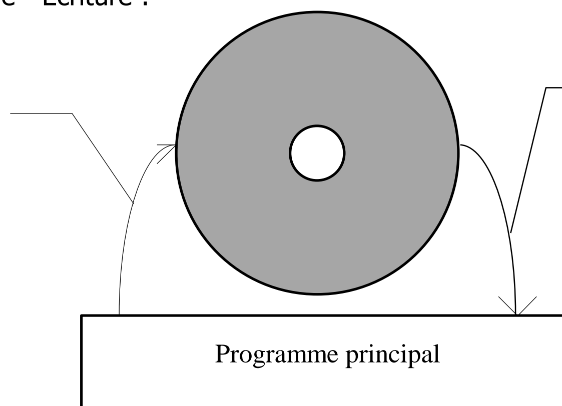
Représentation des noms physiques et logiques :



Principe d'utilisation des fichiers texte :

- L'ouverture en écriture d'un fichier qui existe déjà détruit le fichier existant
- L'ouverture en lecture d'un fichier qui n'existe pas produit une erreur.

Utilisation des modes Lecture - Ecriture :



Chapitre 9 : Le paquetage PFiTex

Programme :

Rédiger un programme CreFiPer permettant de créer un fichier FiPer (de personnes) contenant pour chacune d'elles :

- Son nom (Nom)
- Son prénom (Prénom)
- Son année de naissance (ADN) de 2 chiffres
- Son sexe ('M' ou 'F')
- Marié ? (booléen)

On s'arrête quand on tape FIN.

```
with PChaine, PEntier, PCaractere, PBool, PFiTex;
procedure CreFiPer is
  NomPhy : PFiTex.TNomPhysique(NbMaxCar => 20);
  NomLog : PFiTex.TFiLogique;
  Nom : PChaine.TChaine(NbMaxCar => 20);
  Prenom : PChaine.TChaine(NbMaxCar => 15);
  ADN : PEntier.TNaturel;
  Sexe : character;
  EstMarie : boolean;
  RepMarié : character;
begin
  PFiTex.OuvrirFichier(LeFiLogique => NomLog,
                       DeNomPhysique => +"FiPer",
                       EnMode => PFiTex.SORTIE);
  PChaine.get_line(LaChaine =>Nom);
  while not PChaine.MemeChaine(LaChaine1 => Nom,
                               LaChaine2 => +"FIN");
  loop
    PChaine.get_line(LaChaine => Prenom);
    PEntier.get_line(Lentier => ADN);
    PCaractere.get_line(LeCar => Sexe);
    PCaractere.get_line(LeCar => RepMarie);
    PFiTex.put_line(LaChaine =>Nom,
                    DansLeFilogique =>NomLog);
    PFiTex.put_line(LaChaine => Prenom,
                    DansLeFiLogique => NomLog);
    PFiTex.put_line(LEntier =>ADN,
                    DansLeFiLogique => NomLog);
    PFiTex.put_line(LeCar => Sexe,
                    DansLeFiLogique => NomLog);
    EstMarie := (RepMarie = 'O');
    PFiTex.put_line(LeBool => EstMarie,
                    DansLeFiLogique => NomLog);
    PChaine.get_line(LaChaine =>Nom);
  end loop;
  PFiTex.FermerFichier(LeFiLogique => NomLog);
end CreFiPer;
```

Chapitre 10 : Les tableaux

Notion de tableau

Tableau d'entier :

Indice	1	2	3	4	5
	2	5	2	6	8

Un tableau est une suite d'éléments de même type, contiguë en mémoire, que l'on repère au moyen d'un indice. Le tableau porte un nom.

La notation classique T_i est remplacée par la notation parenthésée T(i).

Types de tableau

Exemple 1 :

```
type TC is array (PEntier.TEntier range 1..5) of PEntier.TEntier;
MonTabC : TC;
TonTabC : TC;
MonTabC := TonTabC;
if (MonTabC = TonTabC)
```

Exemple 2 :

```
type TNC is array(PEntier.TEntier range <>) of PEntier.TEntier;
```

La boîte (<>) symbolise tout intervalle d'indice possible.
Le type TC est dit contraint. La parenthèse de la définition du type définie la contrainte de l'indice.
Le type TNC est dit type non-contraint.

Contraindre le type, correspond à donner une définition de sous-type de TNC, des types dérivés ou à déclarer des variables.

Exemple 3 :

```
subtype TTab1 is TNC(1..20);
type TTab2 is new TNC(1..3);
MonTab1 : TTab1;
MonTab2 : TTab2;
MonTab3 : TNC(1..5);
```

Remarque : L'intervalle d'indice du type TNC pourra contenir n'importe quel intervalle du type TEntier.

Exemple 4 :

```
MonTab : TNC(10..20);
MonTab : TNC(-5..10);
```

Remarque : Un intervalle A..B où A>B (Ex : 5..1) est vide par convention.

Opération sur les tableaux :

Remarque : Il ne faut pas confondre les opérations sur les tableaux avec les éléments du tableau.

Exemple 1 :

```
MonTab1(7) := 25;
MonTab1(2) := MonTab2(1) + 30;
```

Les attributs :

`'first 'last 'length 'range`

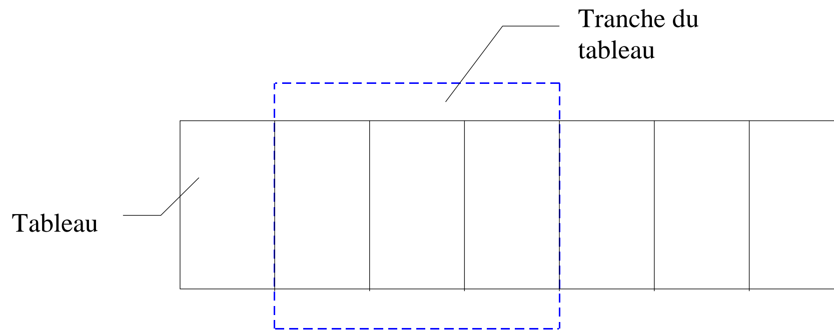
`MonTab1'first` désigne la 1ère valeur de l'indice du tableau.
`MonTab1'last` désigne la dernière valeur de l'indice du tableau.
`MonTab1'length` désigne la longueur du tableau.
`MonTab1'range` désigne l'intervalle entier du tableau.

La notation utilisée seule (Ex : 1..10) ne définie pas le type des valeurs de l'intervalle. Si l'une ou l'autre des valeurs extrêmes de l'intervalle est typées, alors toutes les valeurs de l'intervalle sont de ce type.

Les tranches :

Définition : Une tranche d'un tableau est partie d'un tableau ayant des indices consécutifs.
Une tranche de tableau a pour type son type de base et se désigne sous une forme telle que :

```
MonTab1(3..6);
MonTab(5..2); -- Tranche vide
```



TNC étant un type non-contraint, les tranches de ce tableau de ce type sont également de ce type.

Remarque :

Dans une contrainte, dans une écriture de tranche, les extrémités de l'intervalle peuvent être une expression quelconque (à condition qu'elle soit du bon type).

Ex :

```
a, b : PEntier.TEntier;
Tab : TNC(a..b);
```

Les agrégats :

Initialisation d'un tableau

Ex :

```
Tab : TTab2 := (1=>5 , 2=>-3 , 3=>200);
Tab : TTab2 := (3=>200 , 1=>5 , 2=>-3);
```

ou

```
Tab : TTab2 := (5, -3, 200);
Tab : TTab1 := (3=>15 , 8=>-3 , others =>1);
Tab : TTab1 := (others =>0);
Tab : TTab1 := (3=>5 , 4|10|7=>100);
```

Instructions

```
Tab := (5, 3, ...);
MonTab(5..8) := (5, 3, ...);
```

La concaténation :

On peut concaténer des tableaux ou des tranches grâce à l'opérateur &

Ex :

```
Tab1(5..10) & Tab1(12..16)
Tab1(5..15) & Tab1(20)
Tab1(5) & Tab1(8)
MonTab(2..10) := MonTab(12..17) & MonTab(20..22);
```

Les opérateurs de comparaison :

- = : Tableaux de même longueur et ayant les mêmes éléments 2 à 2.
- /= :2 éléments homologues différents

- `<`, `>` : le signe inférieur sur les tableaux est basé sur l'ordre lexicographique. Les tableaux n'ont pas nécessairement la même taille.

Ex :
```
| 5 | 2 | 6 | 4 |   <   | 5 | 3 | 4 |
```

Les opérateurs logiques `'and'`, `'or'`, `'not'` peuvent être utilisés avec les tableaux de booléens.

Utilisation de tableaux comme paramètres de procédures ou de fonctions :

Ex :
```
type TTab is array(PEntier.TEntier range 1..100) of PEntier.TEntier;
```

Rédiger la fonction `Somme(Tab : in TTab) return PEntier.TEntier` dont la valeur est la somme des éléments de Tab.

```
function Somme (Tab : in TTab) return PEntier.TEntier is
  S : PEntier.TEntier;
  iTab : PEntier.TEntier;
begin
  S := 0;
  iTab := 1;                  -- ou iTab := TTab'first;
  while not (iTab>100)        -- ou (iTab > iTab'last)
  loop
    S := S+ Tab(iTab);
    iTab := iTab +1;
  end loop;
  return S;
end Somme;
```

L'instruction FOR

Exemple :

```
for i in PEntier.TEntier range 1..10   -- i correspond à l'index de la boucle
loop
  PEntier.put(LEntier=>i);
end loop;
```

Remarques :

- L'index ne doit pas être déclaré avant la boucle, c'est la ligne du `for` qui le déclare.
- L'index d'un `for` est local au `for`.
- Dans un `for`, on ne peut pas modifier la valeur de l'index.
- L'intervalle de l'index peut être limité par des expressions.

Ex :
```
for i in PEntier.TEntier range A-1..A+B*5
loop
  ...
  A := A -1;
  ....
end loop;
```

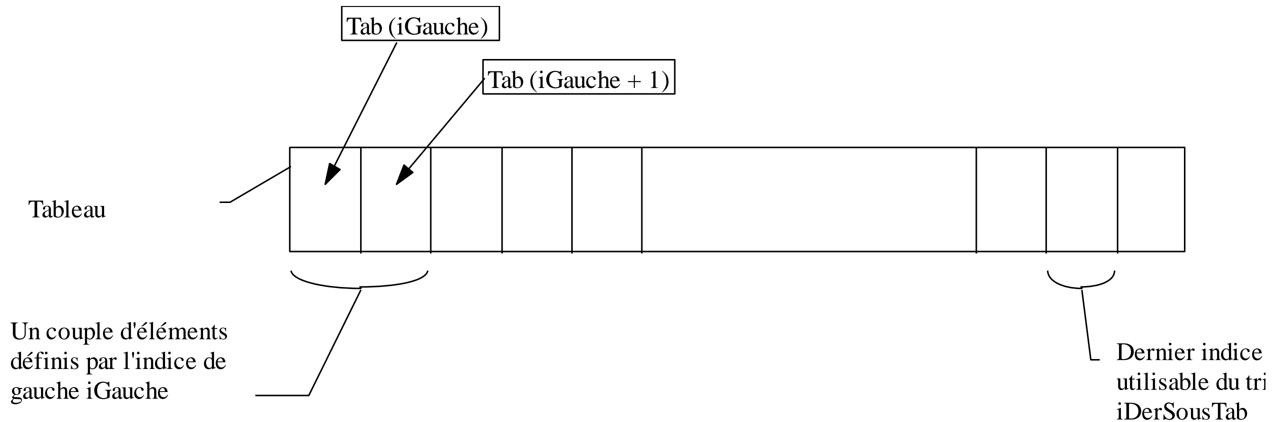
Dans ce cas, les valeurs des limites de l'intervalles sont calculées une fois, à l'arrivée sur le `for`.

Reprise de la fonction Somme dans une boucle `for`:

```
function Somme (Tab : inTTab) return PEntier.TEntier;
  S : PEntier.TEntier;
begin
  S := 0;
  for iTab in TTab 'range
  loop
    S := S + Tab(iTab);
  end loop;
  return S;
end Somme;
```

Chapitre 11 : Le tri à bulle

Explications :



Pour chaque couple nombre du tableau, on vérifie leur grandeur. Si le 1er nombre est supérieur au 2ème, on les échange de place, puis on reprend le tableau depuis le début, jusqu'à ce que tous les nombres aient leur prédécesseur inférieur ou égal à eux-mêmes.

Arbre programmatique :

TrierBulle

F - TrierBulle

D - TrierBulle

Un sous tableau

iDerSousTab = 1

F - Un sous Tableau

* iDerSousTab ← iDerSousTab - 1

D - Un sous Tableau

* iGauche ←

Un couple

iGauche = iDerSousTab

Tab (iGauche) < Tab (iGauche + 1)

F - Un couple

* iGauche ← iGauche +1

Ordre

Désordre

* Aux ← Tab (iGauche)
* Tab(iGauche) ← Tab (iGauche +1)
* Tab(iGauche +1) ← Aux

D- Un couple

Programme ADA :

```
type TTab8 is array (PEntier.TEntier range 1..8) of PEntier.TEntier;
procedure TrierBulle (Tab : in out TTab8) is
begin
  for iDerSousTab in reverse PEntier.TEntier range 2..8
  loop
    for iGauche in 1..iDerSousTab - 1
    loop
      Echange :
      declare
        Aux : PEntier.TEntier;
      begin
        Aux := Tab(iGauche);
        Tab(iGauche) := Tab (iGauche +1);
        Tab(iGauche +1) := Aux;
      end Echange;
    end for;
  end for;
end TrierBulle;
```

Chapitre 12 : Les fichiers de Records

Introduction

Un record est un enregistrement correspondant à l'unité de base des fichiers. Tous les records d'un fichier sont de même type.

PFiSeqGen :

Exercice

Rédiger un programme permettant de créer un fichier de personnes (fichier séquentiel de records) ayant chacune :

- Un nom (20 caractères)
- Une ADN (4 chiffres)
- Un sexe (1 caractère)

```
with PfiSeqGen, PentGen, Pcaractere, PChaine;
procedure CreerFiSeq is
  package PEnt4 is new PEntGen(NbDeChiffres =>4);
  type TPers is
  record
    Nom : PChaine.TChaine(NbMaxCar => 20 );
    ADN : PEnt4.TEntier range 1900..2000;
    Sexe : character;
  end record;
  package PFiSeqPers is new PFiSeqGen(TEnregistrement =>TPers);
  Pers := TPers;
  LoSeqPers : PFiSEqPers.TFiLogique;
begin
  OuvrirFichier(LeFiLogique => LoSeqPers,
                DeNomPhysique => +"FichierPers",
                EnMode =>SORTIE);
  PChaine.get_line(LaChaine => Pers.Nom);
  while not PChaine.MemeChaine (
                 LaChaine1 => PChaine.EnMajuscules(LaChaine => Pers.Nom),
                 LeTabCar => "FIN")
  loop
    PEnt4.get_line(LEntier => Pers.ADN );
    PCaractere.get_line(LeCar => Pers.Sexe );
    PFiSeqPers.Ecrire(LEnreg => Pers;
                      DansLeFiLogique => LoSeqPers);
    PChaine.get_line(LaChaine => Pers.Nom);
  end loop;
  -- fermer le fichier
end CreerFiSeq ;
```

Remarque : On peut ajouter des personnes au bout du fichier à condition que ce soit dans le même programme, et à condition de ne pas fermer le fichier avant.

L'accès direct

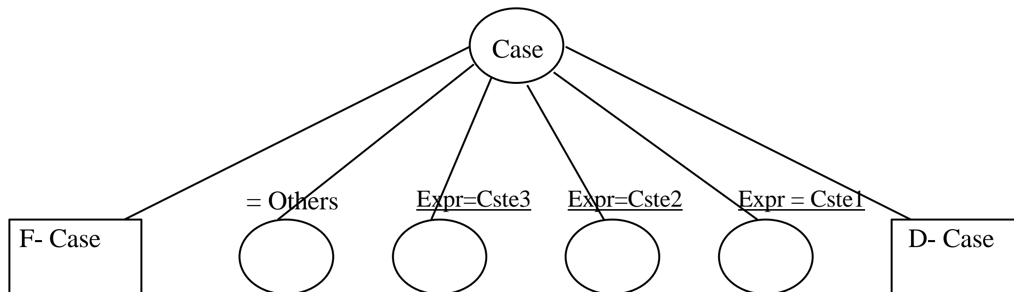
Exercice

Rédiger un programme CreerFiVide préparant un fichier FiDirPers pour 300 personnes.

```
with PFiDirGen, PentGen, Pcaractere, PChaine;
procedure CreerFiVide is
  -- Comme CreerFiSeq avec "Dir" à la place de "Seq".
begin
  PFiDirPers.OuvrirFichier(LeFiLogique => LoDirPers,
                           DeNomPhysique => +"FiDirPers",
                           EnMode => PFiDirSeq.SORTIE);
  Pers := (Nom => " "*20 , ADN => 1900 , Sexe => ' ');
```

```
  for Num in PFiDirPers.TNumero range 1..300
  loop
    PFiDirPers.Ecrire(LEnreg => Pers ,
                     DansLeFiLogique => LoDirPers);
  end loop;
  PFiDirPers.FermerFichier(LeFiLogique => LoDirPers);
end CreerDirVide;
```

L'instruction Case

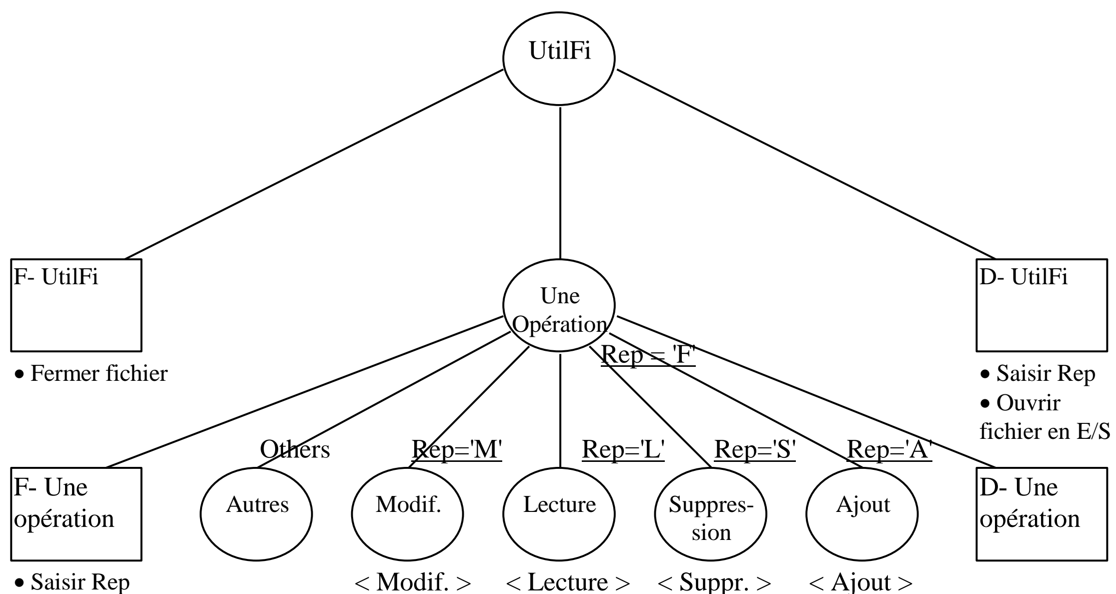



```
case Expression is
  when Constante1 => -- Instruction(s)
  when Constante2 => -- Instruction(s)
  ...
  when ConstanteN => -- Instruction(s)
 [when others     => -- Instruction(s)]
end case;
```

Dans le `case`, les constantes sont du type discret de l'expression (entier ou énumératif)
Remarque : on grouper plusieurs constantes
Ex : `when 7..10 => -- Instruction(s)`
Ex : `when 6|2|25 => -- Instruction(s)`

Exercice :

Rédiger un programme UtilFi permettant de faire sur le fichier FiDirPers les opérations d'Ajout, de Suppression, de Lecture, de Modification, de Fin.



Programme ADA :

```
with PfiDirGen, Pchaine, PentGen, PCaractere;
procedure UtilFi is
  package PEnt4 is new PEntGen(NbDeChiffres =>4);
```

```
  type TPers is
  record
    Nom : PChaine.TChaine(NbMaxCar => 20);
    ADN : PEnt4.TEntier range 1900..2000;
    Sexe : character;
  end record;
  package PFiDirPers is new PFiDirGen(TEnregistrement => TPers);
  Rep : character;
  Num : PFiDirPers.TNumero range 1..300;
  LoDirPers : PFiDirPers.TFiLogique;
begin
  PFiDirPers.OuvrirFichier(LeFiLogique => LoDirPers,
                           DeNomPhysique => +"FiDirPers",
                           EnMode => PFiDirPers.ENTREE_SORTIE);
  PCaractere.get_line(LeCar =>Rep);
  while not(Rep = 'F')
  loop
    case Rep is
      when 'A' =>
        Ajout :
        declare
          ERREUR_SEXE : exception;
          PLACE_PRISE : exception;
        begin
          PFiDirPers.get_line (LEntier=>Num);
          PFiDirPers.lire (LEnreg =>Pers,
                           DansLeFiLogique => LoDirPers,
                           AuNumero => Num);
          if Pers.Nom = 20*' '
          then
            PChaine.get_line (LaChaine =>Pers.Nom);
            PCaractere.get_line (LeCar =>Pers.Sexe);
            case Pers.Sexe is
              when 'M' | 'F' | 'I' =>
                PEnt4.get_line (LEntier =>Pers.ADN);
              when others =>
                raise ERREUR_SEXE;
            end case;
          else
            raise PLACE_PRISE;
          end if;
        end declare;
    end case;
  end loop;

exception
  when ERREUR_SEXE => ...
  when PLACE_PRISE => ...
end UtilFi;
```