Contents

Chapter 1

Introduction

1.1 Objectifs généraux

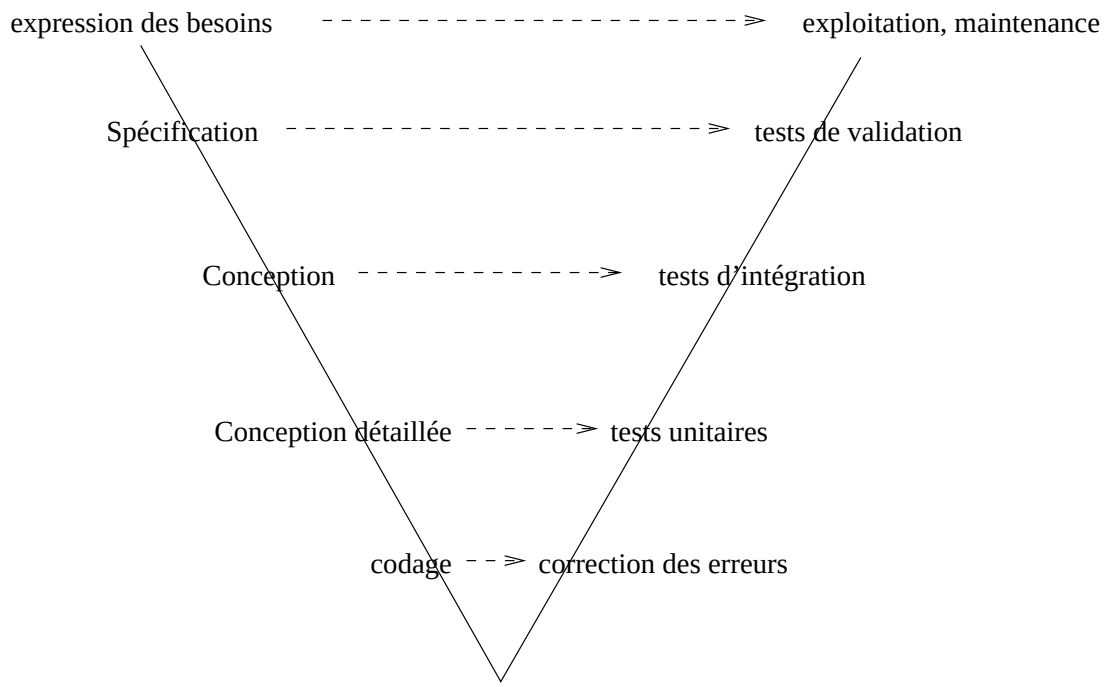
1. Spécifier, concevoir, réaliser des composants logiciels, des applications informatiques.
2. Avec les qualités requises pour un produit industriel
    - efficacité
    - fiabilité
    - lisibilité
    - réutilisabilité
    - extensibilité
3. Donner les bases durables du métier de concepteur, développeur d'applications informatiques
4. Permettre une adaptation rapide aux différents langages de programmation

1.2 Génie logiciel

C'est le domaine qui couvre les étapes reliant l'énoncé d'un problème à la réalisation de sa solution.

1.2.1 Cycle de vie du logiciel

Le cycle de vie d'un logiciel est divisé en 5 étapes indépendantes.



Chaque étape dispose d'outils, de langages adaptés formant un environnement de programmation .

applications informatiques	**génie logiciel**
développement d'une application	cycle de vie
étapes de développement	analyse, conception, codage, tests, maintenance
environnement de programmation	outils, méthodes, langages
méthodes	UML, SADT, SART, Merise
outils	interface utilisateur, gestionnaire de données,
	compilateurs, chargeur, éditeur de liens
langages	ADA, C++, Eiffel, Java

1.2.2 Les étapes du cycle de vie

Analyse Dans cette étape, on s'attache à répondre à trois grandes questions :

- comprendre le problème, identifier les données et les résultats attendus
- dégager les grandes fonctionnalités du système (spécification fonctionnelle)
- identifier les ressources nécessaires (matérielles et humaines)

Cette phase est indépendante de tout langage d'implantation.

Conception Les applications informatiques sont en général suffisamment complexes pour ne pas pouvoir être construites directement. On décompose donc le problème en sous-problèmes plus simples. Ceci donne naissance à un ensemble d'unités informatiques (composants) qui constituent le logiciel d'application. C'est dans cette phase que l'architecture du logiciel est élaborée. Les composants (modules ou unités) et leurs relations sont spécifiés.

Il s'agit d'un processus itératif qui peut conduire à un retour vers l'analyse.

Ada peut être utilisé comme langage de conception.

Codage Il s'agit, dans cette étape, d'implanter (coder) la conception, c'est à dire l'ensemble des composants de l'application. On utilise un langage de programmation (Ada¸ C, Java, C++, Eiffel, Cobol, Fortran, Pascal¸ ...) pour exprimer le code.

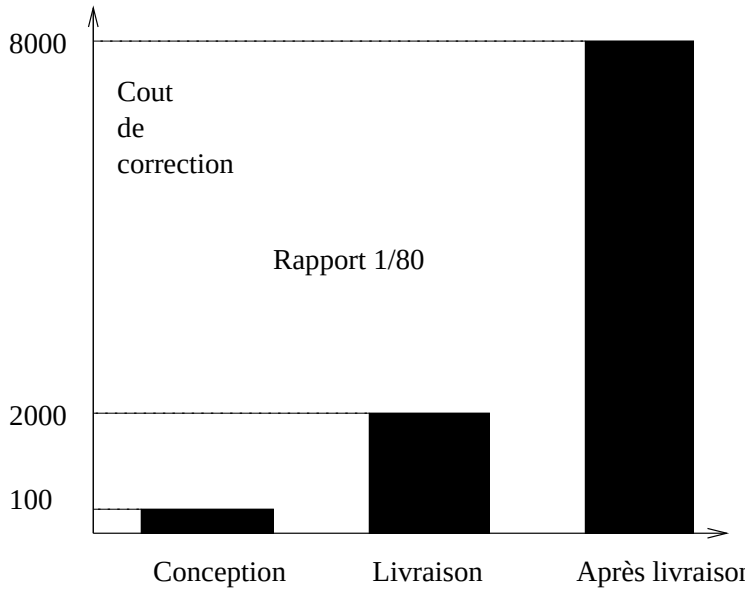
Test

- unitaire (niveau composant) : chaque composant fait, individuellement, l'objet de tests
- d'intégration (niveau système) :

Maintenance L'évolution du système est contrôlée à travers le processus conception/codage/test.

1.3 Qualités d'un logiciel

La qualité du logiciel a un coût. La correction d'une erreur logicielle répond à l'adage traditionnel : mieux vaut prévenir que guérir.



La recherche de la qualité par la prévention plutôt que par la gestion des défaillances apparaît donc comme un excellent investissement.

La qualité a ses critères :

Efficacité

- optimalité de l'utilisation des ressources (temps, espace)
- importance de la macro-efficacité qui se reflète dans une bonne structure de la solution au dépens de la micro-efficacité

Fiabilité L'industrie du logiciel est l'industrie la moins fiable. Un aspect de la fiabilité concerne la correction. On dit qu'un programme est correct si la solution répond bien au problème posé (spécification). L'application réalise les fonctions attendues par l'utilisateur dans les conditions normales et avec les performances attendues.

Un autre aspect de la fiabilité concerne la récupération des pannes. Un programme est considéré comme fiable s'il propose un mode de fonctionnement dégradé, sans provoquer d'effets de bords.

Lisibilité La base d'une bonne lisibilité est le maintien de la relation entre le problème et sa solution.

Au niveau du codage :

- bonne documentation de la solution
- bon style de codage

Au niveau de la conception :

- clarté de l'architecture

Réutilisabilité Les composants logiciels qui forment l'application, conçus de manière à ne pas dépendre du contexte, deviennent réutilisables.

Extensibilité Une application informatique évolue au fil du temps et des besoins de l'entreprise à laquelle elle appartient. Il est important de ne pas devoir reécrire une application lorsque les spécifications subissent des modifications. L'extensibilté est la qualité d'un logiciel peu sensible aux changements de spécification.

Modularité

- couplage faible (interconnexion limitée)
- cohésion forte (éléments internes étroitement reliés)

Portabilité L'application est indépendante du système d'exploitation, du système d'E/S et du matériel.

1.4 Notre approche

1.4.1 Spécification

La spécification est la description du problème à résoudre.

Spécification informelle

- Le problème est décrit en langage naturel.
- La description conserve éventuellement quelques imprécisions, ambiguités.
- Elle est souvent incomplète

Spécification formelle Il ságit d'une description complète et rigoureuse du problème, exprimée dans un langage formel (proche des maths).

Elle permet de faire des preuves de correction et de terminaison d'un calcul.

1.4.2 Codage

Le codage correspond à la traduction de la solution d'un problème dans un langage de programmation (LP).

Par rapport à la phase de spécification, les problèmes posés sont de nature différente. Ils tiennent :

- aux caractéristiques physiques de la machine (E/S)
- à la représentation des données
- à la traduction des actions dans un LP

1.4.3 Tests

En général, un programme ne peut pas être testé pour toutes les données possibles. Des jeux de tests sont élaborés de manière à visiter tous les chemins que peut prendre l'exécution du programme. Cela ne fournit pas une preuve de la correction du programme, mais donne un indice de confiance dans son fonctionnement.

1.5 Exemple de spécification

Spécification informelle On veut savoir si un nombre entier n, non négatif, est un carré parfait.

Le résultat produit est :

- vrai si et seulement si n est un carré parfait
- un entier égal à la racine carrée entière de n.

Spécification formelle $\{n >= 0\} \rightarrow \{(m \in [0..n] \bigwedge vrai \wedge m * m = n) \vee (\exists m \in N \bigwedge faux \wedge m * m < n < (m+1) * (m+1)\}$

1.6 Exemple de spécification

Spécification informelle Calcul de n^p, p est un entier naturel mais il peut être nul sauf si n est nul

Spécification formelle $\{(p > 0) \vee (p = 0 \wedge n \neq 0)\} \rightarrow \{resultat = n^p\}$

1.7 Pourquoi Ada?

1.7.1 Crise du logiciel

- complexité croissante
- coût
- qualité

en 1976, une étude met en lumière le coût prohibitif du logiciel dû à la complexité croissante :

- 75 $ par instruction développée
- 4000$ par instruction modifiée (après livraison)

Un autre étude menée en 1979 par le gouvernement américain et portant sur 487 projets illustre le coût exhorbitant de la maintenance :

- 41,8% est dû à un changement de spécification
- 17,4% est dû à un changement dans le format des données
- 12,4% à des sorties d'urgence
- 9% à des défaillances nécessitant un débogage
- etc ...

La même étude indique que :

- 47% des applications délivrées ne sont jamais utilisées
- 29% payées mais non terminées
- 19% abandonnées ou reécrites
- 3% utilisées après modifications
- 2% utilisées sans changement

1.7.2 Nécessité d'un langage général de haut niveau

- pour applications embarquées
- pour applications de gestion
- pour applications scientifiques

1.7.3 Nécessité d'un langage normalisé

- évitant la multiplication des dialectes
- imposant la certification des compilateurs selon une norme internationale

1.7.4 Cahier des charges d'Ada

L'accent est mis sur la diminution du coût des logiciels en tenant compte de tous les aspects du cycle de vie.

Les caractéristiques mises en avant sont :

- privilégier la facilité de maintenance sur la facilité d'écriture (maintenance=2/3 du coût global du logiciel)
- appliquer un contrôle de type extrémement rigoureux pour diagnostiquer les erreurs le plus en amont possible
- permettre une programmation sûre. Le logiciel traite lui-même les situations anormales
- rendre les applications portables pour ne plus lier les logiciels à un constructeur et donc les rendre indépendantes de toute plate-forme
- permettre des implantations efficaces et donner accès à des interfaces de bas niveau (temps réel)

1.7.5 Ada : norme internationale

Ada est l'aboutissement d'une longue lignée de langages impératifs et procéduraux (Pascal, algol, C).

Il synthétise les meilleurs apports de ces langages et les intègre dans un ensemble cohérent.

Il est utilisé dans des domaines variés : CAO, médical, traitement linguistique, gestion, temps réel, ...

Ada est une norme, aucun dialecte, sur-ensemble ou sous-ensemble n'est admis afin de garantir la portabilité des applications

Une révision périodique de la norme est effectuée. Ada 95 est une révision d'Ada 83. Elle est compatible avec l'ancienne norme.

Pour garantir la conformité à la norme, les compilateurs Ada sont validés :

- le processus de test des compilateurs est devenu un standard international (ISO/IEC-18009:1999)
- la validation est réalisée par des labos indépendants

Actuellement, 26 compilateurs sont validés (Aonix, Intermetrics, ...).

1.7.6 l'histoire d'Ada en deux mots

1968 -> 1973 le DoD constate :

- le coût du logiciel commence à dépasser le coût du matériel des principaux systèmes de défense
- un accroissement de 51% du coût de ses systèmes informatiques
- la diversité et l'inadéquation des langages de programmation utilisés (450)
- le manque d'environnement de développement

1975 -> 1977 établissement du cahier des charges pour un langage de programmation de haut niveau

1978 -> 1979 examen des propositions

1980 publication du manuel de référence

1983 approbation du manuel de référence par la norme ANSI

1987 normalisation ISO, certification des compilateurs (3500 tests)

1.7.7 Caractéristiques principales du langage

Ada supporte les concepts :

- de modules (paquetages)
- de généricité
- d'exceptions
- de tâches (fils de contrôle parallèles avec communication)

Il possède une riche bibliothèque de modules prédéfinis.

Il s'interface avec d'autres langages comme C, fortran, cobol, ...

Le compilateur Ada d'Intermetrics peut générer du J-code (code pour la machine virtuelle java). Il est aussi possible d'utiliser Ada pour développer des applets Java.

1.8 Applications

Ada est le langage le plus populaire pour les applications critiques :

- mise en jeu des vies humaines
- importance des coûts liés aux pannes

Domaines d'applications :

- transports
  - conduite automatique de trains
  - systèmes de contrôle en avionique
- production d'énergie
  - conduite de centrales nucléaires
  - conduite de barrages

Entreprises impliquées :

- industries aéronautiques et spatiales : Boeing¸ Nasa (station spatiale), Chandler Evans (système de contrôle de moteur), ESA (observation spatiale, mission saturne conjointement avac la Nasa), Ford Aerospace, Intermetrics, Lockheed, Rockwell Space System (navette spatiale), ...
- contrôle du trafic aérien (CTA) : Hughes (système de CTA canadien), Loral FSD (système de CTA US), Thomson-CSF (système de CTA français).
- bateaux : Vosper Thornycroft Ltd (contrôle de navigation).
- trains : European Rail (système d'aiguillage), EuroTunnel, Extension du métro de Londres, GEC Alsthom (systèmes de contrôle des signaux de trains et TGV), TGV France (système d'aiguillage), Westinghouse Australia (système de protection automatique des trains).
- médical : Coulter Corp(Onyx analyse de l'hématologie).
- nucléaire.

1.8.1 Bilan de l'utilisation d'Ada

Les études économiques sur l'utilisation du langage Ada font apparaître que, par rapport aux autres langages, Ada :

- coute moins cher en développement
- raccourcit la phase d'intégration
- provoque moins d'erreurs résiduelles

Ada est donc choisi sur des citères économiques et de sûreté.

Chapter 2

Premiers pas en Ada

2.1 Structure d'un programme Ada

```
-- exemple 1
with Ada.Integer_Text_io;
with Ada.Text_io;
procedure exemple1 is
  maNote:Natural;
begin
  Ada.Integer_Text_io.get(maNote);
  maNote:=maNote+2;
  Ada.Text_io.put("Nouvelle note :");
  Ada.Integer_Text_io.put(maNote);
end exemple1;
```

Ce programme reflète la structure générale d'un programme Ada. On remarque 3 parties distinctes :

1. la première permet d'importer des objets externes (`with ...`)
2. la seconde permet de créer des objets localement (`procedure...begin`)
3. la troisième est une suite d'instructions qui opère sur ces objets (`begin...end`)

2.1.1 Structure générale d'un programme Ada

```
--imports
procedure nom_proc is
--spécification des données
begin
--suite d'instructions qui implantent l'algorithme
end  nom_proc;
```

2.1.2 Notion de type de données

Tous les objets (locaux ou externes) ont un type. Un type définit l'ensemble des valeurs que peut prendre un objet ainsi que les opérations permises sur ses valeurs.

Exemple de type :

```
Ensemble de valeurs : N
+
Ensemble d'opérations sur ces valeurs (+, :=, ...)
```

Chaque type possède un nom. En Ada, le type défini ci-dessus se nomme `Natural`.

On associe un objet à son type au moyen d'une déclaration.

Dans le programme précédent, la déclaration associe l'objet de nom `maNote` (qui est ici une variable) au type `Natural`.

```
maNote:Natural;
```

2.1.3 Commentaires

Ligne de texte commençant par - -

Ils peuvent être placés n'importe où dans le programme.

2.2 Introduction aux Entrées/Sorties

```
-- exemple 2
with Ada.Text_io;
with Ada.Integer_Text_io;
procedure exemple2 is
 subtype Note is Natural range 0..20;
 maNote:Note;
 car:Character;
begin
 Ada.Text_io.put_line("debut du programme");
 Ada.Text_io.put("Tapez un caractère : ");
 Ada.Text_io.get(car);
 Ada.Text_io.put("Le caractère tapé est : ");
 Ada.Text_io.put(car);
-- lecture d'une note à partir du clavier
 Ada.Integer_Text_io.get(maNote);
--on ajoute un bonus
 maNote:=maNote+2;
 Ada.Text_io.new_line;
 Ada.Text_io.put("maNote=");
-- affichage de la nouvelle note sur l'écran
 Ada.Text_io.put(Note'image(maNote));
 Ada.Text_io.new_line;
 Ada.Text_io.put("fin du programme");
end exemple2;
```

Les 3 parties du programme apparaissent de nouveau.

La **première** permet d'importer des objets (`put, get, ...`) destinés à permettre la lecture et l'écriture de valeurs textuelles.

La **seconde** permet de déclarer 3 objets : 2 variables (`maNote` et `car`) et un nouveau type de données appelé `Note` dont l'ensemble des valeurs est le sous-ensemble des valeurs du type `Natural` limité à l'intervalle `0..20`. Les opérations possibles sur ce nouveau type sont alors les mêmes que celles du type `Natural`.

On notera aussi l'apparition d'un autre type de données `(Character)` dont l'ensemble des valeurs est l'ensemble des caractères ASCII 7 bits (caractères usuels à l'exception des caractères accentués).

La troisième partie est une suite d'instructions qui a pour but :

- de lire à partir du clavier puis d'afficher sur l'écran un caractère unique compatible avec le type `Character`.
- de lire une valeur compatible avec le type `maNote`, de l'augmenter de `2` puis de l'afficher sur l'écran.

L'attribut *image*

Il traduit la valeur entière à afficher en une représentation sous la forme d'une chaine de caractères. En effet, on se rappelle que les objets importés ne manipulent que des données textuelles.

2.3 L'unité `Text_io`

L'unité `Text_io` Elle regroupe uneEnsemble d'objets qui permettent de réaliser des opérations d'Entrée/Sortie de carctères simples ou de chaines de caractères.

`with Ada.Text_io;` Cette clause importe le *paquetage* (sorte de boite à outils) `Text_io`. L'ensemble des objets (outils) contenus dans ce paquetage deviennent alors utilisables dans le programme.

Exemple d'outils disponibles dans `Text_io`

```
get, put, put_line, new_line
```

Préfixage par `Ada.Text_io` Généralement, plusieurs boites à outils peuvent être importées. Pour spécifier sans ambiguité où est localisé l'outil voulu, il faut prefixé son nom par le nom de l'unité auquel il appartient.

2.4 E/S sur les entiers

```
-- exemple 3
with Ada.Text_io;
with Ada.Integer_Text_io;
procedure exemple3 is
 subtype Entier is Integer range -100..100;
 I:Entier:=0;
 car:Character;
begin
 Ada.Text_io.put_line("debut du programme");
 Ada.Text_io.put("Tapez un caractère : ");
 Ada.Text_io.get(car);
 Ada.Text_io.put("Le caractère tapé est : ");
 Ada.Text_io.put(car);
 Ada.Text_io.put("Tapez un entier : ");
 Ada.Integer_Text_io.get(I);
 I:=I+10;
 AdA.Text_io.new_line;
```

```
 Ada.Text_io.put("I=");
 Ada.Integer_text_io.put(I);
 Ada.Text_io.new_line;
 Ada.Text_io.put("fin du programme");
end exemple3;
```

Le programme utilise les objets (procédures) de 2 bibliothèques (paquetages Ada), `Text_io` et `Integer_Text_io` pour réaliser des entrées et des sorties de données.

Le paquetage `Text_io` permet les lectures et les écritures de chaines de caractères et de caractères.

Le paquetage `Integer_Text_io` permet les lectures et les écritures sur des valeurs entières.

On remarque aussi l'introduction d'un nouveau type de données (`Integer`) qui correspond à l'ensemble des entiers relatifs.

2.4.1 Lecture d'entiers : `get`

Lorsqu'un appel à la procédure `get` est effectué et que son paramètre est une variable d'un type entier (`get(I)`), le programme attend que l'opérateur tape un entier au clavier. Les espaces et les caractères de fin de ligne ne sont pas pris en compte dans la saisie.

2.4.2 Affichage d'entiers : `put`

Lorsque la procédure `put` prend en paramètre une valeur entière, celle-ci est affichée sur l'écran selon un format standard (taille minimum capable d'accueillir toutes les valeurs entières possibles : 11 caractères pour une machine de 32 bits, 6 pour une machine de 16 bits).

```
put(-56);
```

affichera 8 espaces suivis de `-56`

```
        -56
```

Il est possible de spécifier le format d'affichage en ajoutant un paramètre supplémentaire

```
put(-56,3);
```

affichera la valeur `-56` sans espaces préalables

```
-56
```

De même,

```
put(-56,4); -- ajout d'un espace devant
put(-56,1); -- l'affichage n'est pas tronqué
```

2.5 E/S sur les flottants

```
-- exemple 4
with Ada.Text_io;
with Ada.Float_Text_io;
procedure exemple4 is
 r:Float;
begin
 Ada.Text_io.put_line("debut du programme");
 Ada.Text_io.put("Tapez un nombre réel : ");
 Ada.Float_Text_io.get(r);
 Ada.Text_io.put("Le nombre saisi est : ");
 Ada.Float_Text_io.put(r);
 Ada.Text_io.new_line;
 Ada.Text_io.put("fin du programme");
end exemple4;
```

On importe dans ce programme, le paquetage `(Float_Text_io)` qui réunit les objets de lecture et d'écriture sur de nombres réels.

2.5.1 Lecture de réels : get

Lorsqu'un appel à la procédure `get` est effectué et que son paramètre est une variable d'un type réel (`get(r)`), le programme attend que l'opérateur tape un réel au clavier. Les chiffres décimaux sont situés derrière un point (exemples : -234.45, 0.0067). Les espaces et les caractères de fin de ligne ne sont pas pris en compte dans la saisie.

2.5.2 Affichage de réels : put

Par défaut, un réel est affiché selon le format suivant : `x.xxxxxE+xx ou x.xxxxxE-xx`. Soit 5 chiffres décimaux significatifs et un chiffre avant la virgule. L'exposant est formé de la lettre E suivie du signe et des deux chiffres de l'exposant.

Il est possible de préciser son propre format en utilisant la procédure put avec 3 paramètres supplémentaires.

Le second paramètre indique le nombre de caractères avant le point (y compris des espaces si nécessaire car il n'y a toujours qu'un seul chiffre avant le point).

Le troisième paramètre indique le nombre de caractères après le point.

Le quatrième paramètre indique le nombre de caractères après E.

Exemples :

```
put(0.023,3,2,1);
  2.30E-2
put(0.0234,1,2,0);-- pour une notation décimale normale
0.02
```

2.6 La clause use

2.6.1 Intérêt

1. Permet d'alléger l'écriture

2. D'intégrer chaque objet dans l'environnement du programme au lieu d'y accéder grâce au préfixe.

```
-- exemple 5
with Ada.Text_io;use Ada.Text_io;
with Ada.Integer_Text_io;
use Ada.Integer_Text_io;
procedure exemple5 is
 subtype Entier is Integer range -100..100;
 I:Entier:=12;
begin
 put("I=");put(I);new_line;
end exemple5;
```

Sans l'utilisation de la clause `use`, on aurait écrit :

```
-- exemple 5
with Ada.Text_io;
with Ada.Integer_Text_io;
procedure exemple5 is
 subtype Entier is Integer range -100..100;
 I:Entier:=12;
begin
 Ada.Text_io.put("I=");
 Ada.Integer_Text_io.put(I);
 Ada.Text_io.new_line;
end exemple5;
```

2.6.2 Emploi multiple de la clause `use`

Comment peut-on (le compilateur peut-il) distinguer les outils de `Text_io (put, put_line, get, ...)` et ceux de `Integer_Text_io (put, ...)` ?

Réponse : par le type et le nombre des paramètres des procédures impliquées (`put` et `get`) (caractères ou chaines de caractères dans le premier cas, type `Entier` dans l'autre).

```
-- exemple 6
with Ada.Text_io;use Ada.Text_io;
with Ada.Integer_Text_io;
use Ada.Integer_Text_io;
procedure exemple6 is
 subtype Entier is Integer range -100..100;
 I:Entier;
 car:Character;
begin
-- c'est Text_io qui est choisi car le paramètre est une chaine de
-- caractères
 put_line("debut du programme");
 put("Tapez un caractère : ");
 get(car);
-- c'est Integer_Text_io qui est choisi car I est du type Entier
 put(I);
 ....
 ....
end exemple6;
```

Bien qu'elle permette d'alléger l'écriture d'un programme, la clause `use` rend le programme plus difficilement maintenable car moins lisible.

On se rapproche d'une programmation par objets lorsqu'on évite de l'employer. En effet, lors de son utilisation, chaque objet est préfixé par le paquetage auquel il appartient.

2.7 Production d'un programme Ada

2.7.1 Un programme source Ada

- est une suite de caractères (texte écrit dans le langage Ada)
- constitué de plusieurs *unités* nommées (exemple `Text_io`)
- une unité peut être une *fonction*, une *procédure* ou un *module (paquetage)*
- un programme Ada possède une *unité principale* (qui est une procédure)

2.7.2 Doit être compilé

- Chaque unité est compilée *séparément*
- Le résultat de la compilation est conservé dans une *bibliothèque*
- En cas de modification du programme, seules les unités modifiées sont recompilées

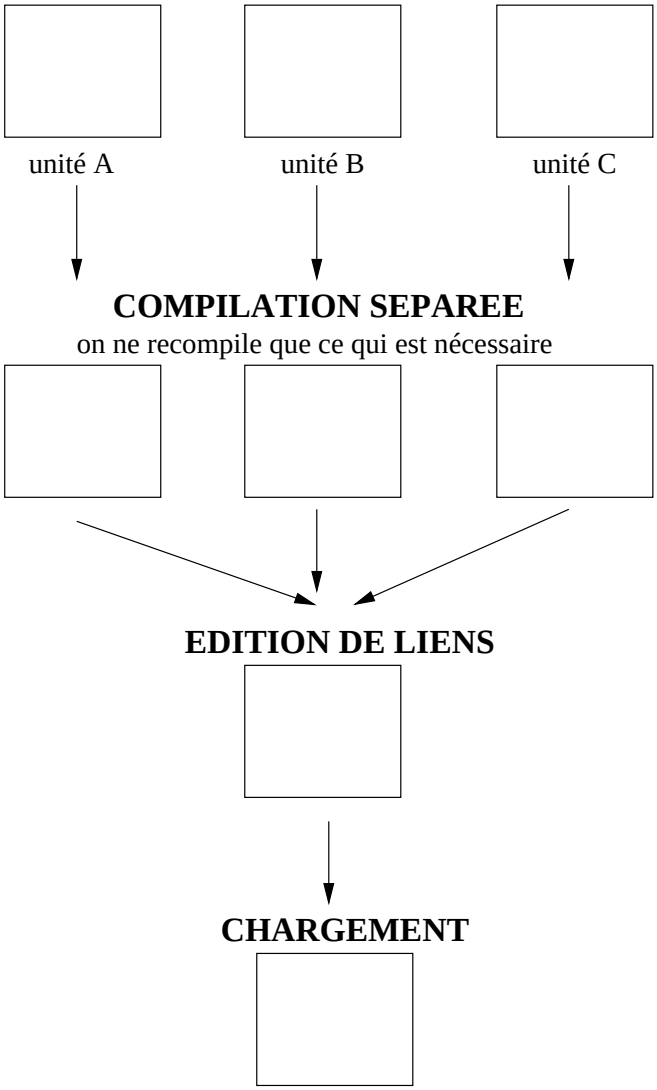
2.7.3 Doit être rendu exécutable

- L'étape d'édition de liens rassemble les unités compilées en une seule entité traduite en langage machine.
- Des liens peuvent être établis avec des unités précompilées dans des bibliothèques prédéfinies.

2.7.4 Doit être chargé

- En mémoire centrale pour être exécuté.

2.8 Chaine de production de programmes



fichier texte Ada
(fichier source)
−> produit par un éditeur de textes
−> stocké sur disque

le résultat de la compilation
est rangé dans une bibliothèque
−> c'est un fichier objet
−> crée par un compilateur
−> stocké sur disque

c'est un fichier exécutable
−> crée par un éditeur de liens
−> stocké sur disque
−> traduit en langage machine

c'est un programme exécutable
−> rangé en mémoire centrale

2.9 Compilation

2.9.1 Notion de grammaire

Une *grammaire* est définie comme un quadruplet :

```
<A,VT,V,P>
```

A : point d'entrée

VT : vocabulaire terminal *(+,-,0,1,_,...)*

V : vocabulaire non terminal *(ident,lettre,symbole,...)*

P : règles de productions *(élément de V::=suite de terminaux et non terminaux)*

Les règles sont exprimées dans le formalisme BNF (Backus et Naur Form) ou assimilé.

Notations : le formalisme BNF étendu

symbole	signification	exemple
[]	partie facultative	[+]
|	alternative	+|−
...	intervalle	0...9
{}	répétition 0 ou n fois	{*lettre*}
$\{\}^+$	répétition stricte (1 à n fois)	$\{lettre\}^+$
()	mise en facteurs	$[+\|-)nombre$

2.9.2 Analyse lexicale

- Transformation de la suite de caractères en une suite de symboles du langage
- Les symboles du langage sont :
  1. les identificateurs : `X_11,VAR,toto`
  2. les constantes numériques : `11, 42.21`
  3. les mots clés : `procedure, begin, with, is, ...`
  4. les opérateurs : `+, -, *, /, ...`
  5. les espaces, les tabulations
- Cette transformation est basée sur un ensemble de règles lexicales

2.9.3 Règles de construction d'un identificateur Ada

<ident>::=<lettre>{<symbole>|_<symbole>}

<symbole>::=<lettre>|<chiffre>

<lettre>::=A...Z|a...z

<chiffre>::=0...9

Exemples Identificateurs non valides

`A_, _A, 1_ADA, O__K, un$, A+B`

Identificateurs valides

`X_1, TINTIN_ET_MILOU, OK`

2.9.4 Analyse syntaxique

- Transformation d'une suite de symboles en une suite de phrases du langage
- Cette transformation est basée sur un ensemble de règles syntaxiques décrivant les structures syntaxiques du langage et exprimées dans le formalisme BNF (par exemple)
- Pour éviter les ambiguités d'interprétation des phrases du langage, à une suite donnée de symboles ne peut correspondre qu'une seule phrase du langage

Structure syntaxique des expressions arithmétiques Les éléments non terminaux de la grammaire sont notés entre < et >

<expression>::=<facteur>|<facteur><opadd><expression>

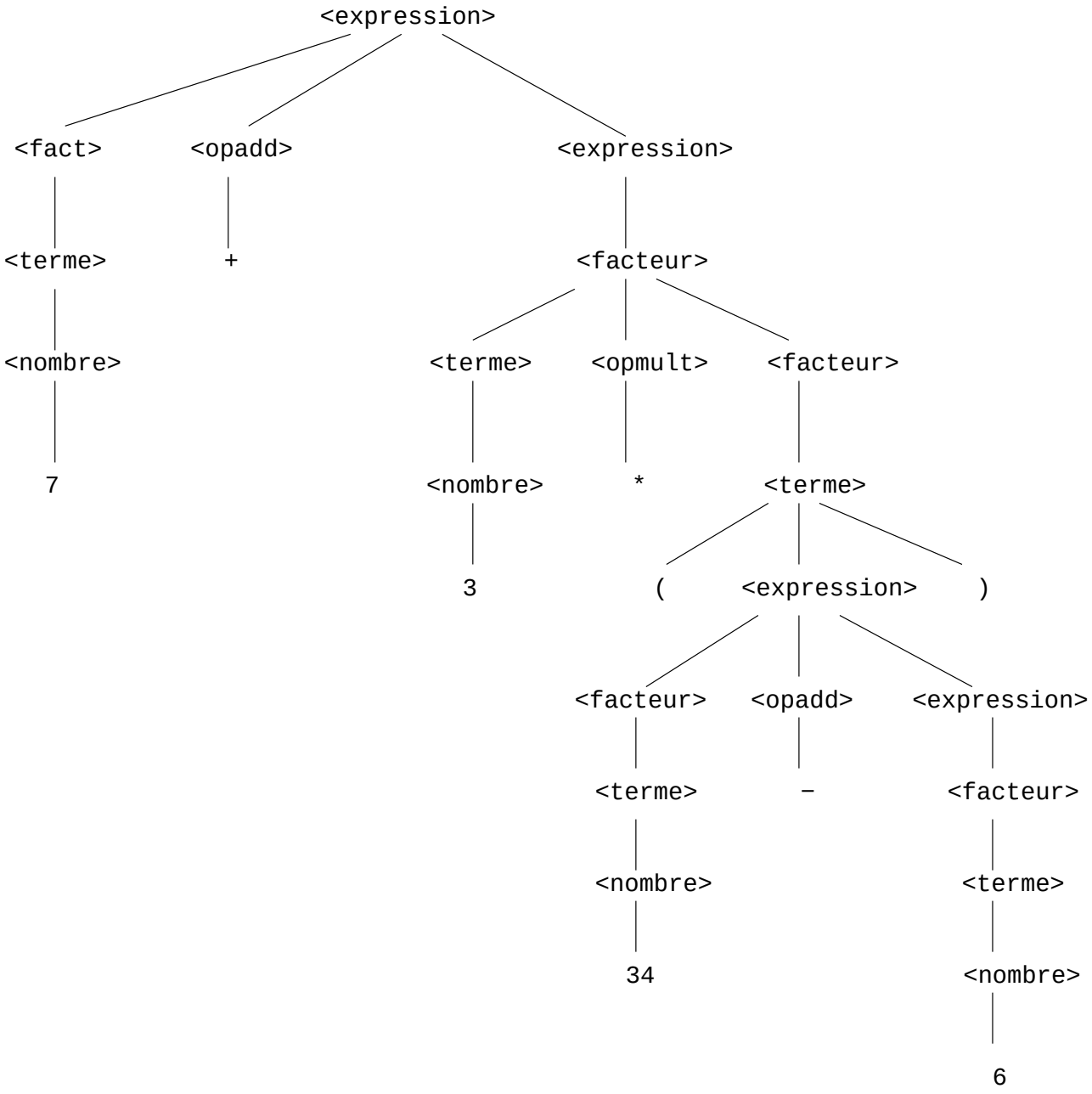
<facteur>::=<terme>|<terme><opmult><facteur>

<terme>::=<ident>|<nombre>|<expression>|(<expression>)

<opadd>::=+|-

<opmult>::=|/*

Exemple Analyse syntaxique de l'expression : `7+3*(34-6)` La structure syntaxique de l'expression est exhibée par l'arbre syntaxique



Pour un évaluateur humain, l'expression vaut `91`

L'évaluation, à partir de cet arbre donne le même résultat

L'expression syntaxique est bien formée (syntaxiquement correcte) si l'analyse syntaxique aboutit.

2.9.5 Analyse sémantique

- Vérifie que le **type** d'une construction syntaxique correspond au type attendu

Exemple L'opérateur `mod` est une fonction qui à 2 entiers fait correspondre un entier. Elle nécessite donc 2 opérandes d'un type entier. Le contrôle de type vérifiera que c'est bien le cas. Elle vérifiera aussi que le résultat est lui-même du type entier.

2.9.6 Génération de code

A partir d'une représentation intermédiaire du code source produite par les phases antérieures de compilation et de la table des symboles, le générateur de code produit le code cible.

Ce code cible peut être exprimé :

- En langage machine translatable (les adresses des objets sont relatives). Ceci permet la compilation séparée de modules (unités)
- En code d'assemblage (les adresses et les instructions sont symboliques). Cette technique facilite la production de code mais nécessite une phase d'assemblage ultérieure.

2.9.7 Edition de liens

- Réunit les différents modules dans un même espace d'adressage
- Permet l'utilisation, dans un programme, de modules déjà compilés

2.10 Exemple de production de programme

2.10.1 Texte source erroné

```
-- exemple 7 [?, Fayard]
with Ada.Text_io;
with Ada.Integer_Text_io;
procedure exemple7 is
 subtype Entier is Integer range -100..100;
 I:Integer;
 car:Character:='X';
begin
 put("Tapez un caractère : ");
 Ada.Integer_Text_io.get(I);
 Ada.Integer_Text_io.put(car);
end exemple7;
```

2.10.2 Résultat de la compilation, première erreur

```
9       put("Tapez un caractère :  ");
                <1>
  1 **IDE No declaration having this name is visible at this point - RM 8.3.
= More Information=======================
                -> Note the potentially visible item(s) :
ENUMERATION_IO.PUT at line 357 of TEXT_IO
Specification, procedure specification
FLOAT_IO.PUT at line 281 of TEXT_IO
Specification, procedure specification
INTEGER_IO.PUT at line 241 of TEXT_IO
Specification, procedure specification
        -> Direct visibility might be achieved by the appropriate use clause
```

2.10.3 Résultat de la compilation, seconde erreur

```
11      Ada.Integer_Text_io.put(car);
                                                        1
  1 **IDE There is a type inconsistency in this expression
= More Information=======================
                -> Different interpretations exists for PUT :
-PUT at line 2 of Ada.Integer_Text_io
specification, its profile is
(STANDARD.STRING;STANDARD.INTEGER;TEXT_IO.NUMBER_BASE)
-PUT at line 2 of Ada.Integer_text_io
specification, its profile is
(TEXT_IO.FILE_TYPE;STANDARD.INTEGER;TEXT_IO.FIELD;TEXT_IO.NUMBER_BASE)
etc ...
        -> But the expression has the following type
        - STANDARD.INTEGER
```

Plusieurs `put` existent dans `Ada.Integer_Text_io` avec un nombre différent de paramètres, mais aucun ne prend un paramètre de type `Character`.

2.11 Type énumératif

Comment définir un nouveau type de données par énumération de toutes ses valeurs.

```
-- exemple 8
with Ada.Text_io;use Ada.Text_io;
procedure exemple8 is
 type Jour is (lundi,mardi,mercredi,jeudi,vendredi,samedi,dimanche);
 type WE is (samedi, dimanche);
 unJour:Jour:=lundi;
 repos:WE:=dimanche;
begin
  if unJour=lundi then
    put("1er jour de la semaine");
    put(Jour'image(unJour));
  elsif repos=WE'(dimanche) then
       put("c'est dimanche");
     else
       put("c'est samedi");
  endif;
end exemple8;
```

Nous déclarons, dans ce programme un nouvel objet, le type `Jour`, dont l'ensemble des valeurs est constitué des valeurs spécifiées dans les parenthèses `(lundi, mardi, ...)`.

On utilise le qualificateur `WE'(dimanche)` pour spécifier que la valeur `dimanche` est celle du type `WE` et non du type `Jour`. Cette facilité permet de rendre le contrôle de type plus rigoureux. En effet, ainsi le programmeur affiche explicitement son intention. La valeur `dimanche` employée est celle du type `WE`. Cela lève l'ambiguité possible au niveau de la compilation.

2.12 Un nouveau programme erroné

```
-- exemple 9
with Ada.Text_io;use Ada.Text_io;
procedure exemple9 is
 type Jour is
(lundi,mardi,mercredi,jeudi,vendredi,samedi,dimanche);
 type WE is (samedi,dimanche);
 unJour:Jour:=lundi;
 repos:WE:=mardi; -- erreur 1
begin
  if unJour=lundi then
    put("1er jour de la semaine");
    put(Jour'image(pred(lundi))); -- erreur 2
  elsif repos=WE'(dimanche) then
       put("c'est dimanche");
     else
       put("c'est samedi");
  endif;
end exemple9;
```

Erreur 1 Le compilateur détecte que la valeur `mardi` n'appartient pas au type `WE`. Une valeur ne peut appartenir qu'à un seul type. On parle de typage fort

Erreur 2 `Jour'pred(lundi)` rend la valeur qui précède `lundi` dans le type `Jour`. Or cette valeur n'existe pas.

Ada envoie un signal (`CONSTRAINT_ERROR`) indiquant que l'on dépasse les bornes de l'intervalle `lundi..dimanche` .

2.13 Boucle `for`

La boucle `for` est une structure de contrôle itérative. Elle exprime la répétition d'une suite d'instructions. Pour employer la boucle `for`, il faut connaître le nombre d'itérations souhaitées. C'est la raison pour laquelle, à la boucle `for`, est associé une variable (un compteur). Dans l'exemple 10, le compteur se nomme `i`. Les 2 instructions qui constituent le corps de la boucle seront répétées pour toutes les valeurs du type `Jour` comprises entre `vendredi` et `dimanche`.

```
-- exemple 10
for i in vendredi..Jour'(dimanche) loop
   Ada.Text_io.put(Jour'image(i));
   Ada.Text_io.put("fin de semaine");
end loop;
-- exemple 11
for unJour in Jour loop
   Ada.Text_io.put(Jour'image(unJour));
   Ada.Text_io.new_line;
end loop;
-- exemple 12
for i in WE loop
   Ada.Text_io.put(WE'image(i));
   Ada.Integer_Text_io.put(WE'pos(i));
-- le rang de i dans la suite des valeurs de WE est affiché
end loop;
```

Remarque nř1 : Pour des raisons de portabilité du programme, en l'occurrence d'indépendance du code par rapport au format des données, on écrira :

```
for unJour in Jour loop
```

plutôt que

```
for unJour in lundi..dimanche loop
```

De cette manière, il sera toujours possible de modifier les valeurs du type sans que le reste du programme en soit affecté. On pourrait, par exemple angliciser les valeurs du type `Jour (monday, tuesday, ...)` sans avoir à modifier la boucle.

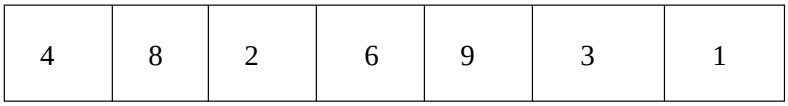
Remarque : La qualification est utilisée pour ne pas confondre la valeur dimanche du type `Jour` et celle du type `WE`.

```
vendredi..Jour'(dimanche)
```

2.14 Type tableau

Pour pouvoir écrire des programmes un peu plus intéressants, il est nécessaire d'appliquer des traitements à des séquences de données. Une manière classique est de les rassembler dans un tableau.

Dans l'exemple qui suit, on réunit dans un tableau le nombre d'heures d'ensoleillement de chaque jour de la semaine. On cherche ensuite quelle est la journée la plus ensoleillée puis on l'affiche à l'écran.



Un tableau est donc une structure de données qui réunit des valeurs (données) d'un certain type (le type `Integer` dans l'exemple qui suit). On peut le voir comme une suite de cases repérées (indicées) par une valeur d'un autre type (ici le type `Jour`).

Un tableau constitue une nouvelle valeur. Or toute valeur doit `appartenir` à un type. Il est donc nécessaire de définir un nouveau type auquel ces éléments (ces valeurs) appartiendront.

Dans l'exemple qui suit, on définit le type `Soleil` comme un ensemble de tableaux contenant des entiers (du type `Integer`) indicés par des valeurs du type `Jour`.

Il est possible alors de déclarer une variable (`uneSemaine`) de ce nouveau type, de rentrer des valeurs dans les cases de ce tableau, de sélectionner une case connaissant son indice (`uneSemaine(i)`).

```
-- exemple 13
-- nombre d'heures d'ensoleillement journalier
-- d'une semaine donnée
with Ada.Text_io;use Ada.Text_io;
with Ada.Integer_Text_io;
procedure exemple13 is
 type Jour is (lundi,mardi,mercredi,jeudi,vendredi,samedi,dimanche);
 type Soleil is array(Jour) of Integer;
 uneSemaine:Soleil ;
 nbHeures:Integer;
 unJour:Jour;
begin
-- construction du tableau uneSemaine
 for i in Jour loop
   put("taper le nb d'heures de soleil");
   put(" pour la journee de ");
   put(Jour'image(i));
   put(" : ");
   Ada.Integer_Text_io.get(uneSemaine(i));
   new_line;
 end loop;
-- calcul du jour le plus ensoleille
 nbHeures:=uneSemaine(lundi);
 unJour:=lundi;
 for x in mardi..dimanche loop
   if uneSemaine(x)>nbHeures
   then
     nbHeures:=uneSemaine(x);
     unJour:=x;
   end if;
 end loop;
-- affichage du résultat
 put("le jour le plus ensoleille de la semaine fut ");
 put(Jour'image(unJour));
 put(" avec ");
 Ada.Integer_Text_io.put(uneSemaine(unJour),WIDTH=>2);
 put_line(" heures d'ensoleillement");
end exemple13;
```

Chapter 3

Types

3.1 Définition

Un type est un ensemble de valeurs et un ensemble d'opérations pouvant être effectuées sur ces valeurs.

3.2 Notion d'objet

Toute entité à laquelle un type est associé (variable, constante, fonction, procédure, exception, paquetage, ...) est un objet informatique.

3.3 Notion de type

3.3.1 Intérêt du concept de type

Comment fournir à la machine les informations nécessaires pour qu'elle soit en mesure de vérifier que l'utilisation des objets d'un programme est bien conforme à l'intention préalable du programmeur ?

Lorsque nous déclarons un objet (variable, constante, fonction, ...), nous spécifions les contraintes de son utilisation, l'ensemble des valeurs qui pourront lui être associées et ainsi définir le cadre légitime de son utilisation.

L'utilisation de cet objet sera précédée d'une vérification effectuée par le compilateur. La valeur qu'il prend appartient-elle à cet ensemble? Son utilisation est-elle conforme aux contraintes spécifiées?

Le concept de type permet d'établir un lien sémantique entre la déclaration et l'utilisation d'un objet.

3.3.2 Ada et le concept de type

Tout objet est introduit dans un programme au moyen d'une *déclaration.* Une déclaration permet d'associer un type à un objet. Toute expression du langage est donc typée.

En Ada, tout objet, toute expression possède *un type et un seul.* On dit qu'Ada est un langage *fortement typé.*

Il existe des types prédéfinis (`Integer, Character, Float, Boolean`).

Il appartient au programmeur, pour les besoins de son application, de définir ses propres objets. Par exemple, si l'on considère l'application de gestion de déclaration des impôts sur le revenu, la feuille de déclaration devient un objet central de cette application. Pour autant, aucun des types prédéfinis ne correspond à un tel objet. Il est donc indispensable pour le programmeur d'avoir la possibilité de définir lui-même des types de données.

Le langage Ada offre de riches possibilités de construction de types spécifiques.

3.4 Types prédéfinis

3.4.1 Le type `Integer`

Ensemble des valeurs

Les valeurs du type `Integer` forment un sous ensemble de l'ensemble Z, symétrique par rapport à 0.

L'ensemble des valeurs du type `Integer` dépend de la machine. Pour une machine dont les mots sont de 32 bits, les valeurs de l'ensemble `Integer` s'étalent entre -2^{31} et $+2^{31-1}$.

Pour des raisons de portabilité des programmes, c'est à dire pour faire en sorte que les programmes soit indépendants de la taille des mots de la machine sur laquelle ils doivent s'exécuter, Ada a introduit les attributs `first` et `last`.

Pour le programmeur le plus petit entier est `Integer'first,` le plus `grand est Integer'last.` Pour une machine 32 bits, `Integer'first=-2147483648 et Integer'last=+2147483647.`

L'ensemble des valeurs du type `Integer` est donc défini par l'intervalle :

```
Integer'first..Integer'last
```

Une installation peut offrir les types prédéfinis : `Short_Integer` et `Long_Integer` pour coder les entiers sur, respectivement, 1/2 mot ou 2 mots.

Ensemble d'opération

opérations	symboles
affectation	:=
opérateurs relationnels	<, <=, >=, >
opérateurs d'égalité	=, /=
opérateurs unaires	+, -, abs
opérateurs binaires	+, -, *, /, **, mod, rem

Les opérateurs sont munis des priorités habituelles.

- Opérateur `mod` : le signe du résultat est celui du deuxième opérande

```
-9 mod 2 = -(9 mod 2) = -1
(-9) mod 2 = 1
9 mod 2 = 1
9 mod (-2) = -1
```

- Opérateur `rem` : le signe du résultat est celui du premier opérande

```
-9 rem 2 = -(9 rem 2) = -(+1) = -1
(-9) rem 2 = -1
9 rem 2 = +1
9 rem (-2) = +1
```

3.4.2 Autres type entiers

Ils sont obtenus en réduisant l'intervalle des valeurs possibles. On définit ainsi des sous types d'entiers.

L'ensemble des opérations reste le même.

Exemple

```
subtype Entier is Integer range 1..200;
```

3.4.3 Types énumératifs

Ensemble de valeurs

C'est un ensemble ordonné d'éléments identificateurs et/ou littéraux

Exemples

```
type Voyelle is (a,e,i,o,u,y);
-- valide car les valeurs sont considérées
-- comme des identificateurs
type Voyellebis is ('a','e','i','o','u','y');
-- valide car les valeurs sont considérées
-- comme des littéraux
-- les quotes seront saisies en entrée
-- et reproduites à l'impression
type Reponse is (oui,non,peut_etre);
```

Opérations sur les types énumératifs

opérations	symboles
affectation	:=
opérateurs relationnels	<, <=, >=, >
opérateurs d'égalité	=, /=

- Relation d'ordre sur les valeurs

```
oui<non<peut_etre
a<e<i<o<u<y
```

- Attributs

```
Voyelle'first=a
Voyelle'last=y
Voyelle'succ(a)=e
Voyelle'pred(e)=a
Voyelle'pos(a)=0
Voyelle'val(1)=e
Reponse'image(oui)="oui"
```

Le successeur de `y` n'existe pas. Le prédecesseur de `oui` non plus.

Remarque : on notera la différence entre l'identificateur `a`, le caractère `'a'` et la chaine de caractère `"a"`.

3.4.4 Le type `Boolean`

Ensemble de valeurs

```
FALSE, TRUE
```

- Relation d'ordre sur les valeurs

```
FALSE<TRUE
```

Opérations sur les types `Boolean`

opérations	symboles
affectation	:=
opérateurs relationnels	<, <=, >=, >
opérateurs d'égalité	=, /=
opérateurs logiques	not, and, or, xor
opérateurs logiques spécifiques	and then, or else

Tables de vérité de opérateurs booléens

and	**false**	**true**
false	false	false
true	false	true

or	**false**	**true**
false	false	true
true	true	true

xor	false	true
false	false	true
true	true	false

Exemples

```
('a'<'y' and 'B'>'X') = FALSE
((X mod 400=0) or (X mod 4=0 and X mod 100/=0))=TRUE
-- si X représente une année bissextile
```

Exemples

```
and then : (A/=0 and then B/A=X)
```

Contrairement à l'évaluation de l'opérateur `and`, Ada évalue d'abord l'expression `A/=0`. Si la réponse est `FALSE`, l'expression `B/A=X` n'est pas évaluée, ce qui protège d'une éventuelle division par `0` qui entrainerait une erreur à l'exécution.

```
or else : (A=0 or else B/A=X)
```

L'idée est la même. Si l'expression `A=0` vaut `TRUE` il est inutile d'évaluer `B/A=X`. On évite aussi, dans ce cas, la division par `0`.

3.4.5 Le type `Character`

Il s'agit d'un type énuméré où chaque valeur est un caractère.

Ensemble de valeurs

en Ada 83, c'est l'ensemble des caractères définis par le code ASCII 7 bits, c'est à dire les caractères usuels à l'exclusion des caractères accentués.

Opérations sur le type `Character`

opérations	symboles
affectation	:=
opérateurs relationnels	<, <=, >=, >
opérateurs d'égalité	=, /=

3.4.6 Le type `Float`

Il existe 3 manières de représenter des nombres réels en Ada :

1. les types `Float` permettent de spécifier le nombre de décimaux significatifs, mais ne garantissent pas un degré de précision sur l'ensemble des valeurs
2. les types `Fixed` maintiennent un degré de précision quelle que soit la valeur

3. les types `Decimal` combinent les deux exigences précédentes

Le plus communément utilisé est le type `Float`, mais les exigences de précision des applications scientifiques sont satisfaites par les différents choix offerts par Ada.

Seul le type `Float` sera utilisé dans ce cours.

Ensemble de valeurs

Il s'agit du sous-ensemble de l'ensemble R des nombres réels (intervalle $[-2^{31}; 2^{31} - 1]$). On peut écrire des littéraux réels selon 2 notations :

```
123.4 ou 1.234E+2
```

Opérations sur le type `Float`

opérations	symboles
affectation	:=
opérateurs relationnels	<, <=, >=, >
opérateurs d'égalité	=, /=
opérateurs arithmétiques	+, -, *, / , abs

Remarques

Les conséquences du typage fort interdisent des opérations mixant le type `Float` et un type entier.

```
i:Integer:=2;
r:Float:=5.6;
...
if r=i then ... -- opération illicite
else ...
end if;
```

De même, il faut se méfier des comparaisons entre 2 valeurs flottantes. En effet, les réels sont représentés par une approximation binaire sur un nombre limité de bits.

```
r1:Float:=3.456387;
r2:Float:=3.456386999;
...
if r1=r2 then ...
else ...
end if;
```

On pourra constater que, bien que les valeurs de `r1` et `r2` soient différentes, la comparaison `r1=r2` sera évaluée à `TRUE` par Ada.

le type `Float` est accompagné de deux attributs qui délimitent l'intervalle de ses valeurs :

```
Float'first..Float'last
```

3.5 Types composés : les tableaux

3.5.1 Types discrets

On parle de types discrets pour des types qui définissent des ensembles de valeurs élémentaires *énumérables* et de plus, totalement *ordonnées.*

Des opérateurs prédéfinis, appelés attributs, sont applicables sur les valeurs de ces types : `first, last, succ, pred, pos, val, value, image.`

Le type `Jour`, vu au chapitre précédent, est un exemple typique de type discret, l'effet de chacun des attributs est décrit dans le tableau suivant :

attribut	valeur	commentaire
Jour'first	lundi	1ère valeur du type
Jour'last	dimanche	derniere valeur du type
Jour'succ(mardi)	mercredi	valeur suivante selon l'ordre d'énumération
Jour'pred(mardi)	lundi	valeur précédente selon l'ordre d'énumération
Jour'pos(mardi)	1	rang dans l'ordre des valeurs
Jour'val(5)	samedi	valeur de rang 5
Jour'value("mardi")	mardi	symbole correspondant à la chaine "mardi"
Jour'image(mardi)	"mardi"	chaine correspondant au symbole mardi

Ada fournit 3 types discrets prédéfinis :

```
Integer
Boolean
Character
```

3.5.2 Type tableau contraint

Définition Un tableau Ada représente un n-uplet dont toutes les composantes sont de même type.

Un n-uplet est un élément d'un produit cartésien de n ensembles identiques (les composantes) :

$\Pi^n A = A * A * A * ... * A = \{(a_1, a_2, ..., a_n) | a_i \in A\}$

Le nombre de composantes d'un tableau est appelé *dimension.*

La position d'un élément dans une composante d'un tableau est appelée *indice.*

La valeur d'un *indice* doit appartenir à un type discret.

Le type tableau représente l'ensemble des tableaux :

1. de même dimension
2. dont les composantes ont le même type
3. dont les indices sont définis dans un intervalle discret

Syntaxe de la définition d'un type tableau

```
<définition_type_tableau>::=
array(<borne_inf>..<borne_sup>) of <type_composante>|
array(<type_indices>) of <type_composant>|
array(<type_indices> range <borne_inf>..<borne_sup>)
                                        of <type_composant>
```

où

<type_indice> représente le type des indices. *<type_indice>* est un type discret.

<borne_inf> et *<borne_sup>* représentent les bornes inférieures et supérieures d'un type d'indices.

<type_composante> représente le type des composantes du tableau. *<type_composante>* est un type quelconque.

Exemples

```
array(lundi..vendredi) of Visite;
-- l'intervalle lundi..vendredi appartient au type discret Jour
-- le type Visite est un type quelconque que l'on suppose défini
array(Jour) of Visite;
array(Integer range 1..7) of Matiere;
```

3.5.3 Ensemble de valeurs, ensemble d'opérations

Ensemble des valeurs

Les valeurs d'un type tableau contraint sont des objets tableaux.

Ce sont des objets composés d'éléments de même type et dont les bornes sont identiques.

Ensemble des opérations

opérations	symboles
affectation	:=
opérateurs relationnels	<, <=, >=, >
opérateur de concaténation	&
opérateur de sélection	( )

3.5.4 Déclaration d'un type tableau

La déclaration d'un type tableau associe un nom à la définition du type.

```
<déclaration_type_tableau>::=
              type <ident_du_type> is <définition_type_tableau>;
```

avec

`<ident_du_type>` est un identificateur Ada.

Exemples

```
type Semaine is array Jour(lundi..vendredi)of Integer;
N:constant Integer:=50;
type Tab is array(10..N) of Integer;
type Heure is (10_12,14_16,16_18,18_20);
type Motif is (reunion,visite,pot,rendez_vous);
type Agenda is array(Heure,Jour) of Motif;
```

3.5.5 Types tableaux anonymes

Il est possible de déclarer des objets tableaux dont le type n'a pas été défini.

La déclaration associe alors directement un identificateur d'objet tableau à la définition de son type.

```
X,Y: array('A'..'Z') of Integer;
-- il s'agit alors de deux définitions
-- différentes, x et y ont deux types distincts
```

3.5.6 Déclaration de variables d'un type tableau

Une déclaration d'un objet de type tableau associe un nom de variable à un type tableau.

- Déclaration de variables d'un type tableau anonyme :

```
hebdo1,hebdo2:array(vendredi..dimanche) of Integer;
--Les variables hebdo1 et hebdo2 sont de type différent
```

- Déclaration de variables d'un type tableau explicitement déclaré

```
hebdo3,hebdo4:Semaine;
-- Les variables hebdo3 et hebdo4 sont de même type
A,B:Tab;
conge:Semaine;
C:Agenda;
```

3.5.7 Agrégats

Un *agrégat* est une valeur d'un type tableau. C'est une liste d'associations entre un indice et une valeur.

Il peut être construit syntaxiquement de 3 manières différentes :

- Association implicite (d'après l'ordre de lecture)

```
(9,9,9,11,19)
--valeur de tableau du type Semaine
```

- Association explicite (l'indice est explicite). Cette notation dispense de respecter l'ordre des indices

```
(jeudi=>11,lundi=>9,mardi=>9,vendredi=>7,
                    mercredi=>9,samedi=>7,dimanche=>7)
```

ou bien

```
(lundi..mercredi=>9,jeudi=>11,others=>7)
```

ou bien

```
(lundi|mardi|mercredi=>9,jeudi=11,others=>7)
-- la construction others permet de compléter l'initialisation des
-- champs qui suivent selon l'ordre
-- la construction a..b  ou a|b permet de factoriser
-- l'initialisation de plusieurs champs successifs
```

- Panachage

```
(9,9,mercredi=>9,11,others=>7)
-- l'ordre est respecté avant l'association explicite
```

3.5.8 Initialisation de variable d'un type tableau

La valeur d'initialisation d'un tableau est un agrégat :

```
RDV:array(lundi..mercredi) of Integer:=(9,13,10);
```

3.5.9 Affectation d'une variable tableau

```
type Vecteur5 is array(1..5) of Float;
monVecteur:Vecteur5;
monVecteur:=(1.0,2..3=>2.0,others=>0.0);
```

3.5.10 Sélection d'un composant de tableau

Pour sélectionner le composant d'un tableau, il suffit de préciser son indice.

```
monVecteur(2):=2.5;
-- la valeur du second composant est modifiée
monVecteur(3):=monVecteur(1);
-- le 3ème composant prend la valeur du premier
-- Les types des expressions droite et gauche sont les mêmes
hebdo1,hebdo2: array(vendredi..dimanche) of Integer;
hebdo1:=(8,7,14);
hebdo2:=(6,5,7);
hebdo1:=hebdo2;
-- affectation illégale
hebdo1(vendredi):=hebdo2(samedi);
-- affectation légale car il s'agit d'une affectation d'un entier à
-- une variable entière
```

3.5.11 Sélection d'une tranche de tableau (sous-tableau)

Pour sélectionner un sous-tableau (suite de composants contigus), il suffit de préciser son intervalle d'indice :

```
hebdo1(samedi..dimanche):=(12,12);
monVecteur(1..3):=monVecteur(3..5);
-- les deux sous-tableaux sont de même dimension
-- leurs composants sont de même type
```

3.5.12 Concaténation

L'opérateur `&` permet de concaténer 2 tableaux (et/ou sous- tableaux) dont les composantes sont de même type :

```
hebdo3,hebdo4,hebdo5:Semaine;
hebdo3:=(9,8,others=>16);
hebdo4:=(9,others=>10);
hebdo5:=hebdo3(lundi) & hebdo4(jeudi..vendredi) & hebdo3(mardi) & hebdo4(mardi);
```

3.5.13 Attributs

Soit la déclaration :

```
tab:array (5..100) of
          Character:=('a','b','c','d','e',others=>'x');
```

Les attributs applicables au tableau `tab` sont :

expression	valeur	signification
`tab'first`	5	borne inférieure
`tab'last`	100	borne supérieure
`tab'length`	95	longueur de l'intervalle d'indices
`tab'range`	5..100	l'intervalle d'indices

3.5.14 Types tableaux non contraints

Pour généraliser les types tableaux, il est possible de les paramétrer en évitant de borner l'intervalle des indices.

Construction syntaxique

```
<définition_type_tableau_non_contraint>::=
     array (<type_des_indices> range <>) of <type_des_composantes>;
```

Le type des indices doit être discret.

La construction `range <>` indique que la taille des tableaux ne sera connue que lorsqu'une valeur aura été spécifiée pour le `<type_des_indices>`.

Exemple

```
type Vecteur is array(Integer range <>) of Jour;
vecteur_1:Vecteur(1..6);-- l'intervalle des indices est fixé
vecteur_2:Vecteur(1..3);
```

Remarque Le système ne peut pas allouer l'espace mémoire nécessaire à une variable dont le type est non contraint car il ne connait pas sa taille.

3.5.15 Exemple

Calcul de la moyenne des valeurs d'un tableau de réels :

```
-- exemple 15
declare
 type Vecteur_reels is array (Integer range <>) of Float;
 V:Vecteur_reels(1..5):=(3.47,8.7,5.32,1.09,0.004);
 somme:Float:=0.0;
begin
 if V'length=0 then
   put("le vecteur est vide");
 else
   for I in V'range loop
     somme:=somme+V(I);
   end loop;
   put("Moyenne des composants :");
   put(somme/float(V'last-V'first+1));
 end if;
end;
```

Remarques

- Déclaration d'un type tableau non contraint
- Utilisation des attributs : `length, range, first, last`
- Conversion de type : `/` est un opérateur de type : `Float*Float-->Float`. Il est donc nécessaire de convertir la valeur entière `V'last-V'first+1` en une valeur réelle. C'est le rôle de la fonction `Float(...)`.

3.5.16 Tableaux dynamiques

Un tableau dynamique est un tableau dont les bornes sont évaluées au cours de l'exécution.

Exemple

```
-- exemple 14
procedure dynamique is
  type TabDyn is array(Integer range <>) of Float;
  N:Integer;
begin
  get(N);
  declare
```

```
    unTabDyn:TabDyn(1..N):=(others=>0.0);
    -- N n'est pas connu à la compilation
  begin
    ...
  end;
end dynamique;
```

3.5.17 Type `String`

Il définit l'ensemble des chaines de caractères de longueur quelconque. C'est un type non contraint prédéfini.

```
type String is array(Positive range <>) of Character;
ligne:String(1..80);
--représente une chaine de 80 caractères
```

Lecture d'une chaine de caractères de longueur variable

```
with Ada.Text_io;
use Ada.Text_io;
procedure chaine is
  ligne:String(1..80);
  -- on suppose une longueur maximum de la chaine de 80 caractères
  lg:Natural range 0..80;
begin
  put("Tapez votre nom : ");
  get_line(ligne,lg);
  -- lg contient le nombre de caractères effectivement saisis
  put("ligne= ");
  declare
    nom:String(1..lg);-- le type String est contraint
  begin
    put("nom=");
    put(nom);
  end;
end chaine;
```

résultat

```
Tapez votre nom : Daniel
ligne=DanielèT‰3 (ôôô%êxõõÃ‰ÛÛ%{êxõ%]
nom=Daniel
```

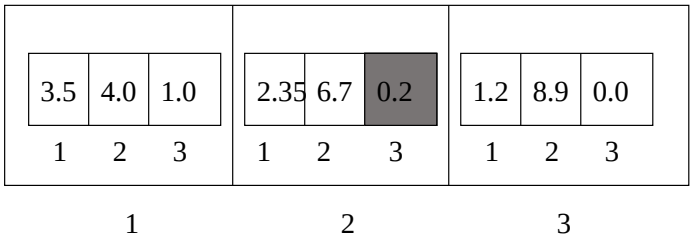
3.5.18 Tableaux multidimensionnels

Le type des composants peut être quelconque, il peut donc être de type tableau.

On peut donc définir des tableaux de tableaux ou bien des tableaux à plusieurs dimensions

Tableaux de tableaux Le type des composants est un tableau :

```
type M3 is array(1..3) of Vecteur(1..3);
mat_1:M3:=((3.5,4.0,1.0),(2.35,6.7,0.2),(1.2,8.9,0.0));
Ada.Float_Text_io.put(mat_1(2)(3));
-- affiche 0.2
```



Tableaux à 2 dimensions On indique les deux intervalles d'indices :

```
type M3_BIS is array(1..3,1..3)of Float;
mat_2:M3_BIS:=((3.5,4.0,1.0),(2.35,6.7,0.2),(1.2,8.9,0.0));
Ada.Float_Text_io.put.mat_2(2,3);
-- affiche 0.2
```

	1	2	3
1	3.5	4.0	1.0
2	2.35	6.7	0.2
3	1.2	8.9	0.0

Remarques Les agrégats sont les mêmes quels que soient le style de définition adopté
Il est possible de définir une tranche de tableaux de tableaux :

```
mat_1(2)(1..3)
mat_1(1..2)(2)
```

Exemple

```
declare
  type Vecteur is array(1..3) of Integer;
  type Matrice is array(1..3) of Vecteur;
  M:Matrice:=((6,7,8),(1,2,3),(4,5,6));
begin
  put("M(1..2)(2)=");
  for i in 1..2 loop
    put(M(i)(2),width=>2);
  end loop;
end;
```

résultat

```
M(1..2)(2)= 7 2
```

Il est illégal de définir une tranche d'un tableau à plusieurs dimensions

```
mat_2(2..3) -- illégal
mat_2(2,1..3) -- illégal
```

3.5.19 Convertibilité

Deux types tableaux sont convertibles s'ils ont :

1. même type d'éléments
2. même nombre de dimensions
3. mêmes types d'indices

Exemple

```
type Tab1 is array(1..3) of Jour;
type Tab2 is array(5..7) of Jour;
mon_t1:Tab1:=(lundi,mardi,mercredi);
mon_t2:Tab2:=(vendredi,samedi,dimanche);
mon_t1:=Tab1(mon_t2);
mon_t2:=Tab2(mon_t1);
```

3.6 Types composés : les articles

Les tableaux ne permettent pas d'associer des données de types différents. La notion d'article permet de rassembler plusieurs valeurs de types quelconques caractérisant une donnée particulière.

3.6.1 But

- Représenter des données complexes : outil de modélisation des données d'une application
- Manipuler globalement une donnée complexe tout en ayant accès à ses parties : possibilité de passer une donnée complexe en paramètre, de l'utiliser comme valeur de retour d'une fonction, de l'affecter à une variable.

3.6.2 Exemple

Supposons que le type `Date` est défini. Il associe les 3 valeurs : `jour, mois, an`

```
naissance: Date; -- déclaration d'une variable
function quantieme(une_date:Date) return Natural;
-- utilisation en tant que paramètre d'une fonction
function fabique_date(jj,mm,aa:Natural) return Date;
-- utilisation comme valeur résultat d'une fonction
```

3.6.3 Définition d'un type article

Un type article Ada correspond à un produit cartésien d'ensembles différents.

Construction syntaxique

```
<définition_type_article>::=
record
  <ident_champ>:<type>[:=<expression>];
  {<ident_champ>:<type>[:=<expression>];}
end record;
```

Le type du champ d'un article peut être quelconque et optionnellement posséder une valeur par défaut.

3.6.4 Déclaration d'un type article

Construction syntaxique

```
<déclaration_type_article>::=
type <ident_type> is <définition_type_article>;
```

Exemple

```
type Date is
   record
     jour: Natural;
     mois: Natural;
     an: Natural;
   end record;
```

3.6.5 Valeur d'un type article : agrégat

- Par association implicite champ-valeur

  ```
  (val_1,val_2,...,val_n)
  ```

- Par association explicite champ-valeur

  ```
  (champ1=>val_1,champ2=>val_2,....,champn=>val_n)
  ```

- En panachant

  ```
  (val_1,val_2,champ3=>val_3 ,....,champn=>val_n)
  ```

Exemples

```
naissance:Date:=(1,12,1985);
une_date:Date:=(jour=>28,mois=>03,an=>1989);
```

ou

```
une_date:Date:= (24,mois=>11,1989);
aujourd_hui:Date:=(mois=>11,an=>2002,jour=>14);
-- dans ce cas, on ne peut pas panacher
```

3.6.6 Exemple

```
-- exemple 16
constant Float SMIC:= 5997.41;
type Employe is
   record
     nom:String(1..10);
     naissance:Date;
     salaire:Float:=SMIC;
   end record;
directeur:Employe:=("Patron    ",(5,8,1954),20045.);
```

3.6.7 Opération d'accès à un champ : .

Syntaxe

<opération_accès_champ>::= <ident_article>.<ident_champ>

Exemples

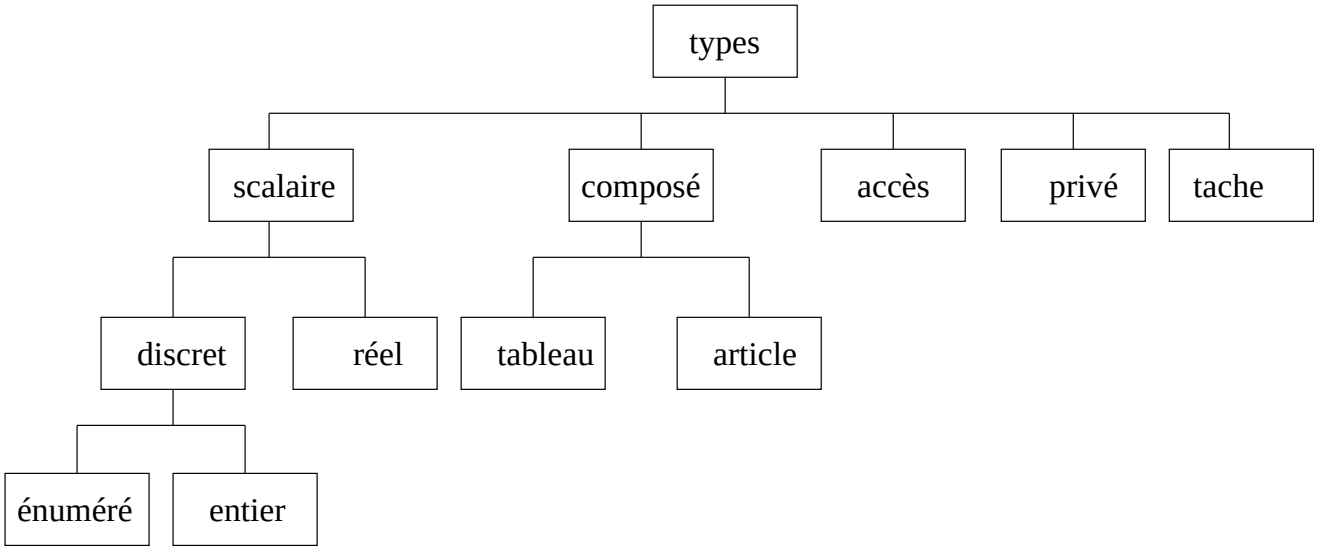
```
directeur.naissance.an
-- représente l'année de naissance de directeur
Ada.Text_io.put(directeur.salaire);
-- affiche le salaire du directeur
if directeur.naissance.an<1900
then
  put("C'est l'heure de la retraite");
else
  put("Toujours vaillant!!");
end if;
```

3.6.8 Classification des types

Les types *scalaires* correspondent à des ensembles de valeurs atomiques (non composées).

Les types *discrets* sont des ensembles de valeurs ordonnées et enumérables (on peut associer un entier à chacune des valeurs).

cf figure 3.3


types
scalaire
composé
accès
privé
tache
discret
réel
tableau
article
énuméré
entier

Chapter 4

Les instructions

4.1 Affectation

Les machines actuelles, dans leur forme la plus rudimentaire, s'apparentent à un modèle qui date de la fin des années 40, le modèle Von Neumann. Selon ce modèle, un ordinateur est constitué de 3 parties :

- une unité centrale de traitement (CPU)
- une mémoire
- un tube qui connecte CPU et mémoire et permet de transmettre des mots de l'un vers l'autre.

Ce tube est appelé le "*Von Neumann bottleneck*" (goulot d'étranglement). En effet, globalement, la tâche d'un programme est de changer le contenu de la mémoire, passant d'un état initial à un état final représentant le résultat. Cette tâche est entièrement accomplie en faisant transiter des données (des mots) à travers ce tube, d'où la dénomination de goulot d'étranglement. Une grande partie de ce trafic ne concerne d'ailleurs pas les données elles-mêmes mais leur adresse dans la mémoire.

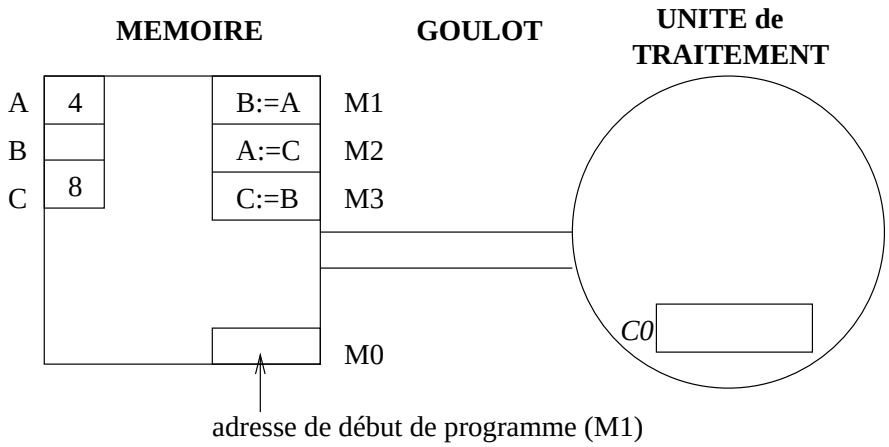
Ainsi, l'activité de programmation revient, in fine, à la mise en ordre et au détaillage de l'énorme trafic des mots à travers le goulot.

Exemple

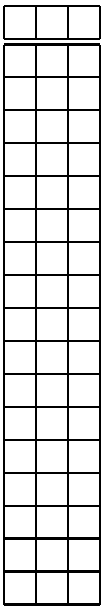
Il s'agit de permuter 2 valeurs 4 et 8 dont les noms en mémoire sont, à l'instant initial A et C. La mémoire contient le programme qui réalise cette permutation :

```
B:=A;
A:=C;
C:=B;
```

La figure qui suit traduit l'image initiale de la machine :



Dans le tableau qui suit, on détaille la suite des actions et des passages dans le goulot provoquée par l'exécution du programme :



actions	nb de passages	commentaires
M0->C0	1	l'adresse du début de programme est chargée dans le compteur ordinal
C0->M1	1	transfert de l'adresse de la 1ère instruction
B:=A->UAL	1	transfert de l'instruction dans l'UAL
UAL->A	1	transfert de l'adresse A
4->UAL	1	transfert de la valeur 4 dans l'UAL
UAL->B	1	transfert de la valeur 4 dans B
C0->M2	1	transfert de l'adresse de la 2ème instruction
A:=C->UAL	1	transfert de l'instruction dans l'UAL
UAL->C	1	transfert de l'adresse C
8->UAL	1	transfert de la valeur 8
UAL->A	1	transfert de la valeur 8 dans A
C0->M3	1	transfert de l'adresse de la 3ème instruction
C:=B->UAL	1	transfert de l'instruction
UAL->B	1	transfert de l'adresse B
val(B)->UAL	1	transfert de la valeur de B dans l(UAL
UAL->C	1	transfert de 4 dans C
	total=16	

Les cellules mémoires nommées `A, B, C, M0, M1`, etc... sont modélisées dans les langages de programmation par des *variables informatiques*. Une variable informatique prend donc des valeurs différentes au cours de l'exécution d'un programme.

On notera la différence entre une variable informatique et une variable au sens mathématique qui n'est qu'une entité abstraite ne pouvant désigner qu'une unique valeur.

Le moyen d'"imiter" les actions d'"aller chercher dans la mémoire" et de "mettre en mémoire" est donné par l'**instruction d'affectation** notée `:=` en Ada (comme dans l'exemple précédent).

C'est l'instruction qui sert essentiellement à modifier la valeur d'une variable.

4.1.1 Syntaxe

<affectation>::=<expression_gauche>:=<expression droite>;

<expression gauche> dénote une adresse

<expression droite> dénote une valeur

On notera que l'instruction d'affectation (ainsi que toute autre instruction) se termine par ";";.

Exemple

```
X:=Y+2;
X:=X/4;
--on va chercher la valeur de X en mémoire, puis on la divise par 4
--dans l'UAL, puis on transfert cette valeur à l'adresse notée
--symboliquement X
```

4.1.2 Sémantique

La valeur de la partie droite est enregistrée à l'adresse de la variable en partie gauche.

L'affectation sépare la programmation en 2 mondes. Le premier, concerné par la partie droite de l'affectation, est un monde d'expressions avec leurs propriétés algébriques, un monde dans lequel ont lieu les calculs. Le second, concerné par la partie gauche est un monde d'adresses permettant de localiser des données (valeurs ou instructions) dans la mémoire.

4.2 Séquence

L'idée imposée par le modèle Von Neumann de penser un programme en terme de trafic à travers le goulot amène à organiser séquentiellement la suite d'affectations.

en Ada, une séquence d'instructions est obtenue par simple juxtaposition textuelle de plusieurs instructions.

4.3 Entrées/Sorties

Un autre moyen de modifier la valeur d'une variable est réalisé par la lecture à partir d'un support externe par la commande :

```
get(X);
```

où `X` dénote une adresse

La commande :

```
put(expression);
```

évalue l'expression et réalise une affectation de cette valeur sur le fichier standard de sortie qui est l'écran. L'affichage de la valeur peut aussi s'appliquer à un fichier de sortie quelconque.

On parle ici de commandes et non d'instructions. En effet, elles ne sont pas partie intégrante du langage mais fournies par une bibliothèque.

4.4 Structures de contrôle

Les instructions dont la seule qui a des effets sur le contenu de la mémoire est `l'affectation`. Les autres instructions, que nous allons détailler dans les paragraphes suivants, ne sont là que pour permettre d'organiser de manière à la fois plus concise et plus lisible la suite des affectations. On appelle ces instructions des structures de contrôle.

On distingue les structures de contrôle *conditionnelles* et *itératives*.

4.5 Conditionnelle

L'instruction alternative ou conditionnelle permet de spécifier la prochaine instruction à exécuter selon l'évaluation d'une expression booléenne.

4.5.1 Syntaxe

```
<conditionnelle>::=
if <expression_booléenne>
then
  <suite_instructions>
[elsif <expression_booléenne>
    then
       <suite_instructions>
```

```
    elsif <expression_booléenne>
       then
          <suite_instructions>
...]
[else
  <suite_instructions>]
end if;
```

4.5.2 Sémantique

- Les parties entre crochets sont optionnelles
- Les conditions sont évaluées en séquence
- Si une condition est vraie, la suite d'instructions associée est exécutée et l'instruction conditionnelle est terminée
- Si toutes les conditions sont fausses, la partie `else` est exécutée et l'instruction conditionnelle est terminée

4.5.3 Exemple

```
if A=0.0
then
-- calcul du cas linéaire
elsif B**2-4.0*A*C>=0.0
   then
   -- calcul des racines réelles
   else
   -- calcul des racines complexes
end if;
```

4.6 Choix multiple

L'alternative peut aussi être posée à partir d'un choix parmi l'ensemble des valeurs d'un type discret.

4.6.1 Syntaxe

```
<choix>::=
case <expression> is
  when <valeur> => <suite_instructions>;
  when <valeur>|<valeur> => <suite_instructions>;
  when <valeur>..<valeur> =><suite_instructions>;
  when others => <suite_instructions>;
end case;
```

4.6.2 Sémantique

- *<expression>* est évaluée. Sa valeur appartient à un type discret
- Les valeurs utilisées en tant que filtre sont comparées de haut en bas à la valeur de *<expression>*

- Dès qu'il y a égalité entre la valeur de l'expression et celle du filtre, la suite d'instructions correspondante est exécutée
- L'ensemble des filtres d'une instruction à choix multiple représente exhaustivement l'ensemble des valeurs du type de l'expression.
- Le filtre *others* capte les valeurs qui n'ont pas été citées en tant que filtre

4.6.3 Exemple

```
with Ada.Text_io;use Ada.Text_io;
with Ada.Integer_Text_io;use Ada.Integer_Text_io;
procedure calculette is
 type Operateur is (add,mult,sous,div);
 package op_io is new Enumeration_io(Operateur);
 use op_io;
 operation:Operateur;
 X,Y,R:Integer;
begin
 get(operation);get(X);get(Y);
 case operation is
  when add  => R:=X+Y;
  when sous => R:=X-Y;
  when mult => R:=X*Y;
  when div  => R:=X/Y;
 end case;
 put(R);
end calculette;
```

4.7 Structure itérative générale

Les instructions itératives permettent de répéter un traitement décrit par une suite d'instructions tant qu'une décision d'interruption de l'itération n'est pas rencontrée.

4.7.1 Syntaxe

```
<boucle>::=
loop
[<suite_instructions>;]
[exit when <condition_arrêt>;]
[<suite_instructions>;]
end loop;
```

4.7.2 Sémantique

- Les parties entre crochets sont facultatives
- La condition d'arrêt est une expression booléenne
- L'instruction *exit* permet de sortir directement de la boucle
- Dans le cas où les boucles sont imbriquées, *exit* permet de sortir vers le niveau immédiatement supérieur
- Une boucle sans condition d'arrêt est une boucle infinie

Exemple Une boucle peut répéter indéfiniment la même suite d'instructions.

```
with Ada.Text_io;use Ada.Text_io;
procedure infinie is
begin
  loop
    put("SOS");
  end loop;
end infinie;
```

Exemple

```
with Ada.Text_io;
procedure saisie is
  X:Integer;
begin
  Ada.Text_io.put("saisie d'entiers positifs");
loop
  Ada.Integer_Text_io.get(X);
exit when X<=0;
  Ada.Text_io.put("la valeur saisie est : ");
  Ada.Integer_Text_io.put(X);
 end loop;
 Ada.Text_io.put("saisie terminée");
end saisie;
```

Exemple

```
with Ada.Text_io;
procedure alpha is
  C:Character:='A';
begin
  Ada.Text_io.put("Caractères alphabétiques");
  loop
    Ada.Text_io.put(C);
  exit when C='Z';
    C:=Character'succ(C);
  end loop;
end alpha;
```

4.8 Boucle `while`

4.8.1 Syntaxe

```
<boucle_while>::=
while not <condition_arrêt> loop
  <suite_instructions>;
end loop;
```

avec

<condition_arrêt> est une expression booléenne.

Exemple Calcul de la racine carrée entière de `N`, `N>=0`

Si `x` est la solution du problème, alors : $x^2 \leq N < (x+1)^2$

```
with Ada.text_io,Ada.Integer_Text_io;
use Ada.text_io,Ada.Integer_Text_io;
procedure racine_carree is
  x:Natural:=0;
  y:Natural:=1;
  N:Natural;
  -- x ≤ N∧y = (x+1)^2
begin
  get(N);
  while N>=y loop
    x:=x+1;
    y:=y+2*x+1;
  end loop;
  put("la racine carrée entière de ");
  put(N);
  put(" est : ");
  put(x);
end racine_carree;
```

4.9 Boucle `for`

La boucle pour est une instruction itérative. Elle est utilisée pour répéter l'exécution d'un traitement un nombre de fois connu à l'avance.

4.9.1 Syntaxe

```
<boucle_for>::=
for <ident_de_compteur> in [reverse] <borne>..<borne> loop
   <suite_instructions>
end loop;
```

où

<borne> est une valeur d'un type discret représentant une borne de l'intervalle de valeurs que peut prendre *<ident_de_compteur>*.

reverse est facultatif. Il indique un parcours inverse (dans l'ordre décroissant) de l'intervalle *<borne>..<borne>*

4.9.2 Sémantique

Pour chaque valeur de l'intervalle discret parcouru dans l'ordre croissant (*in*) ou décroissant (*in reverse*) : *<suite_instructions>* est exécutée.

Exemple

```
for I in 1..3 loop
  U:=I*U;
end loop;
```

Exemple

```
with Ada.Text_io;
with Ada.Numerics.Discrete_Random;
with Ada.Integer_Text_io;
use Ada.Text_io, Ada.Integer_Text_io;

procedure loto is
  subType Numeros is Integer range 1..50;
  package tirage_loto is new
                  Ada.Numerics.Discrete_Random(Numeros);
  use tirage_loto;
  graine:Generator;
begin
  reset(graine);
  put_line("les numéros de la semaine !");
  for i in 1..6 loop
    put(Random(graine),WIDTH=>2);
    new_line;
  end loop;
end loto;
```

4.10 Structure de bloc

Ada est un langage dit "à structure de bloc" en ce sens qu'il permet d'associer un environnement spécifique à une suite d'instructions donnée. Un bloc correspond ainsi à une instruction composée de plusieurs instructions qui évolue dans un environnement propre. Un bloc est dès lors une instruction comme une autre.

4.10.1 Syntaxe

```
<bloc>::=
declare
<liste_déclarations>
begin
<suite_instructions>
[exception]
<traitement_exceptions>
end;
|
begin
<suite_instructions>
[exception]
<traitement_exceptions>
end;
```

Un bloc est une suite d'instructions auquel peut être attachée une partie déclarations et/ou une partie exception.

Les blocs peuvent s'emboiter :

```
declare
-- partie déclarations
begin
-- instructions
  declare
  -- partie déclarations
```

```
  begin
  -- partie instructions
  [exception]
  -- traitement des exceptions
  end;
[exception]
-- traitement des exceptions
end;
```

Les blocs ne peuvent pas se chevaucher.

Les blocs permettent de délimiter la portée des variables, c'est à dire leur domaine d'utilisation à l'intérieur d'un programme.

Remarque

Le corps des structures de contrôle forme un bloc avec une partie déclaration en général vide.

Exemple calcul du produit de 2 matrices : $C_{ij} = \sum_{k=1}^{p} A_{ik} * B_{kj}$

```
with Ada.Text_io;use Ada.Text_io;
with Ada.Float_Text_io;use Ada.Float_Text_io;

procedure produit_matrice is
  m:constant Positive:= -- un entierPositif ;
  n:constant Positive:= -- un entierPositif ;
  p:constant Positive:= -- un entierPositif ;
  type Matrice is
     array(Positive range <>,Positive range <>) of Float;
  A:Matrice(1..m,1..p):= (....); -- agrégat d'intialisation
  B:Matrice(1..p,1..n):= (....); -- agrégat d'intialisation
  C:Matrice(1..m,1..n);
begin
  -- calcul du produit des matrices A et B
  for i in 1..m loop
    for j in 1..n loop
      declare
        s:Float:=0.0;
      begin
        for k in 1..p loop
          s:=s+A(i,k)*B(k,j);
        end loop;
          C(i,j):=s;
      end;
    end loop;
  end loop;
  -- affichage du résultat
  for i in 1..m loop
    for j in 1..n loop
      put(C(i,j));
    end loop;
    new_line;
  end loop;
end produit_matrice;
```

Chapter 5

Fonctions

5.1 Fonctions

5.1.1 Fonctions en mathématiques

Définition Une fonction f définie sur A (domaine de définition, espace de départ) à valeurs dans B (codomaine, espace d'arrivée, espace image) est une correspondance qui, à tout élément x de A fait correspondre **un élément et un seul**, noté *f(x)*, de B. *f(x)* est appelé résultat de l'application de f à l'élément x.

5.1.2 Spécification

Une spécification est une description dans un langage formel (rigoureux) d'une correspondance entre des données et des résultats ainsi que des propriétés de cette correspondance.

5.1.3 Algorithme

Un algorithme est un cas particulier de spécification. C'est une spécification éxécutable.

Remarque Cette définition d'une spécification est proche de celle des fonctions.

5.2 Les fonctions en informatique

5.2.1 Terminologie

$f(x) = ax^2 + bx + c$

x est le **paramètre formel** de la fonction *f*.

a, b, c sont les **variables globales** (libres) de la fonction *f*.

L'expression (ici $ax^2 + bx + c$) définissant la fonction est appelée **corps de la fonction**.

Dans l'application de f à une valeur *v*, notée *f(v)*, v est appelé **paramètre effectif** ou argument.

5.2.2 Fonctions totales

En informatique, une fonction doit être totale (partout définie) => tout élément du domaine peut être **a priori** choisi comme argument.

Les cas particuliers doivent impérativement être traités de manière explicite.

5.3 Déclaration d'une fonction Ada

5.3.1 Type d'une fonction

```
T1*T2**Tn-->T
```

avec

`T1, T2, ..., Tn` : les types des paramètres formels

`T` : le type de la valeur retournée

5.3.2 Syntaxe

<déclaration_fonction>::=<en-tete_fonction> ;
<en-tete_fonction>::=
function <ident_fonction> <liste_paramètres>
return <ident_type>
<liste_paramètres>::=vide| (<déclar_paramètres>)
<déclar_paramètres>::=
<ident_param>:<type_param>[:=< *valeur* >]{,*<déclar_paramètres>*}
<valeur>::=une expression du type associé

Exemples de déclaration de deux fonctions :

```
function double(X:Integer) return Integer;
```

avec

`double`: identificateur de la fonction

`X` : paramètre formel

`Integer` : type du paramètre formel

`Integer` : type de la valeur retournée (résultat)

```
function max(A,B:Integer) return Integer;
```

5.3.3 Sémantique

La fonction dénotée par son identificateur appartient à l'environnement du programme.

A son identificateur est associé son type.

La valeur de la fonction, c'est à dire le code binaire exécutable qui lui est associé, n'est pas encore défini.

Le programmeur déclare ainsi les contraintes d'utilisation de l'objet fonction. Il spécifie ainsi un nouvel outil.

Le compilateur utilisera ces informations pour vérifier que l'utilisation qui est faite de cet objet fonction répond bien a l'intention du programmeur

5.3.4 Exemple

Les déclarations de fonctions suivantes enrichissent l'environnement du programme

```
function double(X:Integer) return Integer;
function max(A,B:Integer) return Integer;
```

A l'identificateur `double` est associé le type

```
Integer-->Integer
```

A l'identificateur `max` est associé le type

```
Integer*Integer-->Integer
```

5.3.5 Exemple

```
function perimetre(A:Float) return Float;
PI:constant Float:=3.14159;
```

2 objets sont introduits dans l'environnement du programme :

- La fonction d'identificateur `perimetre` avec son type : `Float-->Float`
- La constante d'identificateur `PI` avec son type : `Float`
- et sa valeur : `3.1459`

5.4 Déclaration du corps d'une fonction

5.4.1 Syntaxe

```
<declaration_corps_fonction>::=
<en-tete_fonction> is
   <déclaration_objets_locaux>
begin
   <corps_de_fonction>
end <ident_fonction> ;
```

Exemples

```
function max(A,B:Integer) return Integer is
begin
  if (A>B)
  then
     return A;
  else
     return B;
  end if;
end max;
```

```
function double(X:Integer) return Integer is
  begin return 2*X;
end double;
```

Déclaration de fonction locale à une autre fonction

```
function par_2(X:Integer) return Integer is
  function succ(N:Integer) return Integer is
  begin
    return (N+1);
  end succ;
begin
  return succ(succ(X));
end par_2;
```

Remarque L'identificateur peut être un opérateur. Il faut le mettre entre ""

```
declare
 type Complexe is
  record
    X,Y:float;
  end record;
 function "+"(A,B:Complexe ) return Complexe is
 begin
  return(A.X+B.X,A.Y+B.Y);
 end "+";
begin
...
end;
```

5.4.2 Sémantique

La valeur de la fonction est déterminée et associée à son identificateur. Cette valeur est le code binaire de la fonction associé au contexte courant. Elle est appelée *fermeture* car elle réunit en une seule structure le code binaire et le contexte de la déclaration du corps de la fonction.

Exemple

La déclaration du corps de `max`

```
function max(A,B:Integer) return Integer is
begin
  if (A>B)
  then
     return A;
  else
     return B;
  end if;
end max;
```

associe à l'identificateur `max` le code binaire de la fonction permettant de calculer le maximum de deux entiers et son contexte de déclaration.

La déclaration du corps de `double`

```
function double(X:Integer) return Integer is
begin
   return (2*X);
end double;
```

associe à l'identificateur `double` le code binaire de la fonction qui calcule le double d'un entier donné et son contexte de déclaration.

5.4.3 Exemple

La déclaration du corps de `perimetre`

```
function perimetre(R:Float) return Float is
begin
   return 2.0*PI*R;
end perimetre;
```

complète l'association *(*`perimetre, Float-->Float, ??`*)*

Remarque La constante `PI`, ayant été déclarée après la fonction `perimetre`, figure dans le contexte de la déclaration de son corps et donc dans sa fermeture.

5.4.4 Exemple

Le programme calculant le produit de matrice du chapitre précédent se traduit naturellement par une fonction.

On considère que le type `Matrice` appartient à l'environnement de la fonction `produit_matrice`.

```
type Matrice is
    array(Positive range <>,Positive range <>) of Float;
function produit_matrice(A,B: Matrice) return Matrice is
  C:Matrice(1..A'last(1),1..B'last(2));
begin
  for i in 1..m loop
    for j in 1..n loop
      declare
        s:Float:=0.0;
      begin
        for k in 1..p loop
          s:=s+A(i,k)*B(k,j);
        end loop;
        C(i,j):=s;
      end;
    end loop;
  end loop;
  return C;
end produit_matrice;
```

On peut voir que les paramètres formels `A` et `B` sont de type tableau multidimensionnel non contraint et que, malgré tout, leur bornes sont accessibles à l'intérieur de la fonction via les attributs (ici l'attribut `last`). On constate d'autre part que la valeur retournée est elle-même de type tableau non contraint. La contrainte sera fixée au cours de l'exécution.

5.5 Application d'une fonction

5.5.1 Syntaxe

Soit une fonction dont la déclaration est schématisée ainsi :

```
function g(P1:T1;P2:T2;...;Pn:Tn) return T;
```

Le type de la fonction `g` est : `T1*T2*...*Tn --> T`

Association entre arguments et paramètres formels Lorsque l'on applique une fonction à des arguments, l'association peut être spécifiée de différentes façons :

- d'après l'ordre d'écriture

  ```
  g(A1,A2, ...,An)
  ```

- en nommant le paramètre formel

  ```
  g(Pk=>Ak,...,Pi=>Ai)
  ```

- en panachant (les paramètres par position apparaissent en premier et dans l'ordre)

  ```
  g(A1,A2,...,Ak-1,Pk=>Ak,...,Pn=>An)
  ```

Les arguments `Ai` doivent être d'un type compatible avec le type du paramètre formel correspondant `Ti`.

Paramètres par défaut Si, lors de la plupart des appels, un paramètre prend la même valeur, alors il est possible de spécifier cette valeur dans la déclaration de la fonction.

Dans ce cas, il n'est pas obligatoire de mentionner le paramètre effectif au moment de l'appel (sauf s'il est suivi d'autres paramètres non définis par défaut).

```
function eliminer(C:Character:=' ';T:String) return String;
-- cette fonction a pour but de retourner le texte  T passé en
-- paramètre après suppression de tous les caractères C du texte.
```

Si l'on veut supprimer le caractère espace d'un texte donné

```
with Ada.Text_io;use Ada.Text_io;
procedure eliminer_car is
  function eliminer(C:Character:=' ';T:String)
                                      return String is
     function decaler(T:String;inf,sup:Positive)
                                        return String is
       X:String:=T;
     begin
       for i in inf..sup-1 loop
         X(i):=X(i+1);
       end loop;
```

```
      return X(inf..sup-1);
    end decaler;
    X:String:=T;
    deb:Positive:=T'first;
    fin:Positive:=T'last;
  begin
    -- on suppose que T n'est pas vide
    loop
    exit when deb>fin;
      if X(deb)=C
      then
         X(X'first..fin-1):=
                   X(X'first..deb-1)&decaler(X,deb,fin);
         fin:=fin-1;
      end if;
      deb:=deb+1;
     end loop;
     return X(X'first..fin);
   end eliminer;

   texte:String:="hier c'est le passé;demain c'est le futur;aujourd'hui c'est un cadeau.C'est pour ça
begin
  put_line(eliminer(T=>texte));
  -- Si un paramètre est omis (cas du paramètre par défaut,
  -- on doit nommer les autres paramètres
  put_line(eliminer('e',texte));
end eliminer_car;
```

5.5.2 Sémantique

Soit la fonction,

```
function h(P:T) return T';
function h(P:T) return T' is
begin
-- corps de h
end h;
```

Evaluons `h(exp)`,

1. L'évaluation de `h` permet de récupérer le code binaire qui lui est associé ainsi que l'environnement dans lequel son corps avait été déclaré.
2. L'évaluation de `exp` permet d'attribuer une valeur à l'argument.

A partir de ces informations, le contexte d'exécution de l'appel est construit. La valeur de `exp` est associée à l'identificateur du paramètre formel et le code binaire exécuté.

A la fin de l'exécution, ce contexte est détruit et remplacé par le contexte d'avant l'appel.

5.5.3 Exemple

L'application de la fonction `produit_matrice` précédente pourrait se faire dans un bloc

```
declare
X:Matrice(1..5,1..3):= -- agrégat d'initialisation
Y:Matrice(1..3,1..8):= -- agrégat d'initialisation
C:Matrice(1..5,1..8);
begin
C:=produit_matrice(X,Y);
for i in 1..m loop
  for j in 1..n loop
    put(C(i,j));
-- on suppose qu'Ada.Float_Text_io appartient à l'environnement
  end loop;
  new_line;
end loop;
end;
```

Le paramètre formel `A` (resp `B`) (de type `Matrice` non contraint) prendra la valeur du paramètre effectif `X` (resp `Y`). Le code binaire de la fonction sera exécuté pour les valeurs `X` et `Y` des paramètres formels `A` et `B`.

5.5.4 Exemple

Application de la fonction `double`,

```
function double(X:Integer) return Integer;
function double(X:Integer) return Integer is
begin
   return 2*X;
end double;
```

Il s'agit d'évaluer `double(18);`

1. L'évaluation de `double` ramène le code binaire de la fonction
2. Le paramètre formel `X` prend la valeur `18`. Cette association étend l'environnement de la déclaration du corps de la fonction.
3. Le code s'exécute alors, la valeur retournée est `36` et le contexte d'exécution détruit.
4. On revient donc au contexte d'exécution initial.

5.5.5 Exemple

```
TVA : constant Float:=19.6;
function PTTC(X:Float) return Float;
function PTTC(X:Float) return Float is
begin
   return X+0.01*TVA*X;
end PTTC;
```

1. L'évaluation de la déclaration de la constante `TVA` associe l'identificteur `TVA` au type `Float` et à la valeur `19.6`
2. L'évaluation de la déclaration de la fonction `PTTC` associe l'identificateur `PTTC` au type `Float->Float` et à une valeur indéfinie.
3. L'évaluation du corps de la fonction crée la valeur de cette fonction. Cette valeur est composée du code exécutable de la fonction associé au contexte de sa déclaration.

Evaluation de

```
put(PTTC(10.0));
```

- Evaluation de `PTTC(10.0)`
  - Evaluation de `PTTC` (le code binaire est mis à disposition ainsi que son contexte)
  - Evaluation de `10.0`
  - Formation du contexte d'exécution (on remarquera que `TVA` appartient à ce contexte), le paramètre formel `X` prenant la valeur `10.0`.
- Exécution du code binaire dans le contexte mis en place

```
X+0.01*TVA*X ----> 11.96
```

- Destruction du contexte d'exécution de `PTTC`
- Evaluation de l'objet `put(11.96)`
  - Le code binaire de `put` est mis à disposition ainsi que son contexte
  - Formation du contexte d'exécution (avec association du paramètre formel de `put` et de la valeur de son argument soit `11.96`)
  - Exécution (affichage de la valeur `11.96`)
  - Destruction du contexte d'exécution

Remarque L'ordre d'évaluation de `put` et `PTTC(10.0)` est indéterminé.

5.5.6 Exemple

Evaluation de la séquence d'instructions :

```
TVA:=28.0;
put(PTTC(10.0));
```

En imaginant que `TVA` puisse prendre une autre valeur (c'est en fait impossible car `TVA` a été déclarée constante)

- Evaluation de `TVA:=28.0;`
  - `TVA` est associé à la valeur `28.0`
- Evaluation de `put(PTTC(10.0));`
  - Evaluation de `PTTC(10.0)`
    - Evaluation de `PTTC` (le code binaire est mis à disposition ainsi que son contexte)
    - Evaluation de `10.0`
    - Formation du contexte d'exécution (on remarquera que `TVA` appartient à ce contexte et vaut maintenant `28.0`), le paramètre formel `X` prenant la valeur `10.0`.
  - Exécution du code binaire dans le contexte mis en place `X+0.01*TVA*X ----> 12.8`
  - Destruction du contexte d'exécution de `PTTC`
  - Evaluation de l'objet dénoté par l'identificateur `put`
    - Le code binaire est mis à disposition ainsi que son contexte. Le contexte d'exécution est formé (avec association du paramètre formel de `put` et de la valeur de son argument `12.8`).
    - Exécution (affichage de la valeur `12.8`)
    - Destruction du contexte d'exécution de `put`.

Conclusion L'expression `PTTC` évaluée dans deux états différents donne des valeurs différentes.

La leçon a tirer de cet exemple est le danger d'utiliser des variables externes (globales) dans le corps d'une fonction. En effet, la notion même de fonction n'est plus respectée puisque, pour deux données identiques, la fonction peut produire (selon la valeur de la variable globale) deux résultats différents. Toutes les valeurs nécessaires au calcul de la fonction doivent être contenues (encapsulées) dans l'environnement de déclaration de la fonction. En effet, cet environnement constitue la base immuable pour tout calcul (tout appel de la fonction).

5.6 Résumé

- La déclaration d'une fonction Ada se fait en deux parties séparées :
  - Déclaration de la fonction (introduction de l'identificateur de la fonction ainsi que du nom et du type de ses paramètres).
  - Déclaration du corps de la fonction (introduction de la définition du corps de la fonction).
- Une fonction peut être déclarée localement à une autre fonction.
- Toute fonction Ada est naturellement récursive.
- Une fonction peut redéfinir un opérateur du langage (à condition de le mettre entre guillemets).
- D'une manière générale, toute fonction peut être surchargée. Cela revient à lier plusieurs valeurs à un même identificateur. Dans ce cas, il appartient au compilateur de déterminer le type et la valeur qui convient selon le contexte d'utilisation de l'identificateur.
- L'évaluation d'une application est réalisée selon le même schéma :
  - Evaluation de l'identificateur de la fonction dans l'état courant
  - Evaluation des paramêtres effectifs dans l'état courant
  - Construction de l'environnement d'exécution du corps
  - Exécution du corps
  - Restitution de l'environnement initial

5.7 Fonctions récursives

La fermeture d'une fonction contient une liaison pour cette fonction donc toute fonction est naturellement récursive. Autrement dit, toute fonction Ada se connait elle-même.

5.7.1 Exécution d'une fonction récursive

Soit la fonction `SommeCarres(n)` qui calcule la somme des carrés des `n` premiers entiers. Elle est définie récursivement par :

```
SommeCarres(n)= si n=0 alors 0
                sinon SommeCarres(n-1)+n*n
```

étapes du calcul	environnement d'exécution	pile d'exécution
SommeCarres(3)	n=3	[]
SommeCarres(n-1)+n*n	n=3	[]
SommeCarres(2)	n=2	[n=3]
SommeCarres(n-1)+n*n	n=2	[n=3]
SommeCarres(1)	n=1	[n=2,n=3]
SommeCarres(n-1)+n*n	n=1	[n=2,n=3]
SommeCarres(0)	n=0	[n=1,n=2,n=3]
0	n=0	[n=1,n=2,n=3]
0+n*n-->1	n=1	[n=2,n=3]
1+n*n-->5	n=2	[n=3]
5+n*n-->14	n=3	[]

A chaque appel récursif correspond un empilement de l'environnement d'exécution courant.

A chaque retour d'appel récursif correspond un dépilement et donc la restitution de l'environnement d'exécution approprié au calcul courant.

5.7.2 Conception d'une fonction récursive

Reprenons l'exemple précédent où il s'agit d'éliminer toutes les occurrences d'un caractère donné dans un texte. Nous en avons donné une version itérative, voyons maintenant comment concevoir son codage itératif.

Nous nous appuyons sur le raisonnement par récurrence.

Soit `T` le texte et `C` le caractère à éliminer. Considérons l'ensemble des textes `T` possibles. Leurs longueurs varient entre `0` et `n`.

Pour T0 (un texte de longueur 0), on connait immédiatement le résultat de l'élimination des occurrences d'un caractère quelconque :

```
si longueur(T)=0 alors le résultat est T
```

Supposons que l'on est capable d'éliminer tous les caractères `C` d'un texte `T` de longueur `i`.

Alors, il reste à montrer que l'on est capable d'éliminer les caractères `C` d'un texte de longueur `i+i`.

```
si le premier caractère=C
alors
  le résultat provient de l'elimination de toutes les
  occurrences de C du texte T(i+1..n) (de longueur i)
sinon
  le résultat provient de la concaténation du 1er caractère
  du texte (T(i)) au texte T(i+1..n) (de longueur i) dans
  laquelle toute occurrence de C a été supprimée.
```

Ce qui se traduit par le code Ada :

```
with Ada.Text_io;use Ada.Text_io;
procedure eliminer_rec is
   function eliminer(C:Character:=' ';T:String)
                                      return String is
```

```
      X:String:=T;
      i:Positive:=T'first;
      n:Positive:=T'last;
   begin
      if n<i
      then
        return "";
      else
        if X(i)=C
        then
          return eliminer(C,X(i+1..n));
        else
          return X(i)&eliminer(C,X(i+1..n));
        end if;
      end if;
   end eliminer;

   texte:String:="hier c'est le passé;demain c'est le futur;aujourd'hui c'est un cadeau.C'est pour ça
begin
  put_line(eliminer(T=>texte));
  put_line(eliminer('e',texte));
  -- Si un paramètre est omis (cas du paramètre par défaut,
  -- on doit nommer les autres paramètres
end eliminer_rec;
```

5.7.3 Fonctions mutuellement récursives

La séparation entre la déclaration d'une fonction et la déclaration de son corps, permet les fonctions mutuellement récursives.

Exemple Soient deux fonctions définies par leurs suites récurrentes,

$\begin{cases} U_0=1 \\ U_n=2*V_{n-1}+U_{n-1} \end{cases}$

$\begin{cases} V_0=-1 \\ V_n=2*U_{n-1}+V_{n-1} \end{cases}$

```
function V(N:Integer) return Integer;
function U(N:Integer) return Integer;
function U(N:Integer) return Integer is
begin
   if (N=0) then
      return 1;
   else
      return 2*V(N-1)+U(N-1);
   end if;
end U;
function V(N:Integer) return Integer is
begin
   if (N=0) then
       return (-1);
    else
       return 2*U(N-1)+V(N-1);
    end if;
end V;
```

Chapter 6

Procédures

6.1 Intérêt

- Créer une instruction nouvelle qui deviendra une primitive pour le programmeur
- Structurer le texte source du programme et améliorer sa lisibilité
- Factoriser l'écriture lorsque la suite d'actions nommée par la procédure intervient plusieurs fois

6.2 Différence entre fonction et procédure

6.2.1 Fonction

Elle peut être vue comme un opérateur produisant une valeur.

Une fonction n'a pas pour rôle de modifier l'état courant du programme en exécution.

Une bonne programmation interdit aux fonctions de réaliser des *effets de bords.*

On appelle *effet de bord* toute modification de la mémoire (affectation d'une variable, opération de lecture en mémoire) ou toute modification d'un support externe (disque, écran, disquette, etc).

Une fonction n'est pas une instruction, elle n'est donc pas en mesure de modifier l'état du programme. Une fonction réalise une simple opération dont le résultat peut être, par la suite, utilisé par une instruction.

6.2.2 Procédure

Une procédure est une instruction composée qui peut prendre des paramètres et dont le rôle est de modifier l'état courant. Les procédures ne retournent pas de résultat.

Les 3 aspects d'une procédure :

1. Déclaration
2. Déclaration du corps
3. Appel

6.3 Déclaration de procédure

6.3.1 But

Introduction dans l'environnement courant :

- de l'identificateur
- du type
- du mode de communication

6.3.2 Syntaxe

<déclaration_proc>::=<en-tete_proc>;
<en-tete_proc>::=procedure <ident_proc> <liste_paramètres>
<liste_paramètres>::=vide|(<déclar_paramètres>)
<déclar_paramètres>::=
<ident_param>:<mode> <type_param>{;<déclar_paramètres>}

où

<ident_param> ::= *identificateurs des paramètres formels*

<mode> ::=*mode de passage des paramètres*

<type> ::= *type des paramètres formels*

Exemple

```
procedure permuter(a:in out Character;b:in out Integer);
```

Remarques

- La déclaration d'une procédure n'est pas obligatoire, la déclaration de son corps peut en tenir lieu.
- Une valeur par défaut peut être associée à chaque paramètre formel

Exemple

```
procedure accumuler(accu:in out Integer:=0;
                    x:in Tableau;
                    n:in Positive);
```

6.3.3 Type d'une procédure

Une procédure ne renvoie pas de valeur.

Son type est de la forme : `t1*t2*....*tn->vide`

Le type de la procédure `permuter` est :

```
Character*Integer->vide
```

6.3.4 Sémantique

- Soit la **déclaration** de la procédure :

```
procedure permuter(a:in out Character;b:in out Integer);
```

L'objet de nom `permuter` est maintenant connu et utilisable dans le programme.

A cet objet est associé le type : `Character*Integer->vide`

Aucune valeur ne lui est encore liée.

- Et la **déclaration du corps** de cette procédure :

```
procedure permuter(a:in out Character;b:in out Integer) is
  z:Integer;
begin
  z:=Character'pos(a);
  a:=Character'val(b);
  b:=z;
end permuter;
```

Le code exécutable de cette procédure associé au contexte de sa déclaration tient lieu de valeur liée à l'identificateur `permuter`.

6.4 Appel de procédure

L'application d'une procédure déclarée et définie à des paramètres effectifs constitue l'appel de cette procédure.

6.4.1 Syntaxe

<appel_procédure>::=<id_proc>(<liste_params_effectifs>)|<id_proc>;
<liste_params_effectifs>::=<param>{,<param>}
<param>::=<valeur>|<id_param>

Le type des paramètres effectifs doit être le même que celui du paramètre formel correspondant.

Si le paramètre est déclaré en mode `in`, toute expression peut être utilisée pour le décrire.

Si le paramètre est déclaré en mode `out` ou `in out`, le paramètre effectif doit être une variable.

6.4.2 Association paramètre effectif-paramètre formel

- Liaison d'après l'ordre d'écriture

```
ident_proc(a1,a2,...,an);
```

- Liaison en nommant le paramètre formel

```
ident_proc(pi=>ai,pj=>aj,...,p2=>a2);
```

- Panachage

```
ident_proc(a1,a2,...,pj=>aj,...,pi=>ai);
```

6.4.3 Exemple

```
begin
  declare
     m:Character:='*';
     n:Integer:=65;
  begin
     put(" m=");put(m);
     put("n=");
     put(n,width=>2);new_line;
     permuter(m,n);
     -- ou permuter(b=>n,a=>m);
     put("permuter(m,n);");new_line;
     put(" m=");put(m);
     put("n=");
     put(n,width=>2);new_line;
  end;
end;
```

```
m=* n=65
permuter(m,n);
m=A n=42
```

6.4.4 Sémantique de l'appel

```
permuter(b=>n,a=>m);
```

`b` prend la valeur `'*'`

`a` prend la valeur `65`

Le contexte d'exécution de la procédure est formé, la procédure est exécutée.

A la fin de l'exécution :

```
m vaut 'A'
n vaut 42
```

Retour vers l'appelant après exécution de la procédure :

```
b vaut 42
a vaut 'A'
```

6.5 Passage de paramètres : mode `in`

Soit l'unité,

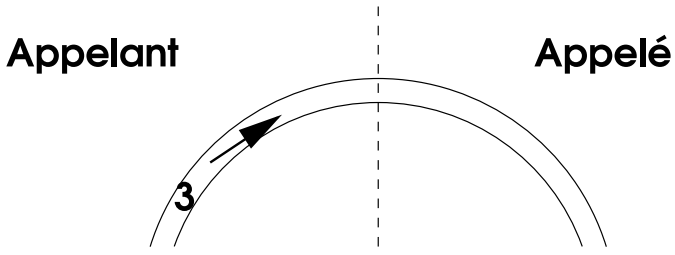
```
declare
 y,z:Integer;
 procedure doubler(x:in Integer;t:out Integer) is
 begin
```

```
   t:=2*x;
 end doubler;
begin
 y:=3;
 doubler(y,z);
 put("z=");put(z,width=>3);
end;
```

6.5.1 Appel de la procédure

Le contrôle de type a été effectué. Les paramètres formels et effectifs correspondant ont donc le même type.

- Transmission du paramètre passé en mode `in`
  - Etablissement d'un canal de communication
  - Et d'un sens de la communication
  - Copie de la valeur du paramètre effectif `y` et affectation de cette valeur au paramètre formel correspondant `x`
  - Le paramètre formel `x` est une constante pendant toute l'exécution de la procédure.




```
doubler(y,z);   procedure doubler(x:in Integer;t:out Integer) is
                   ...
                begin
                   ...
                end doubler;
```

- Exécution de l'appelé
- Retour vers l'appelant
  - Le contrôle est passé à l'appelant derrière l'appel.
  - **Aucune valeur n'est transmise**

6.6 Passage de paramètres : mode `out`

Soit le bloc,

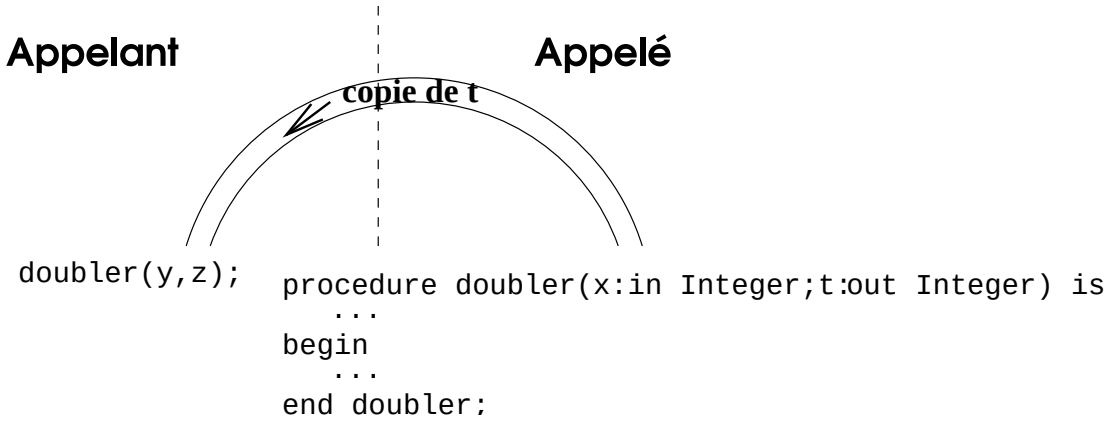
```
declare
 y,z:Integer;
 procedure doubler(x:in Integer;t:out Integer) is
 begin
```

```
    t:=2*x;
  end doubler;
begin
  y:=3;
  doubler(y,z);
  put("z=");put(z,width=>3);
end;
```

Le contrôle de types a été effectué. Les paramètres formels et effectifs correspondant ont donc le même type.

- Transmission du paramètre passé en mode `out`
  - Etablissement d'un canal de communication
  - Et d'un sens de communication

Le paramètre formel `t` est une *variable* dont la valeur initiale est *indéfinie.*



- Exécution de l'appelé
- Retour vers l'appelant

Copie de la valeur du paramètre formel `t` et affectation de cette valeur au paramètre effectif `z` (qui doit être une variable).

Le contrôle est passé à l'appelant derrière l'appel.

6.7 Passage de paramètres : mode `in out`

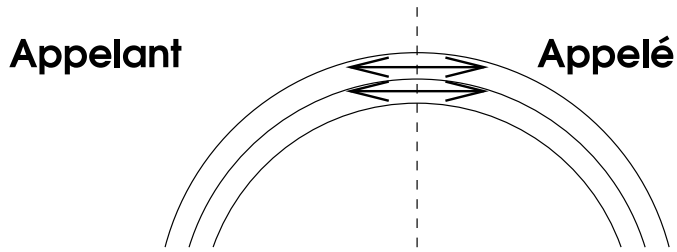
Soit le bloc,

```
declare
  x:Integer:=56; y:Integer:=57;
  procedure echange(u,v:in out Integer) is
    z:Integer;
  begin
    z:=u;u:=v;v:=z;
  end echange;
begin
  echange(x,y);
end;
```

Le contrôle de types a été effectué. les paramètres formels et effectifs correspondant ont donc le même type.

- Transmission du paramètre passé en mode `in out`
  - Etablissement d'un canal de communication
  - Et d'un sens de communication

Les paramètres formels `u` et `v` sont des variables dont les valeurs initiales sont les copies des valeurs des paramètres effectifs `x` et `y`.




```
doubler(x,y);    procedure doubler(u,v: in out Integer)is
                  ...
                 begin
                  ...
                 end doubler;
```

- Exécution de l'appelé
- Retour vers l'appelant

Copie des valeurs des paramètres formels `u` et `v` et affectation de ces valeurs aux paramètres effectifs `x` et `y` (qui doivent être des variables)

Le contrôle est passé à l'appelant derrière l'appel.

6.8 Passage de paramètres : par référence

- Les passages en mode `in, out` et `in out` imposent une recopie de valeur.
- Cette recopie peut être pénalisante lorsque les valeurs sont de taille importante (grande matrice).
- Les compilateurs Ada ont la possibilité d'utiliser un autre mode de transmission de paramètres : le *passage par référence* pour pallier cet inconvénient.
- Ce choix est transparent à l'utilisateur.

6.8.1 Exemple

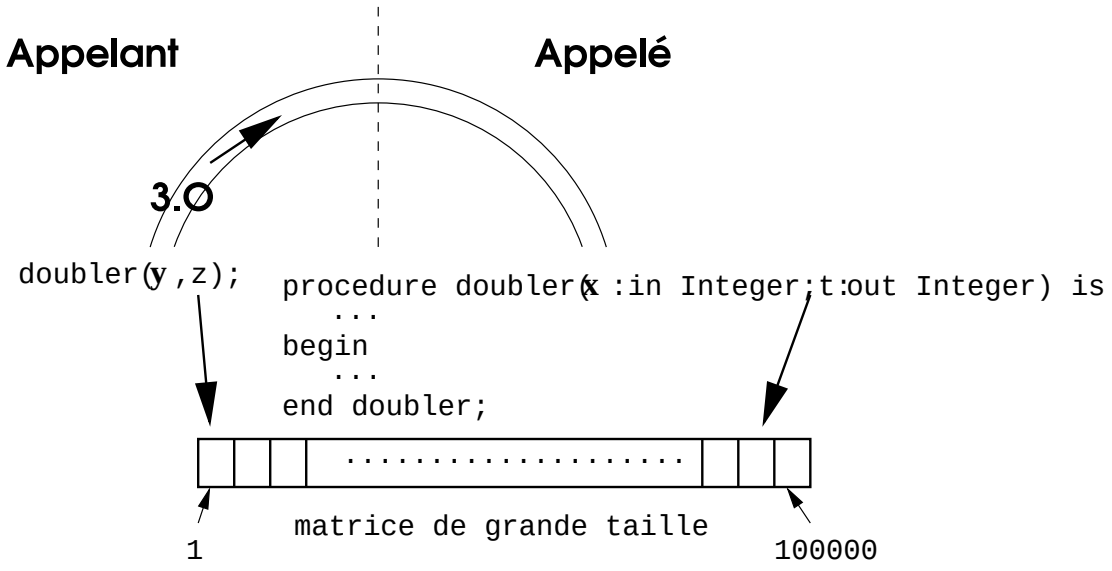
Soit le bloc,

```
declare
 type Vecteur is array(Positive) of Float;
 y:Float;z:Vecteur(1..100000):=(1.12,3.21,6.71,....,2.45);
 procedure doubler(x:in Integer;t:out Vecteur) is
 begin
   for i in Vecteur'range loop
     t(i):=2.0*x;
   end loop;
 end doubler;
begin
 y:=3.0;
 doubler(y,z);
 put("z=");put(z);
end;
```

Après transmission des paramètres, le vecteur de dimension `1000` possède 2 synonymes :

1. `z` dans l'environnement de l'appelant
2. `t` dans l'environnement de l'appelé qui lui est lié

Conséquence, toute modification apportée à `t` dans l'appelé modifie aussi `z`.



6.9 Passage de paramètres : par valeur

- Ce mode de passage n'existe pas en Ada.
- On le trouve dans des langages impératifs comme C, C++, Pascal.
- Comme dans le mode `in`, une valeur est transmise de l'appelant vers l'appelé via un paramètre.
- La différence est que, dans l'appelé, ce paramètre n'est pas considéré comme une constante mais comme une variable.
- De même que pour le mode `in`, après retour vers l'appelant, ce type de passage garantit que le paramètre effectif conserve sa valeur d'appel.

Exemple

```
declare
  type Tableau is array(Positive) of Integer;
  procedure init(x:in Positive;T:out Tableau) is
  begin
    for i in T'range loop
      T(i):=x;
      x:=x+1;
-- affectation illicite en Ada car x est une constante
-- mais affecation licite en Pascal ou C
    end loop;
  end init;
  Tab:Tableau(1..20);
begin
  init(1,Tab);
end;
```

6.10 Surcharge des procédures

6.10.1 Profil des paramètres

Deux procédures ont le même *profil* de paramètres si :

1. elles ont le même nombre de paramètres
2. à chaque position, les paramètres ont le même type

```
procedure P(a:in Character;b:out Integer);

procedure Q(x:in Character;y:out Integer);
```

Les procédures `P` et `Q` ont le même profil de paramètres

6.10.2 Masquage des procédures

Si deux pocédures ont même *profil* et même identificateur, la dernière déclarée *masque* la précédente.

Une déclaration de procédure ne peut pas *masquer* une déclaration de fonction et vice versa.

6.10.3 Surcharge des procédures

Un identificateur de procédure est *surchargé* si :

1. il identifie plusieurs procédures
2. si ces procédures n'ont pas le même profil de paramètres

Un tel identificateur possède plusieurs sémantiques

```
procedure permuter(a:in out Character; b:in out Character);

procedure permuter(x:in out Integer; y:in out Float);
```

Le code de ces deux procédures sera différent puisque leur type est différent.

Ces deux déclarations sont dites *surchargées.*

Remarque Nous avons déjà remarqué l'intérêt de ce concept de surcharge. En effet, dans les bibliothèques Ada, la procédure `put` est de nombreuses fois surchargée. Elle est aussi bien utilisée pour afficher des caractères, des entiers, des flottants, des valeurs énumérées. Cela évite de multiplier les identificateurs. En fait ce concept permet de capturer une fonctionnalité commune à plusieurs procédures, ici il s'agit de la fonctionnalité d'affichage.

6.11 Procédures récursives

Comme les fonctions, les procédures Ada sont naturellement récursives.

Exemple Les tours de hanoi

3 pieux `A,B,C` sont plantés en terre. Initialement, sur le pieu `A` sont empilés des disques de taille décroissante. Les pieux `B` et `C` sont alors vides.

A l'état final tous les disques sont empilés sur le pieu `C`.

Les règles sont les suivantes :

- on ne peut déplacer qu'un disque à la fois et bien sûr, il ne peut être placé qu'en haut d'une pile (au sommet d'un pieu).
- il est possible d'utiliser le pieu `B`
- on peut toujours empiler un disque sur un autre à la condition que sa taille soit inférieure

Le programme qui suit construit et affiche la suite des déplacements de disques à l'aide d'une procédure récursive. Les 3 pieux sont représentés par les caractères `'A, 'B', 'C'`. Les disques sont représentés par un entier qui simule leur taille.

```
with Ada.Text_io;use Ada.Text_io;
with Ada.Integer_Text_io;use  Ada.Integer_Text_io;

procedure tours_hanoi is
  procedure hanoi(n:in Natural;X,Y,Z:in Character) is
  begin
    if n/=0
    then
      hanoi(n-1,X,Z,Y);
      put("disque : ");put(n,WIDTH=>2);put(" : ");
      put(X);put(" --> ");put(Y);new_line;
      hanoi(n-1,Z,Y,X);
    end if;
  end hanoi;
  A:Character:='A';
  B:Character:='B';
  C:Character:='C';
begin
  hanoi(3,A,C,B);
end tours_hanoi;
```

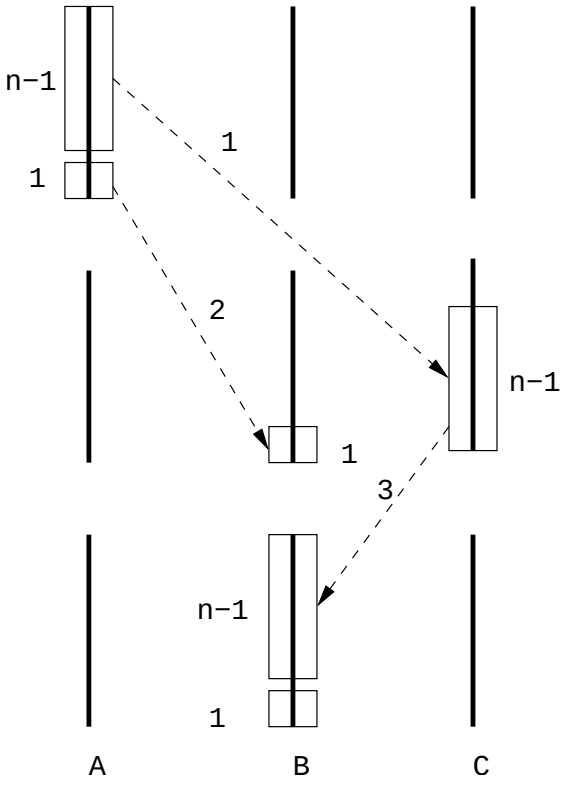


Figure 6.1: *Tours de hanoi*

Le résultat pour un pieu `A` où sont empilés 3 disques est :

```
disque :  1 : A --> C
disque :  2 : A --> B
disque :  1 : C --> B
disque :  3 : A --> C
disque :  1 : B --> A
disque :  2 : B --> C
disque :  1 : A --> C
```

L'idée est, ici de dire :

- si aucun disque n'est empilé sur A, alors il n'y a rien à faire
- supposons que je sache déplacer les n-1 disques les plus petits de A vers C en utilisant B
- alors il suffit de placer le plus gros disque de A sur le pieu libre B et de recommencer ce que j'ai supposé savoir faire, c'est à dire déplacer les n-1 disques (cette fois de C vers B en utilisant A).

Chapter 7

Modèle sémantique

7.1 Calcul, programme, exécution

7.1.1 Calcul

Un calcul exprime une solution à un problème. Il est formé d'expressions construites à l'aide de valeurs (arithmétiques ou autres) et d'opérateurs.

7.1.2 Valeurs

Tout programme manipule, lit, écrit, crée des valeurs.

Les valeurs élémentaires sont :

- les caractères
- les entiers
- les réels
- les booléens
- les chaines de caractères

Pour les distinguer, une représentation Ada leur est associée.

Exemple

```
3       3.      '3'     "3"
TRUE    "TRUE"
```

7.1.3 Représentation Ada des valeurs

- Un entier est représenté classiquement : `3` `56` `987`
- Un réel est représenté avec la notation pointée : `3.1415`

- Un caractère est représenté entre quotes pour le distinguer de tout autre identificateur : `'A'  '7'  '$'`
- Une chaine de caractères est représentée entre guillemets pour la distinguer de tout autre identificateur ou mot clé : `"Hello World "`
- Un booléen est représenté par les symboles : `True, False`

En dehors de leur signification particulière, elles se distinguent par la manière dont elles sont représentées en machine.

7.1.4 Expressions

Les *valeurs* constituent des ensembles.

L'environnement initial (qu'il est inutile d'importer explicitement) comporte toutes les opérations sur les éléments de ces ensembles de base.

Une *expression* est formée à partir des *valeurs* et *opérateurs* associées à cet ensemble.

Une *expression* crée ainsi une nouvelle valeur.

Exemple

```
(15+30)/2
"AU"&"SECOURS"
True and False
```

7.1.5 Programme

Un programme met en oeuvre un calcul sur un exécutant informatique.

Il manipule des objets informatiques représentant les différentes valeurs (autres que celles qui sont exprimées littéralement).

Ces objets peuvent être de nature différente :

- variable informatique
- constante
- fonction
- procédure
- type
- exception
- paquetage
- ...

7.1.6 Objets informatiques et identificateurs

Outre une valeur, les objets informatiques possèdent un type. En Ada, ils possèdent **un type et un seul.**

Les objets Ada, pour être manipulés, doivent posséder un nom.

On parle d'identificateur pour signifier qu'un nom est construit en respectant une grammaire .

7.1.7 Exécution et contrôle

Un programme contient l'expression d'un calcul et les éléments qui permettent son exécution, les mécanismes de contrôle (boucles, séquence, conditionnelle, exceptions, appels de fonctions et de procédures)

Les mécanismes de contrôle permettent l'ordonnancement des instructions d'un programme.

Pour ordonnancer les instructions, il faut définir comment on continue un calcul..

On répond ainsi à la question : quelle instruction exécuter ensuite ?

7.1.8 Exécutant

Il est défini par :

- un modèle d'exécution implanté (mécanisme de gestion de la continuation)
- un mécanisme d'exécution des instructions

7.1.9 Formalisation d'une exécution

C'est une fonction de transformation :

```
P*D->D
```

Toute donnée est exécutable (évaluable)

Un programme est une donnée particulière qui nécéssite d'autres données pour s'exécuter.

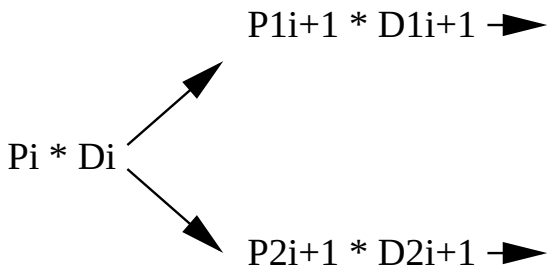
7.2 Notion d'état

7.2.1 Décomposition d'une exécution

- En exécution séquentielle (ordre total)

```
P1*D1-> P2*D2->...->Pn*Dn->...
```

- En exécution parallèle (ordre partiel restitué par la synchronisation des processus)



7.2.2 Etat d'un programme ([?]T.hardin&V.Donzeau-Gouge)

L'état d'un programme correspond à la description d'une étape de son exécution. On parle d'état courant pour spécifier un état à un instant donné de l'exécution.

L'état d'un programme est constitué des objets informatiques manipulés par le programme avec leur valeur à une étape donnée de l'exécution.

Plus formellement, l'état d'un programme est représenté par le couple formé de l'*environnement* (Env) et de la *mémoire* (Mem).

7.2.3 Environnement

L'environnement rassemble l'ensemble des identificateurs d'objets, utilisables au cours d'une exécution, associés à leur type et leur valeur

Identificateur	**Valeur**
variable	adresse
constante	valeur
exception	valeur d'exception
fonction/procédure	fermeture
paquetage	environnement
type	valeur de type

C'est une liste de triplets (identificateur, type, valeur)

7.2.4 Mémoire

De manière simplifiée, la *mémoire* contient des valeurs référencées par des adresses. La *mémoire* est vue comme une liste de couples *(adresse, valeur referencée)*.

Tout identificateur appartenant à l'environnement référence une valeur de la mémoire. Toute valeur de la mémoire non associée à un identificateur devient inaccessible.

Un programme *ramasse-miettes* ("garbage collector") récupère les zones mémoire allouées à des objets devenus inaccessibles.

7.3 Déclaration

7.3.1 Généralités sur les déclarations

Une déclaration d'objet permet :

- d'introduire son identificateur dans l'environnement.
- de préciser les conditions de son utilisation (son type)
- de définir une valeur initiale (facultativement)

7.3.2 Variables informatiques

Une *variable informatique* est un objet informatique dont la valeur peut changer au cours de l'exécution du programme.

A une adresse donnée correspond un ensemble de valeurs référencées possibles (notion de type) au cours de l'exécution d'un même programme .

Une variable peut être :

- anonyme (création dynamique)
- nommée (liée à un identificateur)

7.4 Déclaration de variable

7.4.1 Déclaration d'une variable : syntaxe

<declaration_variable>::=<identificateur>:<type>:=<expression>;

Toute déclaration est réalisée dans un environnement `Env` donné.

Exemples

```
demain:Jour:=mardi;
numero:Natural:=0;
meteo:Temps:=pluvieux;
X:Integer;
```

7.4.2 Déclaration d'une variable : sémantique

Le mécanisme de déclaration d'une variable se décompose ainsi :

Soit l'état courant

```
Etat0=(Env0, Mem0)
```

- Extension de l'environnement `Env0` avec le couple *:* `(identificateur, ??)`

```
Env1=((identificateur,??),Env0)
Etat1=(Env1,Mem0)
```

- Détermination du type et des contraintes
- Evaluation de l'expression d'initialisation de la variable dans ce nouvel état *(*`Etat1`*)*
- Adjonction du couple *(*`@ident, valeur de l'expression`*)* dans `Mem0` qui devient `Mem1` *(*`@ident` représente la référence associée à l'identificateur de la variable)
- Modification de `Env1`, donc après évaluation de la déclaration,

```
Env1=((identificateur,@ident),Env0)
Etat1=(Env1,Mem1)
```

7.4.3 Déclaration d'une variable : exemple 1

Etant donné l'état :

```
Etat0=(Env0, Mem0)
```

et la déclaration

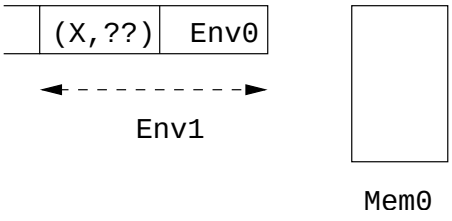
```
X:Natural:=12;
```

- Extension de l'environnement Env avec le couple

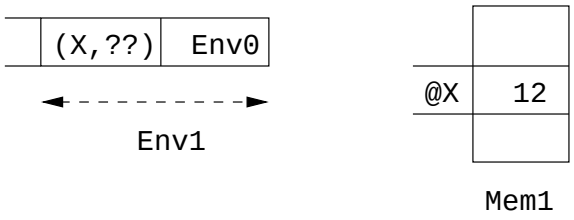
```
Env1=((X,??),Env0)
```

d'où

```
Etat1=(Env1,Mem0)
```

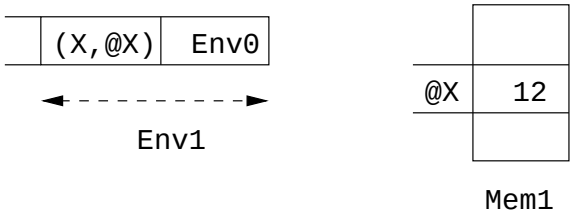


- Détermination du type et des contraintes
- Evaluation de l'expression dans ce nouvel état (Etat1) : l'expression vaut 12
- Adjonction du couple *(@X,12)* dans Mem0 qui devient Mem1



- Modification de Env1

```
Env1=((X,@X),Env0)
```



Après évaluation de la déclaration,

```
Etat3=(Env1,Mem1)
```

7.4.4 Déclaration d'une variable : exemple 2

Etant donné l'état :

```
Etat0=(Env0,Mem0)
```

et la déclaration

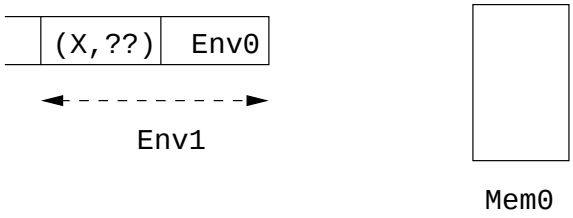
```
X:Natural:=X+1;
```

- Extension de l'environnement `Env` avec le couple

```
Env1=((X, ??),Env0)
```

d'où

```
Etat1=(Env1,Mem0)
```



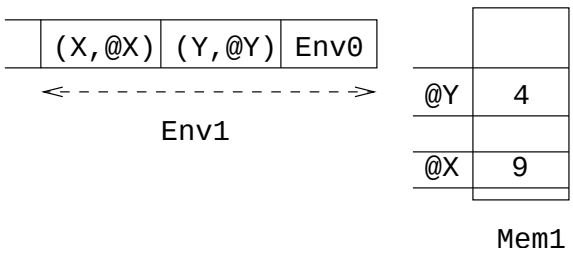
- Détermination du type et des contraintes
- Evaluation de l'expression `X+1` dans `Etat1`

L'identificateur `X` n'étant lié à aucune valeur, l'évaluation est impossible. Une telle **initialisation** est donc **interdite**.

7.4.5 Déclaration d'une variable : exemple 3

Etant donné l'état :

```
Etat1=(Env1, Mem1)
```



et la déclaration

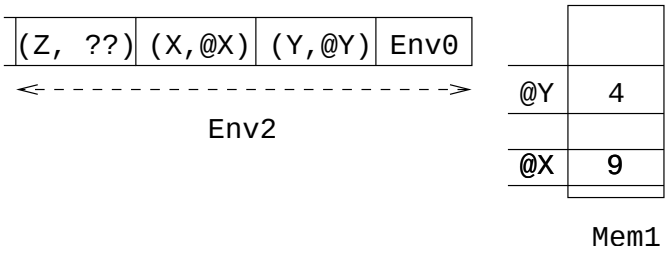
```
Z:Natural:=Y-X;
```

- Extension de l'environnement `Env1` avec le couple

```
Env2=((Z,??),Env1)
```

d'où

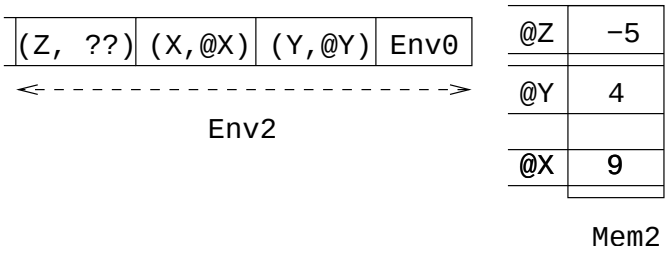
```
Etat2=(Env2,Mem1)
```



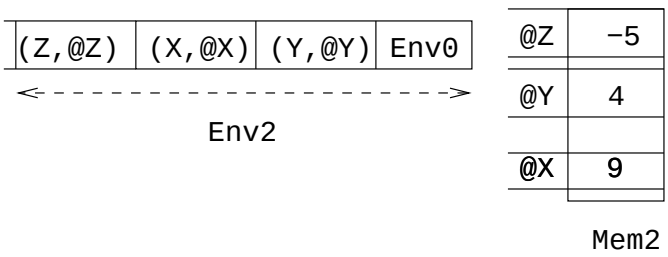
- Détermination du type et des contraintes
- Evaluation de l'expression dans ce nouvel état (`Etat2`)

l'expression *Y-X* vaut *-5*

- Adjonction du couple *(@Z, -5)* dans `Mem1` qui devient `Mem2`. `Etat3` est le nouvel état



- Modification de `Env2`



Après évaluation de la déclaration,

```
EtatFinal=(Env2,Mem2)
```

7.5 Déclaration de constante

Un constante est une valeur qui ne change pas durant l'exécution d'un programme. La déclaration d'une constante permet de nommer un objet et de l'ajouter à l'environnement courant.

7.5.1 Déclaration d'une constante : syntaxe

<déclaration_constante>::=
<ident>:constant<type>:=<expression_constante>;

7.5.2 Exemples

```
PI:constant Float:=3.14159;
CLOVIS:constant Natural:=496;
e:constant Float:=2.712;
DEUX_PI:constant Float:=PI*2.0;
```

7.5.3 Déclaration d'une constante : sémantique

- Ajout de l'identificateur associé à une valeur indéfinie dans l'environnement
- Evaluation de l'expression constante
- Remplacement de la valeur indéfinie par la valeur évaluée dans l'environnement

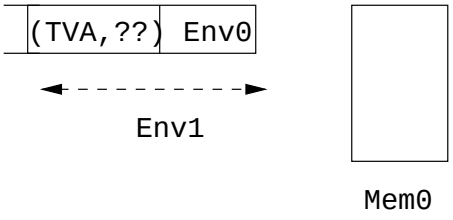
7.5.4 Exemple

Soit, dans l'état `(Env0,Mem0)` la déclarationă:

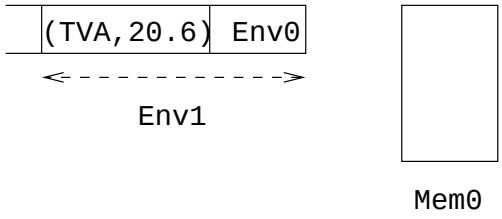
```
TVA:constant Float:=20.6;
```

Elaboration en 3 étapes :

- Ajout de l'identificateur `TVA` associé à une valeur indéfinie dans `Env0`. On obtient le nouvel environnement `Env1`



- Evaluation de l'expression constante —> `20.6`
- Remplacement de la valeur indéfinie (??) par la valeur (évaluée dans `Env1`)

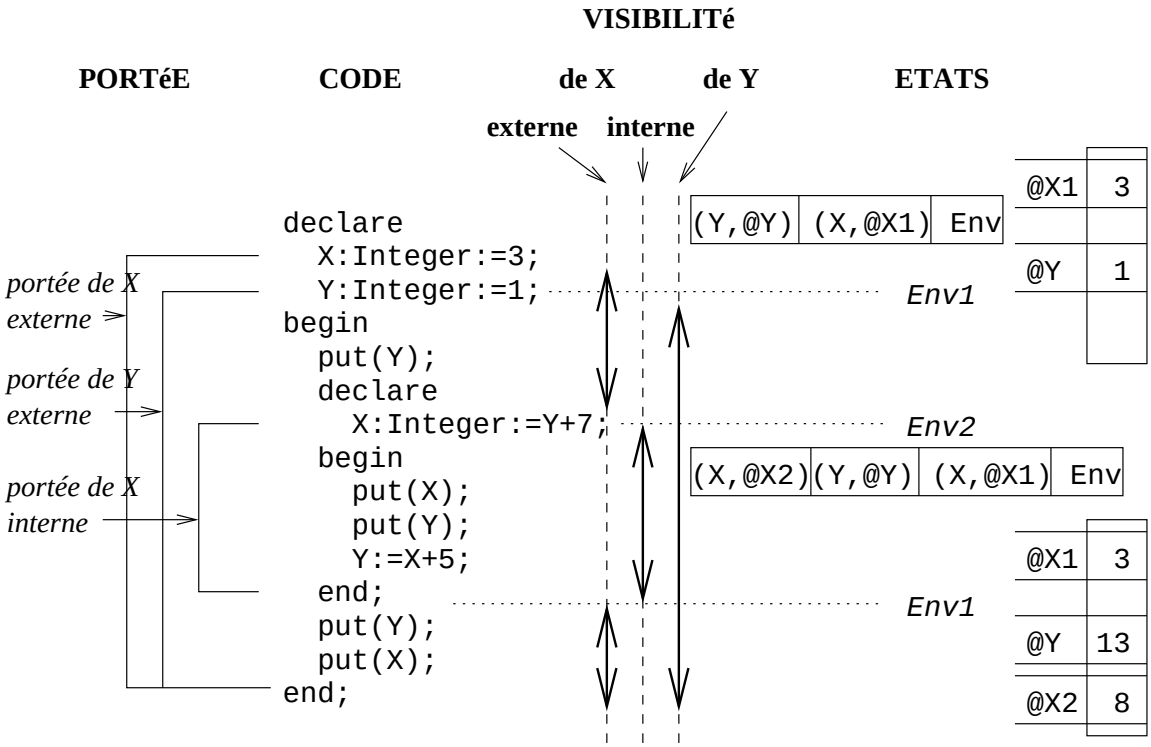


7.6 Portée et visibilité

La *portée* d'un objet informatique est la portion de texte à l'intérieur duquel il a une existence.

Sa *visibilité* ne couvre pas obligatoirement cette même portion. Il se peut qu'un objet de même identificateur le *masque* temporairement.

7.6.1 Exemple 1



Conflit d'identificateurs

Dans cet exemple, le conflit entre les variables de même identificateur `X` est résolu en choisissant celui dont la portée est la plus locale.

En effet, considérant que l'environnement est parcouru de gauche à droite, le seul identificateur `X` visible est celui dont la déclaration est la plus récente; l'autre est masqué (invisible).

Le résultat de l'exécution de ce bloc est :

```
1   8    1    13    3
```

7.6.2 Exemple 2

Dans l'exemple qui suit, nous avons remplacé `7+y` par `7+x`

```
with Ada.Text_io;
use Ada.Text_io;
procedure exemple is
   x:Integer:=3;
   y:Integer:=1;
begin
   put("y=");put(Integer'image(y));
   declare
      x:Integer:=7+x;
   begin
      put("x=");put(Integer'image(x));
      put("y=");put(Integer'image(y));
   end;
   put("x=");put(Integer'image(x));
end exemple;
```

Dans ce cas une erreur est signalée par le compilateur :

`object x cannot be used before end of its declaration`

Pour comprendre cette erreur, il suffit de dérouler le mécanisme de la déclaration d'une variable appliquée à l'instruction :

```
x:Integer:=7+x;
```

Evaluation de cette nouvelle déclaration de variable :

1. ajout de `(x,??)` dans l'environnement courant qui devient le plus récent
2. détermination du type *`(Integer)`*
3. évaluation de l'expression `x+y`. L'évaluation de `x` s'avère impossible puisque la valeur qui lui est associée est indéfinie.

7.6.3 Exemple 3

```
with Ada.Text_io;
use Ada.Text_io;
procedure masquage is
  a,b:Character;
  procedure permuter(x,y:in out Character) is
     z:Character;
  begin
     z:=x;x:=y;y:=z;
  end permuter;
  function permuter(x,y:in Character) return Character is
  begin
     return x;
  end permuter;
begin
  put("a=");get(a);
  put("b=");get(b);
  permuter(a,b);
  put("a=");put(permuter(a,b));
end masquage;
```

On voit ici que les fonction et procédure `permuter` (l'identificateur de la fonction est mal choisi pour illustrer l'exemple), bien que de même *portée* ne se masque pas. On peut observer que le profil des paramètres est semblable mais rappelons qu'une fonction ne peut pas *masquer* une procédure et vice versa.

7.7 Instructions

7.7.1 Affectation

Syntaxe

<affectation>::= <expression_gauche>:=<expression droite>;

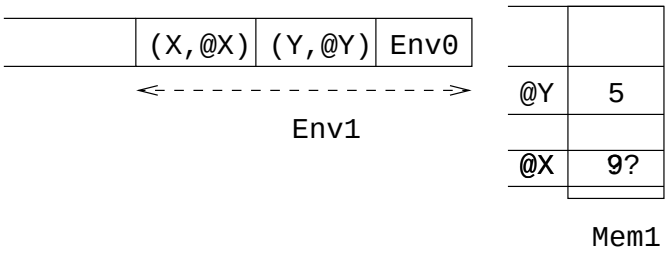
<expression_gauche> dénote une adresse

<expression droite> dénote une valeur

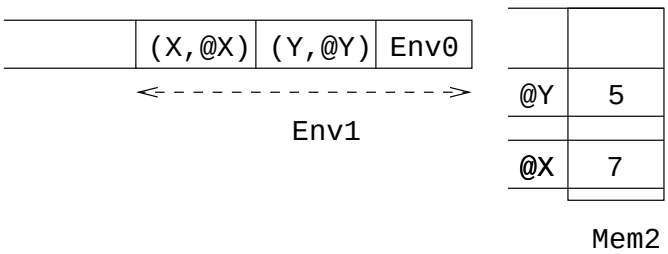
Sémantique : exemple Evaluation de l'affectation

```
X:=Y+2;
```

dans l'état1



- L'évaluation de *<expression_gauche>* (c'est à dire `X`) dans `état1` retourne l'adresse de `X : @X`
- L'évaluation de *<expression_droite>* (c'est à dire `Y+2`) dans `état1` nécessite d'abord l'évaluation des 3 symboles `Y, +, 2`.
  1. La valeur de `Y` est : `@Y`, la valeur de la variable est donc `5`
  2. *2* est une constante littérale, elle s'évalue en elle-même: `2`
  3. La valeur de `+` est sa fermeture, l'environnement d'exécution peut donc etre constitué et son code binaire exécuté.
  4. Le résultat de l'évaluation de `Y+2` est : `7`
- La valeur obtenue est enregistrée à l'adresse de `X`



Remarque L'ordre d'évaluation des expressions gauche et droite est indéterminé.

7.7.2 Boucle for

Syntaxe

```
<boucle_for>::=
for <ident_de_compteur> in [reverse] <intervalle>
loop
    <suite _instructions>
end loop;
<intervalle>::=<ident_de_type>|<début> .. <fin>
```

où

<début> et *<fin>* sont des valeurs d'un type discret

reverse est facultatif. Il indique un parcours inverse (dans l'ordre décroissant) de l'intervalle *<début> .. <fin>*

Sémantique Pour chaque valeur *val* de l'intervalle discret parcouru dans l'ordre croissant (*in*) ou décroissant (*in reverse*):

1. adjonction de la liaison (*ident_ de_ compteur,val*) dans l'environnement courant
2. exécution de la suite d'instructions dans le nouvel état
3. suppression de la liaison *(ident_ de_ compteur,val)* dans l'environnement courant

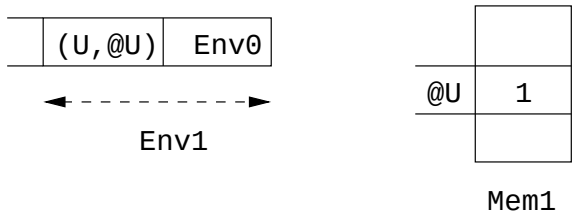
Remarques

- L'identificateur de compteur est une constante pendant l'exécution de la suite d'instructions
- L'identificateur de compteur est local à la boucle. Il est détruit après l'exécution de la boucle

7.7.3 Exemple

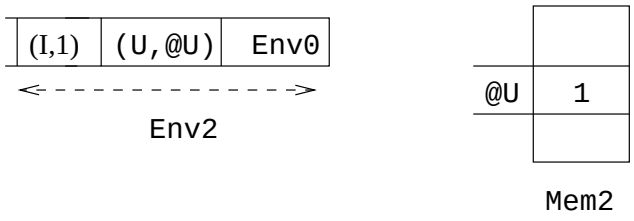
```
-- dans l'état (Env1,Mem1)
for I in 1..3 loop
  U:=I*U;
end loop;
```

état initial



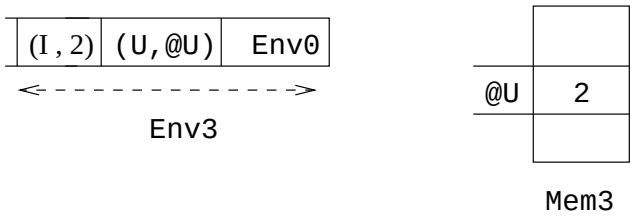

```
  for i in 1..3 loop
    U:=I*U;
  end loop;

-- état avant le second
-- passage dans la boucle
```



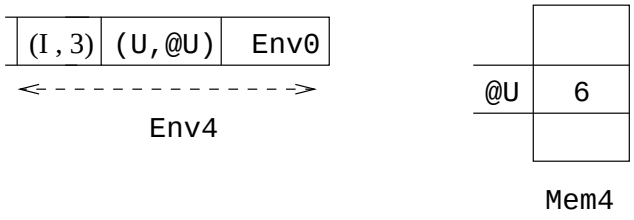

```
  for i in 1..3 loop
    U:=I*U;
  end loop;

-- état avant le troisième
-- passage dans la boucle
```

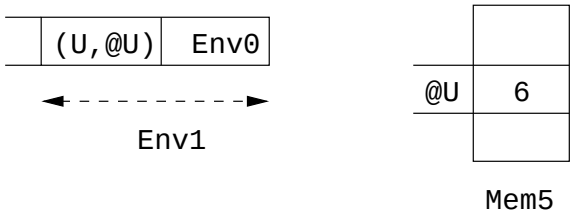



```
  for i in 1..3 loop
    U:=I*U;
  end loop;

-- état avant la sortie
-- de boucle
```




```
-- état après la sortie de la
-- boucle
-- retour à l'environnement Env1
```



7.7.4 Importation de module

- Tout programme s'exécute dans un environnement constitué d'objets informatiques. Par exemple, il est souvent utile de disposer des opérateurs arithmétiques $+$, -, $*$, $/$, ..., de commandes de lecture et d'écriture.

- Tout programme bénéficie d'un environnement initial standard (les opérateurs arithmétiques en font partie).

- Tout programme commence par la définition d'un environnement de travail de base. Celui-ci est construit par importation d'unités à partir de bibliothèques Ada.

Un *environnement* est une liste de couples qui associent un nom d'objet (identificateur) à sa valeur.

L'importation d'une unité est réalisée par la construction Ada :

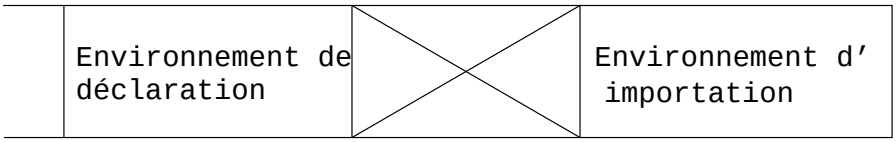
<importation_module>::=with <nom_unite>;

par exemple,

```
with Ada.Text_io;
```

Le couple (*Ada.Text_io,val_Ada.Text_io*) appartient maintenant à l'environnement de l'unité importatrice.

L'environnement d'un module (unité) est constitué de 2 parties :



7.7.5 Environnement de déclaration et d'importation

Environnement de déclaration C'est la liste des objets déclarés dans le module. Un module est lui-même un objet.

Environnement d'importation Il est constitué de le liste des objets déclarés dans le module importé (appartenant donc à l'environnement de déclaration du module importé).

Remarque L'environnement de déclaration a priorité sur l'environnement d'importation

Exemple Soit l'environnement du module `M`

environnement de déclaration	X	environnement d'importation
(A,val_A1)(B,val_B1)(Ada.Text_io,val_T)	X	(B,val_B2)(A,val_A2)

Il existe deux objets d'identificateur A et deux objets d'identificateur B dans l'environnement du module M, les valeurs associées à A et B dans le module M seront donc respectivement *val_A1* et *val_B1*, puisque l'environnement de déclaration a priorité sur l'environnement d'importation.

Exemple 1

```
------------ déclaration -----------
with Ada.Text_io;
-- Env1:(Ada.Text_io,val_text_io)</Env0
procedure debut is
  x:Character:='$';
-- Env2:(x,@x)(Ada.Text_io,val_text_io)</Env0
begin
  Ada.Text_io.put("Hello World");
  Ada.Text_io.new_line;
  Ada.Text_io.put("-----------");
  Ada.Text_io.new_line;
  Ada.Text_io.put_line(X);
end debut;
```

Exemple 2

```
------------ déclaration -----------
with Ada.Text_io;
-- Env1:(Ada.Text_io,val_text_io)</Env0
-------------importation----------------
use Ada.Text_io;
-- environnement du module Ada.Text_io
-- Env import:(put, val_put)(new_line,val_new_line)
--  ....
procedure debut is
   x:Character:='$';
Env: (x,@x)(Ada.Text_io,val_text_io)</Env0
     |
     (put, val_put)(put_line,val_put_line)(new_line,val_new_line)
begin
  put("Hello World");
  new_line;
  put("-----------");
  new_line;
  put_line(X);
end debut;
```

Remarques

- 3 sortes de composants dans le texte :
  1. mots réservés : `with, use, procedure, is, begin, end`
  2. identificateurs : `Ada.Text_io, debut, put, put_line, new_line, X`
  3. valeurs : `"Hello World", "---------"`
- Une structure rigide du programme : la structure de bloc

7.8 Les fonctions

7.8.1 Déclaration d'une fonction : sémantique

Exemple 1

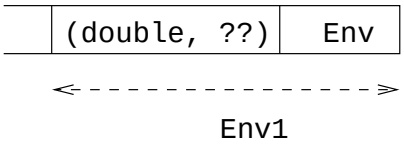
```
function double(X:Integer) return Integer;
```

Mécanisme d'évaluation de cette déclaration de fonction dans l'environnement `Env`.

- Création de la liaison

```
(double,??)
```

- Extension de `Env` avec cette liaison



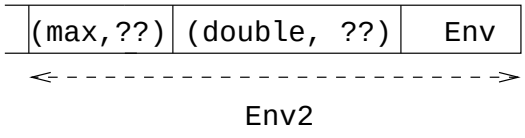
- Détermination du type des paramètres et du type du résultat. Le type de la fonction est :

```
Integer-->Integer
```

Après le nouvelle déclaration

```
function max(A,B:Integer) return Integer;
```

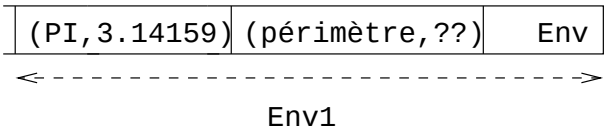
l'environnement devient



Exemple 2

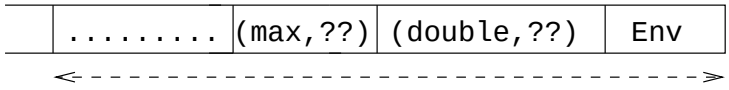
```
function perimetre(A:Float) return Float;
PI:constant Float:=3.14159;
```

Ces déclarations, réalisées dans un environnement courant `Env`, produisent le nouvel environnement `Env1`.



7.8.2 Déclaration du corps d'une fonction : sémantique

Exemple 1 Déclaration du corps de `max` dans un environnement `EnvX` .



EnvX

```
function max(A,B:Integer) return Integer is
begin
  if (A>B)
  then
    return A;
  else
    return B;
  end if;
end max;
```

Mécanisme d'évaluation de cette déclaration :

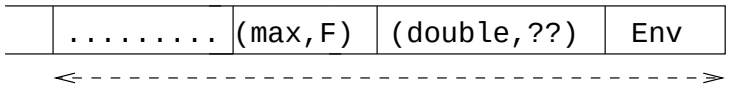
- Détermination de la fermeture `F` de `max` (la valeur de `max`), dans l'environnement `EnvX`, à partir du corps de la fonction

  `F=<<A,B-->corps de max,EnvX>>`

- Modification de la liaison de l'environnement `EnvX`

Remarques

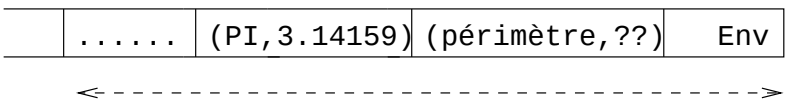
1. L'environnement courant `EnvX` au moment de la déclaration du corps d'une fonction n'est, en général, pas le même que celui de sa déclaration.
2. La liaison `(max,??)` est présente dans l'environnement au moment du calcul de sa fermeture.



EnvX

F=<<A,B--> corps de max, EnvX>>

Exemple 2 Déclaration du corps de `perimetre` dans un environnement `EnvY`.



EnvY

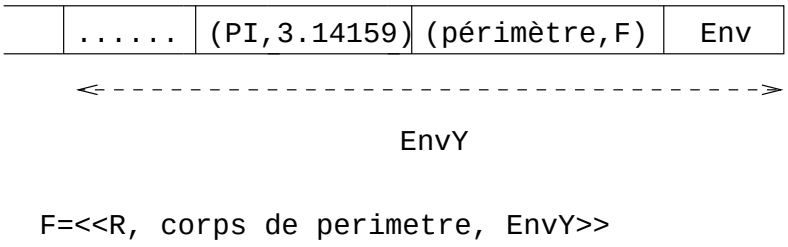
```
function perimetre(R:Float) return Float is
begin
return 2.0*PI*R;
end perimetre;
```

Mécanisme d'évaluation de cette déclaration

- Détermination de la fermeture `F` de perimetre (la valeur de `perimetre`), dans l'environnement `EnvY`, à partir du corps de la fonction :

```
F=<<R-->corps de perimetre,EnvY>>
```

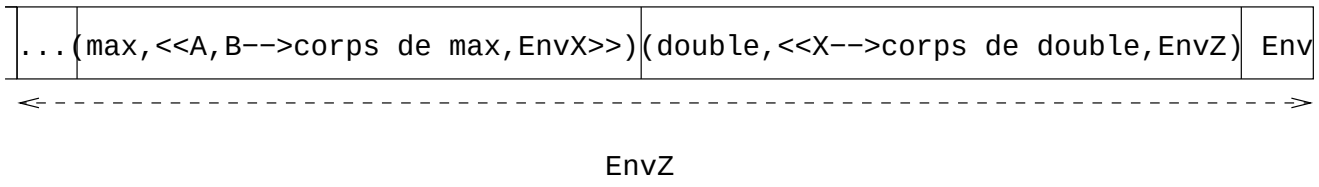
- Modification de la liaison de l'environnement `EnvY`



Remarque La constante `PI` n'était pas dans l'environnement de la déclaration de `perimetre`, mais elle est dans sa fermeture.

7.8.3 Application d'une fonction : sémantique

Exemple 1 Après déclaration du corps de la fonction `double` dans l'environnement `EnvZ`, on obtient



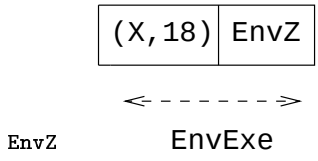
Examinons maintenant le mécanisme déclenché lors de l'appel dans un état `(EnvK,MemK)` obtenu après d'éventuelles modifications de l'environnement `EnvZ` et/ou de la mémoire :

```
double(18)
```

Evaluation de `double` et `18`

- L'évaluation de `double` retourne la fermeture `<<X->Corps _double,EnvZ>>`
- `18` s'évalue en lui-meme

Construction de l'environnement d'exécution `EnvExe` du corps de `double` à partir de



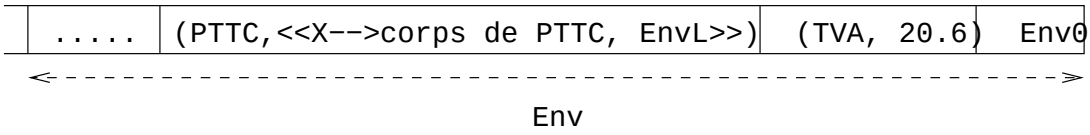
Exécution du corps de *`double`* dans l'état *`(EnvExe,MemK)`* La valeur retournée par la fonction `double` est: `36`

Destruction de `EnvExe` et restitution de l'environnement initial `EnvK`

Exemple 2

```
with Ada.Float_Text_io;
procedure calculPrix is
  TVA : constant Float:=20.6;
  function PTTC(X:Float) return Float;
  function PTTC(X:float) return float is
  begin
    return X+0.01*TVA*X;
  end PTTC;
begin
  Ada.Float_Text_io.put(PTTC(10.0));
end calculPrix;
```

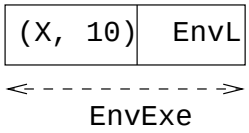
Evaluation de `put(PTTC(10.0))` dans l'état `(Env,Mem)`



- Evaluation de `PTTC` et `10`

```
PTTC => <<X-->Corps de PTTC,EnvL>>
10 => 10
```

- Construction de l'environnement d'exécution `EnvExe` pour l'appel de `PTTC(10.0)` à partir de `EnvL`



- Exécution du corps de `PTTC` dans l'état `(EnvExe,Mem)`

```
=>  X+0.01*TVA*X = 12.06
```

- Destruction de `EnvExe` et restitution de l'environnement initial `Env`
- Evaluation de `put` dans `(Env,Mem)`
- Récupération de la fermeture de `put`

```
put=><<X-->corps de put,EnvPut>>
```

- Constitution de l'environnement d'exécution pour l'appel `put(PTTC(10.0))`: le paramètre formel de `put` prend la valeur de `PTTC(10.0)`, c'est à dire `12.06`
- Exécution de la procédure : affichage de la valeur `12.06` sur l'écran
- Destruction de l'environnement d'exécution et restitution de l'environnement intial `Env`

Remarque Bien que nous ayons choisi, dans cet exemple d'évaluer `PTTC` avant `put`, l'ordre d'évaluation de `put` et `PTTC` est indéterminé en Ada.

7.9 Les procédures

Soient les déclarations :

```
procedure permuter(a:in out Character;b:in out Integer);
```

dans l'environnement `Env` de l'état courant.

et

```
procedure permuter(a:in out Character; b:in out Integer) is
  z:Integer;
begin
  z:=Character'pos(a);
  a:=Character'val(b);
  b:=z;
end permuter;
```

dans l'environnement `Env'` de l'état courant.

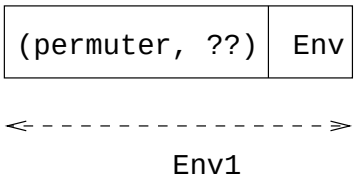
7.9.1 Sémantique de la déclaration (1)

Mécanisme d'évaluation de cette déclaration de procédure dans l'environnement `Env`.

- Création de la liaison

```
(permuter,??)
```

- Extension de `Env` avec cette liaison : on obtient le nouvel environnement `Env1`



- Détermination du type des paramètres et du type du résultat. Le type de la fonction est :

```
Character*Integer-->vide
```

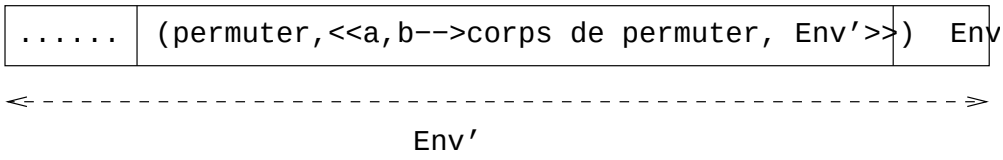
7.9.2 Sémantique de la déclaration du corps de `permuter`(2)

Mécanisme d'évaluation de cette déclaration

- Détermination de la fermeture `F` de permuter, dans un environnement `Env'`, à partir du corps de la procédure

```
F=<<a,b-->Corps de permuter,Env'>>
```

- Modification de la liaison de l'environnement `Env'`



Remarques

1. L'environnement courant `Env'` au moment de la déclaration du corps d'une procédure n'est, en général, pas le même que celui de sa déclaration (`Env`).
2. La liaison `(permuter,??)` est présente dans l'environnement au moment du calcul de sa fermeture.

7.9.3 Sémantique de l'appel de procédure

Reprenons l'exemple

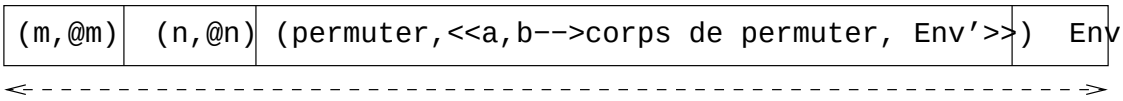
```
with Ada.Text_io;with Ada.Integer_Text_io;
use Ada.Text_io;use Ada.Integer_Text_io;
procedure car_int is
  m:Character:='*';
  n:Integer:=65;
begin
  put(" m=");put(m);
  put("n=");
  put(n,width=>2);new_line;
  permuter(m,n);
```

```
-- ou permuter(b=>n,a=>m);
  put("permuter(m,n);");new_line;
  put(" m=");put(m);
  put("n=");
  put(n,width=>2);new_line;
end car_int;
```

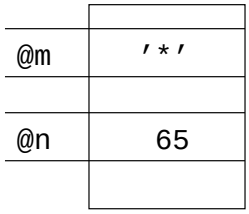
Résultat de l'exécution

```
-------------------------
m=* n=65
permuter(m,n);
m=A n=42
-------------------------
```

Etat avant l'appel `permuter(m,n)` : `(EnvAppel,Mem)`



EnvAppel

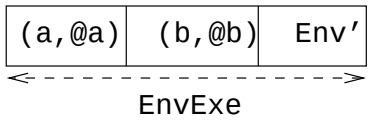


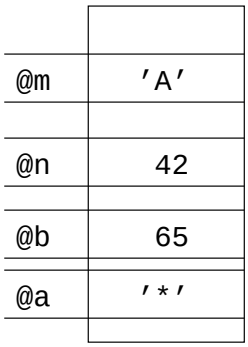
Mem

Appel

```
permuter(b=>n,a=>m);
```

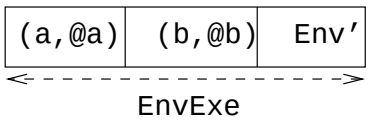
Etat initial de l'appelé (après transmission des paramètres) : `(EnvExe,Mem1)`

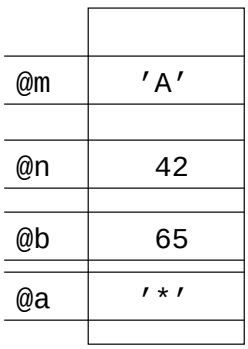




Exécution du corps de la procédure

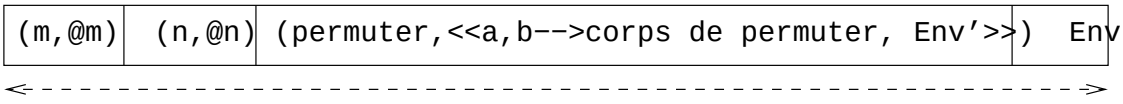
```
z:=Character'pos(a);
a:=Character'val(b);
b:=z;
```



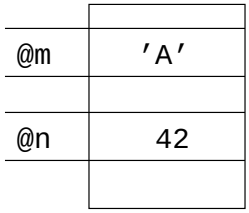


Etat final de la procédure appelée `(EnvExe',Mem2)`

Etat après retour vers l'appelent de la procédure `(EnvAppel',Mem3)`



EnvAppel'



Mem3

Chapter 8

Exceptions

8.1 Qu'est-ce qu'une exception

Une exception est une erreur ou bien une action inattendue qui se produit durant l'exécution d'un programme.

Une exception se traduit par un signal qui interrompt le déroulement normal d'un programme.

En Ada, une exception est un signal nommé. Dès qu'elle est levée, elle interrompt le traitement en cours jusqu'à ce qu'elle soit récupérée et éventuellement traitée.

8.2 Point de vue théorique

D'un point de vue théorique, on peut assimiler un programme au calcul d'une fonction en mathématiques.

Tout programme établit une correspondance entre un ensemble de définition ou domaine (les données du programme) et un ensemble image ou codomaine (les résultats).

Or, cette assimilation ne vaut que pour les fonctions totales, partout définies. Rien interdit à un programme de s'exécuter sur des valeurs particulières (données) pour lesquelles la fonction n'est pas définie (exemple : division par zéro).

Le mécanisme des exceptions dans les Langages de Programmation (LP) tels que Java, C++, Ada permet de prendre en compte les cas des données exceptionnelles (inattendues) de manière à construire des programmes robustes.

Par exemple, lorsqu'un programme demande à l'utilisateur de saisir des données numériques dans un certain intervalle, il doit, pour être robuste, déterminer si les données sont numériques et si elles appartiennent à l'intervalle. Dans le cas contraire, le programme doit prévoir une action spécifique; ici, demander une nouvelle saisie des données.

8.3 Gestion traditionnelle des cas d'erreur

Dans les langages qui ne possèdent pas de mécanisme de traitement des exceptions, on utilise les structures de contrôle classiques pour gérer les cas d'erreur.

Exemple lecture d'un entier dans un pseudo-Ada qui ne supporte pas les exceptions

```
get(X);
```

Pour rendre robuste le programme qui utilise cette procédure, il faut vérifier que la donnée lue est bien un entier valide.

Une première méthode est de modifier la méthode `get` en lui passant un paramètre qui reflète l'état de la saisie.

```
declare
  etat:Boolean;
  X:Integer;
begin
  get(X,etat);
  if not etat then put("erreur de saisie");end if;
end;
```

3 inconvénients à cette méthode :

1. il faut rajouter un paramètre supplémentaire à la méthode `get`
2. il faut déclarer une variable supplémentaire : `etat`
3. il faut ajouter un test de cette variable alors que dans la grande majorité des cas, cette variable aura pour valeur `true`.

Un schéma qui rend encore plus difficile cette approche :

```
declare
  type Point is
    record
      abs:Integer;
      ord:Integer;
    end record;
  P:Point;
  procedure get(X:out Point;C:out Integer) is
     C1,C2:Integer;
  begin
    get(X.abs,C1);
    case C1 is
      when -1=>C:=-1;
   -- les caractères lus ne sont pas numériques
      when -2=>C:=-2;
   -- dépassement de capacité
      when others=>NULL;
    end case;
    get(X.ord,C2);
     case C2 is
      when -1=>C:=-3;
   -- les caractères lus ne sont pas numériques
      when -2=>C:=-4;
   -- dépassement de capacité
      when others=>NULL;
    end case;
  end get;
  etat:Integer;
  unPoint:Point;
begin
```

```
  loop
  get(unPoint,etat);
     case etat is
      when -1=> put("abscisse non numérique");
      when -2=> put("abscisse:");
                put("dépassement de capacité");
      when -3=> put("ordonnée non numérique");
      when -4=> put("ordonnée:");
                put("dépassement de capacité");
       when others=>exit;
    end case;
     skip_line;
  end loop;
end;
```

On suppose ici que l'état est propagé vers une méthode appelante sachant traiter ces cas d'erreurs.

8.4 Intérêt

Il s'agit de contrôler le déroulement des calculs en localisant des évènements exceptionnels pour:

- traiter des erreurs
- arrêter un calcul

et de rendre plus lisible et plus structuré le texte d'une unité en séparant traitement du cas général et traitement des cas exceptionnels.

8.5 Exceptions prédéfinies

Elles sont présentes dans l'environnement initial et levées :

- en cas de manque d'espace mémoire (STORAGE_ERROR)
- en cas de tentative de violation d'une structure de contrôle ou l'appel d'une procédure non encore élaborée (PROGRAM_ERROR)
- si la valeur d'une expression dépasse les bornes de l'ensemble des valeurs possibles (type) (CONSTRAINT_ERROR)

8.6 Définition

Une exception est un signal nommé. Dès qu'elle est levée, elle interrompt le traitement en cours jusqu'à ce qu'elle soit récupérée.

8.7 Déclaration d'une exception

8.7.1 Syntaxe

<déclaration_exception>::=<identificateur_exception>:exception;

8.7.2 Sémantique

L'identificateur est introduit dans l'environnement de l'état courant.

8.7.3 Exemple

```
erreur,div_par_zero:exception;
```

8.8 Levée d'une exception

Lever une exception, c'est signaler qu'une situation anormale est détectée.

La levée d'une exception est la seule opération associée au type *exception*.

8.8.1 Syntaxe

<levée_exception>::=raise <identificateur_exception>;

8.8.2 Sémantique

L'exécution de l'instruction `raise` interrompt le déroulement normal du code et transfert le contrôle à un traitement de cette exception qui peut :

- être prédéfini (cas des exceptions prédéfinies)
- être codé par le programmeur
- consister en une propagation du signal vers les unités appelantes

8.8.3 Exemple

```
declare
divParZero:exception;
function divise(X,Y:Integer) return Integer is
begin
   if Y=0
   then
     raise divParZero;
   else
     return X/Y;
   end if;
end divise;
begin
...
end;
```

8.9 Récupération d'une exception

Récupérer une exception, c'est se donner la possibilité d'associer un traitement particulier à la situation détectée.

La construction syntaxique ci-dessous complète la structure générale d'un bloc :

```
declare
...
begin
...
exception
...
end;
```

Il s'agit ici d'un bloc non nommé. Rappelons que procedure et fonction forment aussi un bloc.

8.9.1 Syntaxe

```
<récupération_exception>::=
exception
when <identificateur_exception>=><traitement>;
when <identificateur_exception>=><traitement>;
...
when others=><traitement>;
```

8.9.2 Sémantique

Les identificateurs d'exception servent de *filtre* pour l'exécution des traitements.

Ils doivent donc être présents dans l'environnement.

Le filtre `others` permet de filtrer toutes les autres exceptions

8.9.3 Exemple

```
function demain(X:Jour) return Jour is
begin
   return Jour'succ(X);
exception
   when CONSTRAINT_ERROR=> return Jour'first;
end demain;
```

8.10 Portée des exceptions

Une exception est connue par son nom dans son unité de déclaration ainsi que dans toutes les unités englobées.

Si une exception est levée dans une unité, le traitement est interrompu et le contrôle est transféré au traitant d'exception.

Mais si l'exception n'est pas traitée dans cette même unité, alors elle est propagée vers l'unité appelante; le contrôle est passé à l'unité appelante et ainsi de suite jusqu'à l'unité principale.

```
-- exemple 27
declare
  erreur:exception;
  procedure A is
  begin
    C;
  exception
```

```
    when erreur=>traitement_erreur_A;
  end A;
  procedure C is
  begin
    B;
  exception
    when erreur=>traitement_erreur_C;
  end C;
  procedure B is
  begin
     raise erreur;
  end B;
begin
  A;
end;
```

8.11 Propagation d'une exception hors de sa portée

L'exception peut aussi se retrouver en dehors de la portée de sa déclaration.

Elle est, dans ce cas, traitée par le filtre `others`.

```
procedure recupere is
  procedure declenche is
    exLocale:exception;
  begin
    raise exLocale;
  end declenche ;
begin
  declenche ;
exception
  when others=>...;
     --traitement_de_toutes_les_exceptions;
end recupere ;
```

8.12 Exception dans une déclaration

Si une exception est déclenchée dans une déclaration, elle est propagée vers l'unité appelante.

```
declare
  ex: exception;
  function declenche return Integer is
  begin
    raise ex;
  end declenche ;
begin
  declare
    X:Integer:=declenche ;
  begin
    put("interne");
  exception
    when EX=>put("recupere interne");
   end;
exception
```

```
  when EX=>put("recupere externe");
end;
```

Le résultat affiché sera :

```
recupere externe
```

8.13 Exception dans un traitement d'exception

Une exception levée dans un traitement d'exception est propagée vers l'unité appelante.

Pour un traitement couche par couche de l'exception, on peut propager la même exception en la levant de nouveau par `raise`;

Lorsque plusieurs exceptions sont filtrées par `others`, il n'y a pas moyen de nommer explicitement l'exception survenue.

Dans ce cas, il est possible de la lever de nouveau par `raise`;.

```
when others=> put("l'exception filtrée va être de nouveau levée");raise;
```

8.14 Utilisation

8.14.1 Pour améliorer la structure et la lisibilité

```
declare
  N:Integer;
begin
  loop
    begin
      get(N);
      exit;
    exception
      when DATA_ERROR=>
        Ada.Text_io.skip_line;
        Ada.Text_io.put_line("Recommencez!");
    end;
  end loop;
end;
```

`DATA_ERROR` est une autre exception prédéfinie. Elle est levée lorsque la valeur lue ne correspond pas au type de sa variable d'accueil.

8.14.2 Pour contrôler des calculs : exception et récursion

```
declare
  ex:exception;
  function fac(N:Integer) return Integer is
  begin
    if N=1
    then
      raise ex;
    else
```

```
      return N*fac(N-1);
    end if;
  exception
    when ex=>
      Ada.Integer_Text_io.put(N);raise;
  end FAC;
begin
  put(FAC(3));
exception
  when ex=>Ada.Text_io.put_line("stop");
end;
```

Le résultat affiché sera :

```
1       2       3       stop
```

8.14.3 Récursion (suite)

Que se passe-t-il si on ne propage pas l'exception par l'instruction `raise`.

```
declare
  ex:exception;
  function F(N:Integer) return Integer is
  begin
    if N=1
    then
      raise ex;
    else
      return N*F(N-1);
    end if;
  exception
    when ex=>
      Ada.Integer_Text_io.put(N);
  end F;
begin
  put(F(1));
exception
  when ex=>Ada.Text_io.put_line("stop");
end;
```

Résultat de l'exécution

```
1
raised PROGRAM_ERROR : fonction.adb:7
```

Puisque l'exécution de `F(1)` se termine par un traitement d'exception non propagée, la fonction doit retournée une valeur de type `Integer` pour que l'exécution se poursuive normalement. Le compilateur affiche des *"warnings"* pour signaler que l'instruction `return` manque à la fin du traitant d'exception de la fonction `F` et que l'exception `PROGRAM_ERROR` pourra être levée.

Dans l'exemple précédent l'exécution se poursuivait jusqu'à la fin, de traitant d'exception en traitant d'exception. Dans ce cas tout se passe comme si l'appel à la fonction `FAC` se terminait avec le programme.

8.15 Exemple détaillé

8.15.1 Réalisation d'un programme de "login" sur une machine hôte.

La procédure `login` affiche tout d'abord l'invite :

```
login :
```

L'utilisateur tape alors son nom d'utilisateur.

Si ce nom est inconnu du système, la tentative est arrêtée, l'exception `login_errone` est levée et propagée.

Si le nom est connu du système, la procédure affiche :

```
password :
```

et l'utilisateur est invité à taper son mot de passe.

Si le mot de passe est correct, c'est terminé, sinon la procédure `login` imprime un message ad hoc et l'utilisateur doit retaper son mot de passe.

L'utilisateur a trois essais, à la suite desquels, l'exception `login_errone` est levée et propagée.

Les procédures suivantes appartiennent à l'environnement d'exécution de la procédure `login` et sont rassemblées dans le paquetage `test`:

```
procedure teste_nom(N:string);
-- vérifie que N est connu du système.
-- si ce n'est pas le cas, l'exception
--  nom_errone est levée
procedure teste_mot_de_passe(M:string,N:string);
-- vérifie que M est connu et associé à N,
-- sinon l'exception mot_errone est levée
```

8.15.2 Code du programme de "login" pour des données valides

```
with Ada.Text_io;use Ada.Text_io;
with test;use test;

procedure login is
  nom,mot:string(1..8);
begin
  put("login : ");get(nom);
  teste_nom(nom);
  put_line(nom);
  put("password : ");
  for I in 1..3 loop
    get(mot);
    teste_mot_de_passe(nom,mot);
    exit;
  end loop;
end login;
```

8.15.3 Code du programme de "login": cas général

```
with Ada.Text_io;use Ada.Text_io;
with test;use test;

procedure login is
  login_errone:exception;
  nom,mot:string(1..8);
begin
  put("login : ");get(nom);
  teste_nom(nom);
  put_line(nom);
  put("password : ");
  for I in 1..3 loop
    begin
      get(mot);
      teste_mot_de_passe(nom,mot);
      exit;
    exception
      when mot_de_passe_errone=>
         put("mot de passe errone");
         if I<3 then
           put("Essayez encore!");
         else
           raise;
         end if;
    end;
  end loop;
exception
  when nom_errone=>
         put("nom "&nom&" inconnu");
         put("login interrompu");
         raise login_errone;
  when mot_de_passe_errone=>
         raise login_errone;
end login;
```

Chapter 9

Types avancés

9.1 Typage et implantation

9.1.1 Informations associées à un type:

- Son nom
- Une *information abstraite* (domaine abstrait) ensemble de valeurs (abstraites) ensemble d'opérations (abstraites) sur les valeurs (abstraites).
- Une *information physique* (domaine concret) une représentation des valeurs en machine une représentation des opérations en machine (algorithme codé).

9.1.2 Exemple

Le type entier (son nom `Integer`)

Domaine abstrait : l'ensemble des entiers muni des opérations arithmétiques `(+, -, *, /)`

Domaine concrêt : représentation binaire des entiers dans des mots de 32 bits $[-2^{31}, +2^{31} - 1]$, représentation des opérations au moyen des opérations binaires de la machine.

9.1.3 Contraintes physiques

Tous les entiers ne sont pas représentables

Perte de propriétés de l'ensemble abstrait de valeurs

```
216 * 216 = 232 => non représentable
```

Perte de l'associativité: si l'entier maximum représentable est `1000`,

```
600+500+(-400)=600+(500-400)=600+100=700
```

est un calcul correct, alors que

```
600+500+(-400)=(600+500)-400
```

lève une erreur, car `1100` n'est pas représentable, on dit qu'il y a *débordement*

9.2 Notion de sous-type

On peut restreindre l'ensemble des valeurs que peut prendre un objet:

```
nb_jour_mois:Integer range 28..31;
```

9.2.1 Définition

Un sous type définit un sous ensemble particulier des valeurs d'un type appelé type de base

Les valeurs du sous type appartiennent à un sous-ensemble de l'ensemble des valeurs du type de base

Les opérations sur les valeurs du sous type sont les opérations sur les valeurs du type de base

9.2.2 Déclaration de sous type

```
<déclaration_sous_type>::=
subtype <identificateur_sous_type> is <type_de_base>range<contrainte>;
```

9.2.3 Exemples

```
subtype NbJours is Integer range 1..31;
subtype NbJoursMois is NbJours range 28..31;
subtype JoursMois is NbJours;
```

Remarque Les bornes des intervalles ne sont pas nécessairement statiques, l'ensemble des valeurs d'un sous type n'est donc pas forcément connu à la compilation.

```
with Ada.Integer_Text_io;use Ada.Integer_Text_io;
with Ada.Text_io;use Ada.Text_io;
procedure soustype is
  N:Integer;
begin
  N:=4;
  declare
    subtype Jours is Integer range 1..N;
    X:Jours;
  begin
    X:=2;
    put(X);
  end;
end soustype;
```

Remarque Pour que `N` puisse être évalué dynamiquement, il faut qu'il soit au préalable référencé. Cela implique que le sous type `Jours` soit déclaré dans un bloc interne.

9.3 Notion de type dérivé

9.3.1 Intérêt

```
V:Float:=221.2; T:Float:=64.5;
V:=V*T;
-- opération licite, mais aucune vérification
-- sémantique sur les valeurs
```

Dans cet exemple, `V` représente une vitesse alors que `T` représente un temps. L'affectation `V:=V*T` est correcte au niveau du typage, mais on voit bien qu'elle n'a pas de sens si l'on considère l'équation aux dimensions $(m/s \neq (m/s) * s)$.

La notion de *type dérivé* permet d'introduire une sémantique plus fine des objets manipulés.

9.3.2 Déclaration de type dérivé

```
<déclaration_type_dérivé>::=
type <identificateur_type_dérivé> is new <définition_type_dérivé>;
```

9.3.3 Exemple

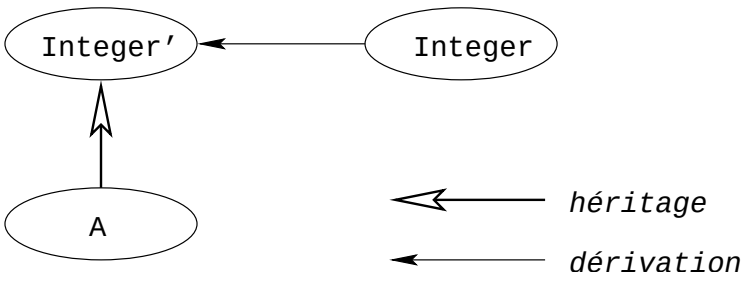
```
type Vitesse is new Float;
type Temps is new Float;
V:Vitesse:=221.2;
T:Temps:=64.5;
V:=V*T;
-- opération illicite,erreur de typage
```

9.4 Construction d'un type dérivé

9.4.1 Exemple

```
type A is new Integer range 1..10;
```

1. Création d'un nouveau type de base, dérivé de `Integer`
2. Création d'un sous type de ce nouveau type de base avec les contraintes de la définition du sous type.



9.5 Définition d'un type dérivé

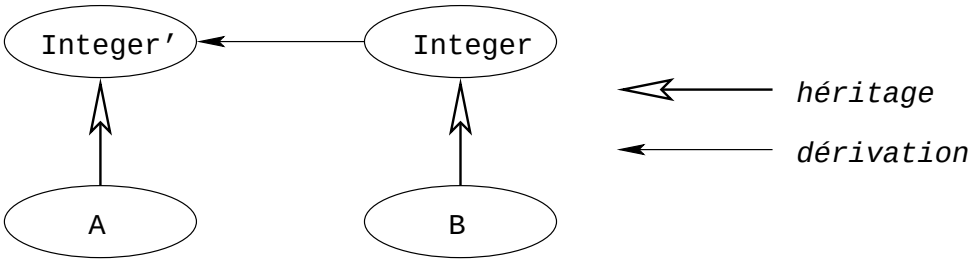
9.5.1 Définition

Un *type dérivé* définit un nouveau type qui possède le même domaine abstrait et le même domaine concret que le type dont il dérive, appelé le type père.

- L'ensemble des valeurs du type est une copie de l'ensemble des valeurs du type père
- Les opérations sur les valeurs du type dérivé sont identiques aux opérations sur le type père

Remarques

```
subtype B is Integer range 1..31;
-- sous type
type A is new Integer range 1..31;
-- type dérivé,sous type d'Integer
```



9.6 Compatibilité entre types

```
declare
  X:Integer range 1..100;
  -- sous type anonyme
  subtype Jour is Integer range 1..31;
  -- définition de sous type
  type Semaine is range 1..7;
  -- définition de type dérivé
  Y:Jour;
  Z:Semaine ;
begin
  X:=5;
  Y:=X;
  -- affectation licite car héritée de Integer
  Z:=X;
  -- affectation illicite entre type
  -- et type dérivé
  Z:=Y; -- idem
  Y:=53;-- CONSTRAINT_ERROR
end;
```

9.7 Conversion de types

9.7.1 Notion de conversion

Transformation d'un valeur `V` d'un type en une valeur `V'` d'un autre type

Cas nŕ1 Le domaine concret du type de `V` est inclus dans celui du type de `V'`.

Exemple :

1. Le domaine concret de [1..10] appartient au domaine concret de $\aleph$ ($[-2^{31}, +2^{31} - 1]$).
2. Les représentations des valeurs de [1..10] et celles de $[-2^{31}, +2^{31} - 1]$ sont identiques

Cas nř2 Le domaine abstrait du type de `V` est inclus dans celui du type de `V'`.

Exemple : $\aleph \subset \Re$

Il faut transformer la représentation de$\aleph$ en celle de $\Re$

Cas nř 3 Le domaine abstrait du type de `V` contient celui du type de `V'`.

Exemple : $\Re \supset \aleph$

Pour transformer la représentation de $\Re$ en celle de $\aleph$, il faut passer par une approximation.

9.8 Conversions implicites et explicites

9.8.1 Conversions implicites

- Entiers vers type entier
- Réels vers type réel

9.8.2 Conversions explicites

Elles sont réalisées par la construction :

```
<conversion>::=<type_destination>(<expression_source>);
```

- Entre types numériques

```
Float(2*J)  Integer(3.14)
```

- Entre types dérivés

```
type Point is new Complexe;
C:Complexe:=(3.5,6.2);
P:Point ;
P:=Point(C);
C:=Complexe(P);
```

- Entre tableaux

  1. même dimension,
  2. même type d'indice,
  3. composants même type

```
type T1 is array(1..5) of Integer;
type T2 is array(1..5) of Integer;
X:T1:=(5,4,8,9,3);
Y:T2:=(1,4,9,0,6);
X:=T1(Y);
Y:=T2(X);--conversion possible
```

9.9 Type article paramétré

Un type article peut être paramétré.

Les paramètres d'un type *article* sont des *champs* particuliers appelés *discriminants.*

Une dépendance est établie entre les discriminants et certains autres champs

Cela permet de définir des types *article* dont la structure est fonction de paramètres (les discriminants)

Certains champs du type sont donc variants : ils dépendent de la valeur du paramètre.

9.9.1 Déclaration d'un type article paramétré

```
<déclaration_type_paramétré>::=
type <identificateur_type>
(<discriminant>:<type_discret>[:=<expression>]
{;<discriminant>:<type_discret>[:=<expression>]}) is
record
 <champ>:<type>[:=<expression>];
{<champ>:<type>[:=<expression>];}
end record;
```

Exemple

```
declare
  SMIC:constant Float:=5972.0;
  type Employe(Borne:Positive:=9) is
    record
      nom:string(1..Borne);
      naissance:Date;
      salaire:Float:=SMIC;
    end record;
begin
  ...
end;
```

9.9.2 Valeur d'un type article paramétré

Les discriminants d'une valeur du type paramétré ne peuvent pas avoir une valeur indéterminée.

```
directeur:Employe(Borne=>6);
directeur.nom:="Patron";
-- nom sur 6 caractères
marcel:Employe:="Alexandre";
-- nom sur 9 caractères
```

9.9.3 Accès à la valeur d'un discriminant

On accède à la valeur d'un discriminant de la même manière qu'à n'importe quel champ.

Exemple

```
declare
  type Paire(Borne:Integer:=100) is
    record
      gauche,droite:Integer range 1..Borne;
    end record;
  P:Paire;
begin
  Ada.Integer_Text_io.put(P.Borne); --imprime 100
  Q:Paire(50); -- sous type de Paire
end;
```

- Modification de la valeur d'un discriminant

```
Q.Borne:=30; -- illégal
Q.Borne:=(30,4,25); --légal modification globale
```

- Affectation entre valeurs d'un type paramètré

```
Q:=P; -- illégal car P n'est pas un sous-type de Q
```

9.10 Type article avec partie variante

Il est possible de définir des types dans lesquels une partie de la structure est fixe et l'autre variante.

Le choix de la partie variante, qui détermine la structure effective des valeurs du type, dépend de la valeur d'un discriminant.

Ces types reflètent la somme disjointe d'une famille d'ensembles de valeurs.

9.10.1 Syntaxe

```
<déclaration_type_variant>::=
type<identificateur_de_type>
         (<etiquette>:<type_discret>)is
record
  case
   when <choix>{ | <choix>} =><liste_de_champs>;
  {when <choix>{ | <choix>} =><liste_de_champs>;}
  end case;
end record;
```

Exemple

```
type Genre is(masculin,feminin);
type Personne(sexe:Genre) is
  record
    naissance:Date;
    case sexe is
      when masculin=>barbe:Boolean;
      when feminin=> enfants:Natural;
    end case;
  end record;
```

Tous les objets du type sont contraints car aucune valeur par défaut n'est donnée au discriminant

```
paul:Personne(masculin);
jeanne:Personne(feminin);
```

ou

```
subtype Homme is Personne(sexe=>masculin);
pierre:Homme;
```

- Affectation globale

```
pierre:=(masculin,(12,2,1950),false);
```

- Accès aux composants

```
pierre.barbe:=true;
```

9.11 Modélisation de données

On veut modéliser les cartes à jouer. Plusieurs solutions s'offrent à nous.

```
declare
  type Couleur is (trefle,carreau,coeur,pique);
  type Figure is(as,roi,dame,valet,petite);
  type Niveau(F:Figure) is
    record
      case F is
        when petite=>v:integer range 7..10;
        when others=>NULL;
      end case;
    end record;
  type Carte is
    record
      le_niveau:Niveau;
      la_couleur:Couleur;
    end record;
  dame_pique:Carte:=(dame,pique);
  neuf_carreau:Carte:=((petite,9),carreau);
  neuf_carreau:Carte:=((petite,9),carreau);
begin
  ...
end;
```

9.12 Filtrage sur les types articles variants

```
declare
  type Valeur is Integer range 0..20;
  function val_belote (atout:Couleur;c:Carte)
                               return Valeur is
  begin
    case c.le_niveau.F is
```

```
      when as=>return 11;
      when roi=>return 4;
      when dame=>return 3;
      when valet=>
          if c.la_couleur=atout
          then return 20;
          else return 2;
          end if;
      when petite=>
          case c.le_niveau.v is
            when 10=> return 10;
            when 9=>
              if c.la_couleur=atout
              then return 14;
              else return 0;
              end if;
            when others=> return 0;
          end case;
    end case;
 end val_belote;
begin
  ...
end;
```

9.13 Autre solution

```
declare
 type Couleur is (trefle,carreau,coeur,pique);
 type Figure is(as,roi,dame,valet,petite);
 type Carte(F:Figure) is
  record
    la_couleur:Couleur;
    case F is
       when petite=>v:integer range 7..10;
        when others=>NULL;
    end case;
  end record;
 un_roi(roi); --couleur variable
 valet_coeur:Carte:=(valet,coeur);
 une_carte:Carte:=(as,carreau);
 une_carte.v:=7; -- CONSTRAINT_ERROR
begin
 ...
end;
```

Chapter 10

Type accès

10.1 Objets dynamiques

10.1.1 Objets statiques

- Ils sont déclarés
- Un nom (identificateur) est associé à leur valeur
- Leur durée de vie est celle de leur bloc de déclaration
- L'allocation de l'espace nécessaire à leur représentation est effectué par le compilateur

10.1.2 Objets dynamiques

- Pas de déclaration
- Pas d'association nom-valeur
- Durée de vie indépendante de la structure de bloc
- Allocation mémoire réalisée dynamiquement

10.2 Accès aux objets dynamiques et types accès

Les objets de type *access* fournissent un accès à d'autres objets éventuellement alloués dynamiquement.

Un objet de type *access* peut être considéré comme une référence ou un pointeur.

10.3 Le type access

10.3.1 Syntaxe

```
<déclaration_type_accès>::=
type <identificateur_type> is access <définition_de_type>;
```

10.3.2 Valeurs d'un type accès

Une déclaration d'un type *access* crée un ensemble d'adresses permettant de référencer les objets d'un type donné.

10.3.3 Exemple

```
type Ref_int is access Integer;
X:Ref_int ;
```

L'ensemble des valeurs du type `Ref_int` est un ensemble d'adresses ne pouvant référencer que des objets de type `Integer` ou d'un sous-type d'`Integer`.

Les valeurs de la variable `X` sont ces adresses.

La valeur `null` est une valeur d'adresse particulière.

Elle ne référence aucun objet.

10.4 Création dynamique d'objet

10.4.1 Le constructeur *new*

<création_objet_dynamique>::=
new <type _objet_référencé>'(<expression>);

- Création d'un objet du type référencé (objet anonyme). Cela implique l'allocation de la ressource mémoire nécéssaire.
- Initialisation de cet objet avec la valeur de l'expression.
- Retourne l'adresse de cet objet.

10.4.2 Le constructeur *.all*

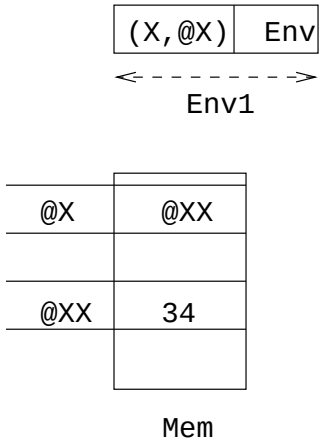
Soit `X` un objet de type accès, pour accéder à la valeur de l'objet référencé par `X`, on utilise la construction `.all`

```
X.all -- désigne l'objet référencé par X
```

10.4.3 Exemple

Sémantique de la déclaration dans l'état `(Env,Mem)`

```
X:Ref_int:=new Integer'(34);
```



2 objets ont été crées :

1. Une variable de type `Ref_int` (pointeur sur un objet de type `Integer`)

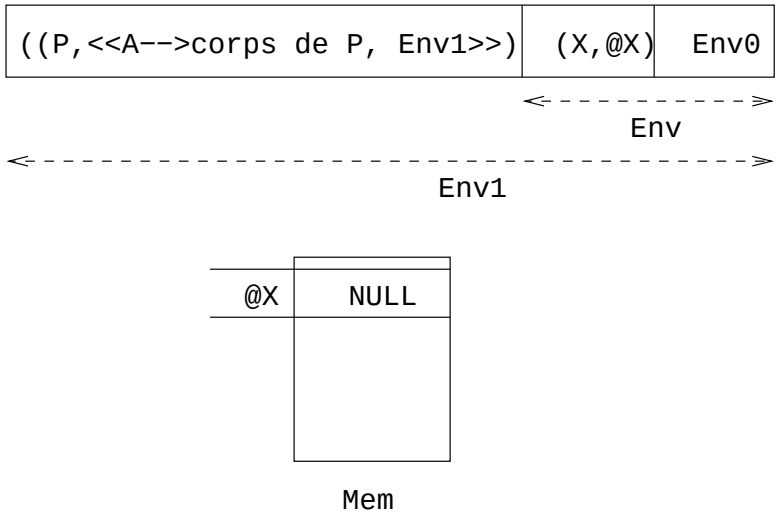
2. Une variable anonyme de type `Integer` et de valeur `4`

La portée de l'objet anonyme (crée par accès) est liée à la portée de la définition du type *access*.

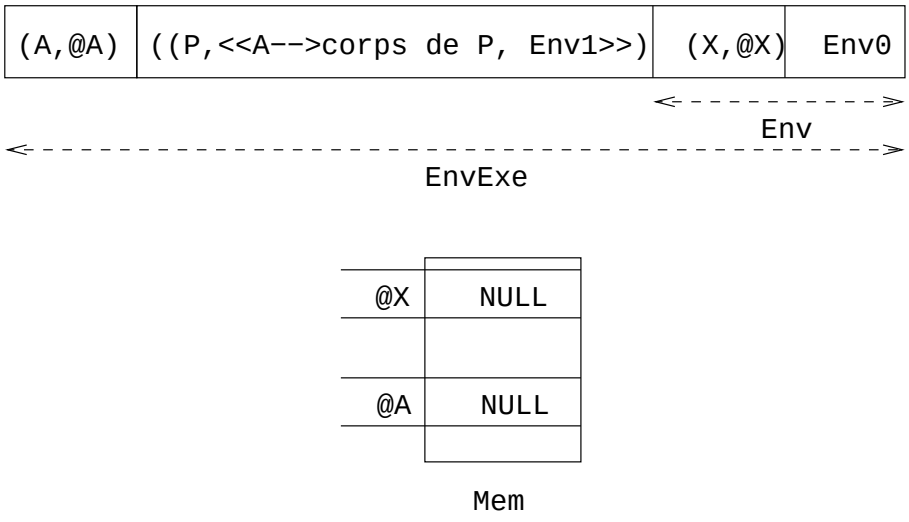
Exemple

```
declare
  type Ref_int is access Integer;
  X:Ref_int:=null;
  procedure P(A:out Ref_int ) is
  begin
  -- état 2 (état initial d'exécution)
    A:=new Integer'(100);
  -- état 3 (état final de l'exécution)
  end P;
-- état1 (après les déclarations)
begin
  P(X);
-- état 4 (état de retour après appel de la procédure
  put(X.all);
end;
```

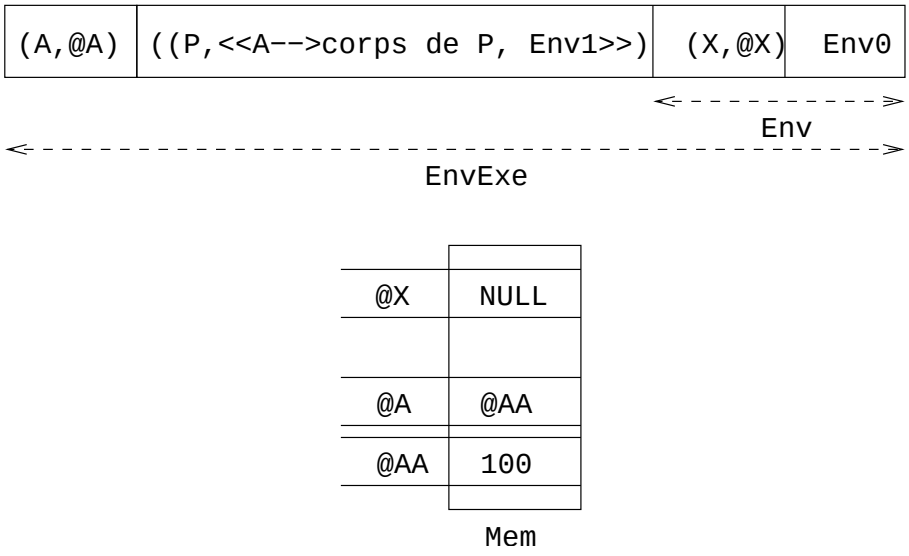
Etat1 après les déclarations



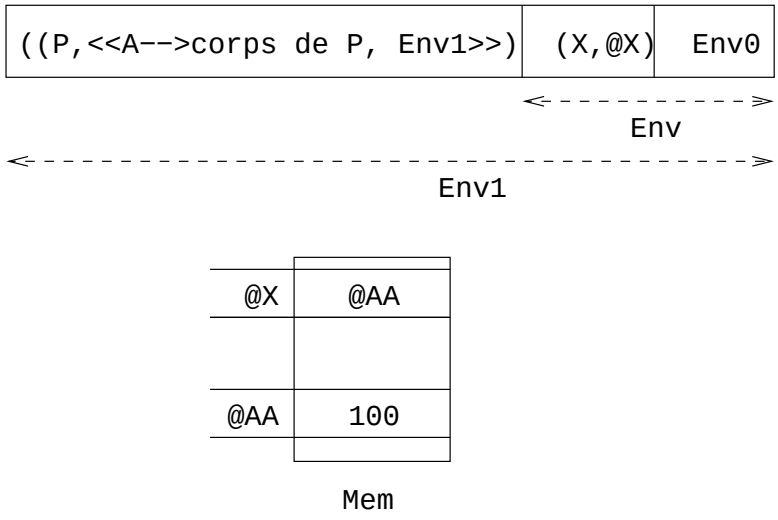
Etat2 état initial d'exécution de la procédure



Etat3 avant transmission du paramètre en retour *(*`out`)

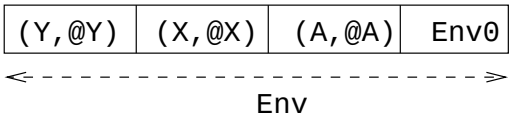


Etat4 après retour vers l'appelant

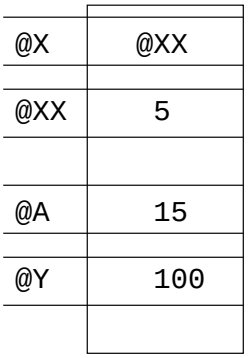


Exemple

```
A:Integer:=15;
X:Ref_int:=new Integer'(5);
Y:constant Ref_int:=new Integer'(100);
Y:=X;
-- affectation à la constante
-- Y=>interdit
X:=Y;
-- remarque : @XX devient inaccessible
X:=A;
-- incompatibilité de types
Y.all:=A;
```



Env



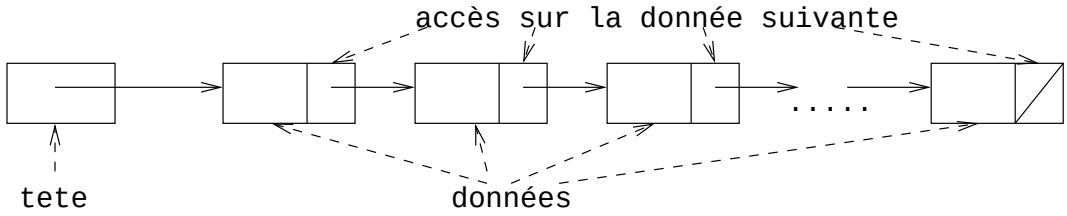
Mem

Chapter 11

Types récursifs : les listes

11.1 Les listes en Ada

Une liste est une structure de données qui relie des données de type quelconque et identique. Bien qu'il existe plusieurs manières de relier les données, on peut en donner une représentation graphique simple.



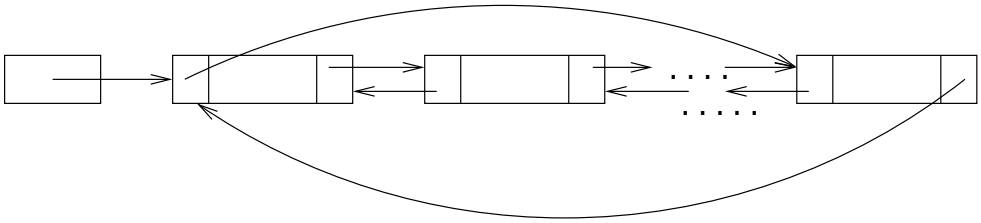
11.1.1 Typage

Pour manipuler ces structures de données en Ada, il faut tout d'abord définir un type. En remarquant que l'ensemble des données chainées ont la même structure, il est possible de définir chaque type de donnée (Cellule) en supposant que le type des suivantes est connu. On parle alors de définition récursive du type.

```
type Cellule;
type Integer_list is access Cellule;
type Cellule is
   record
     valeur:Integer;
     suivant:Integer_list ;
   end record;
```

- Le type `Cellule` est d'abord déclaré, son nom est introduit dans l'environnement associé à une valeur indéfinie.
- Le type `Integer_list` est défini récursivement

On peut imaginer d'autres structures de listes mieux adaptées aux besoins d'une application spécifique.



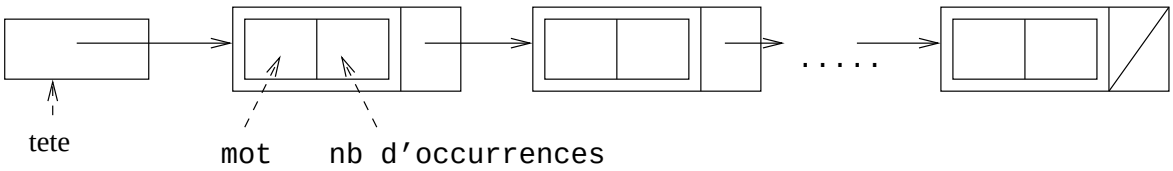
Le type d'une liste doublement chainée est défini comme suit :

```
type Cellule;
type Integer_list is access Cellule;
type Cellule is
   record
     valeur:Integer;
     suivant:Integer_list;
     precedent:Integer_list;
   end record;
```

11.1.2 Interêt

Les listes sont utilisées lorsque ni les données à chainer ni leur nombre ne sont connus a priori. Par exemple, supposons un programme dont le but est d'afficher les mots d'un texte avec leur fréquence d'apparition. Le programme est donc chargé de parcourir le texte et d'ajouter chaque mot nouveau à la liste ou bien, si ce mot est déjà présent dans cette liste, d'incrémenter son nombre d'apparitions dans le texte.

Une structure de données adaptée à ce problème est représentée graphiquement



La définition Ada de ce type liste est :

```
type Donnee(lg:Positive) is
  record
    mot:String(1..lg);
    occ:Natural:=0;
  end record;

  type Cellule;
  type Liste is access Cellule;
  type Cellule(lg:Positive) is
    record
      valeur:Donnee(lg);
      suivant:Liste;
    end record;
```

On remarquera l'utilisation de type article paramétré pour prendre en compte des valeurs de champ de taille variable.

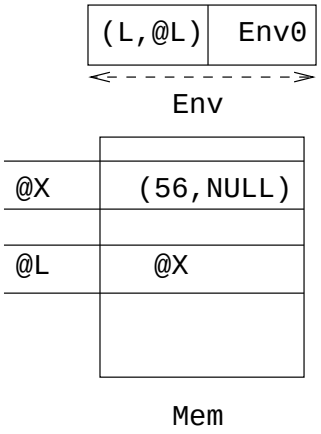
11.2 Manipulation de liste en Ada

```
L:Integer_list:=NULL;
L:=new Cellule;
-- le champ suivant est initialisé à NULL par défaut
L.valeur:=56;
```

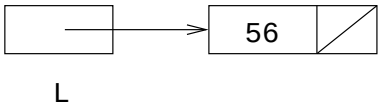
ou

```
L:= new Cellule'(56,NULL);
```

Etat après construction de la liste L

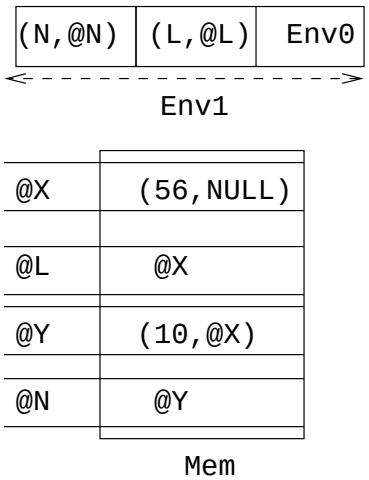


Représentation graphique de la liste L

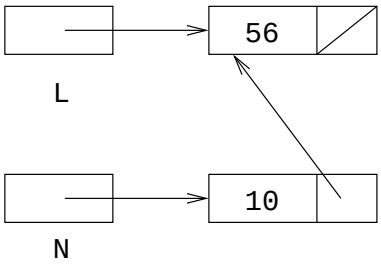



```
N:Integer_list :=new Cellule'(10,L);
```

Etat après construction des listes L et N

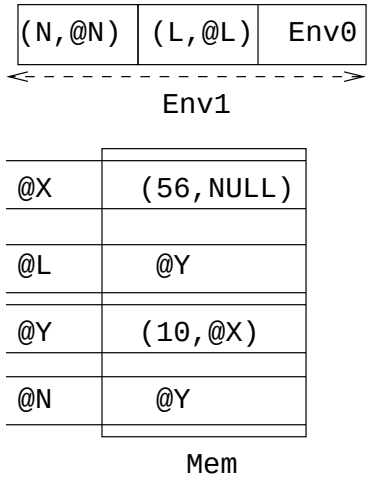


Représentation graphique des listes L et N

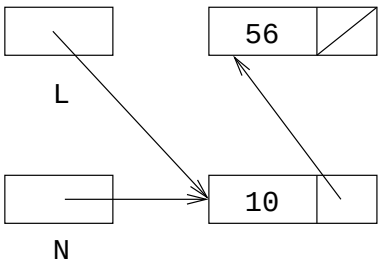



```
L:=N;
-- copie des pointeurs et non des objets
```

Etat après l'affectation : L:=N;



Représentation graphique des listes L et N



11.3 Pointeurs et objets pointés

11.3.1 Copie d'objets

`L` désigne le pointeur et `L.all` l'objet pointé. Pour accéder un champ de l'objet pointé, on utilise naturellement l'opérateur d'accès : .

```
L.all.valeur:=N.all.valeur;
L.all.suivant:=N.all.suivant;
```

La simplification suivante est possible

```
L.valeur:=N.valeur;
L.suivant:=N.suivant;
```

On peut aussi copier un objet pointé dans un autre par une simple affectation

```
L.all:=N.all;
```

11.3.2 Comparaison d'objets

```
L=N
-- vaut TRUE si L et N réfèrent le même objet
L.all=N.all
-- vaut TRUE si les objets pointés ont même valeur
L=NULL
-- vaut TRUE si la liste L est vide
```

11.3.3 Exemple

On souhaite lancer des invitattions pour le prochain réveillon. Les seules informations pertinentes sont le nř de téléphone et le sexe de la personne.

Sans réellement créer une liste, on peut envisager les définitions de types suivantes

```
type Genre is (masculin,feminin);
type Invite is
  record
    tel:String(1..10);
    sexe:Genre;
  end record;
type Reveillon is access Invite;
```

ainsi que la déclaration de quelques variables :

```
X,Y:Reveillon;I:Invite;
```

Pour créer dynamiquement un nouvel invité :

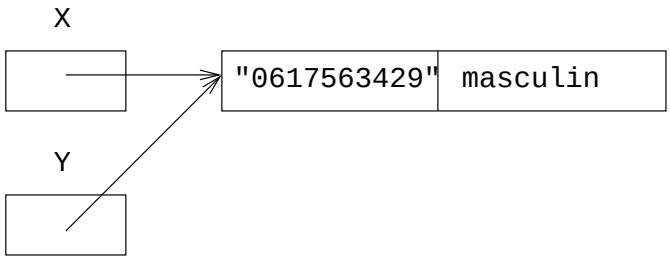
```
X:=new Invite'(tel=>"0617563429",sexe=>masculin);
```

Les affectations suivantes sont légales :

```
I:=X.all;
I.tel:=X.all.tel;
I.tel:=X.nom;
```

Après l'affectation suivante, `X` et `Y` pointent-ils sur le même invité ?

```
Y:=X;
```



la comparaison `Y.all=X.all` vaut donc `TRUE`.

Soit la nouvelle affectation :

```
Y.tel:="0123986246"
```

Est-ce que `X.tel=Y.tel` ?

Même question après la création d'un nouvel invité :

```
Y:=new Invite'("0657924175",feminin);
```

11.4 Exemple

On souhaite ajouter un nouveau complexe au début d'une liste de complexes

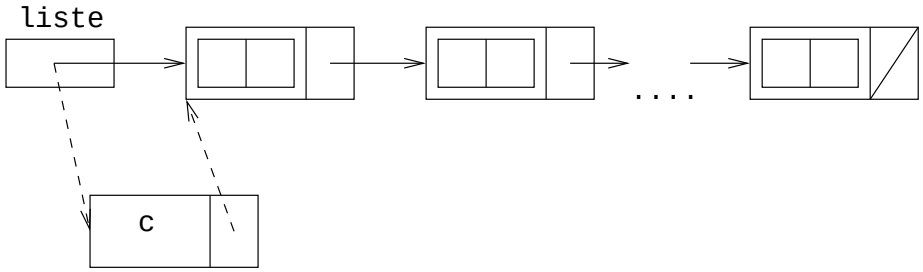
```
type Complexe is
  record
    re,im:Float;
  end record;

type Cellule;

type ListeComplexe is access Cellule;

type Cellule is
  record
    valeur:Complexe;
    suivant:ListeComplexe;
  end record;
procedure ajouter(c:in Complexe;liste:in out ListeComplexe) is
begin
  liste:=new Complexe'(c,liste);
end ajouter;
```

Ajout d'un nouveau complexe dans la liste



11.5 Exemple

Reprenons l'exemple dans lequel il s'agit de dresser la liste des mots d'un texte accompagnés de leur nombre d'occurence.

Pour résoudre le problème nous introduisons les fonctions :

- `vide` qui prend une liste en paramètre et retourne `TRUE` si la liste ne possède aucun élément, `FALSE` sinon
- `enTete` qui, à partir d'un mot et d'une liste retourne `TRUE` si le mot appartient à la première `Cellule` de la liste, `FALSE` sinon

et les procédures :

- `ajouter` qui, étant donné un mot, forme une nouvelle `Cellule` et l'ajoute en tête de la liste accompagné de la valeur 1 pour le nombre d'occurrence ou bien, si le mot est déjà présent dans la liste, incrémente son nombre d'occurrences.
- `put` qui est une procédure d'affichage sur l'écran du contenu de la liste

On remarquera la définition récursive de la procédure `ajouter`. En effet le problème d'ajouter un mot à une liste non vide dont la premiere `Cellule` contient un mot différent revient à celui d'ajouter ce même mot à la liste privée de sa première `Cellule`.

On notera aussi la nécessité d'introduire un bloc interne afin de déclarer la variable `r` de type `Donnee` dont on ne connait pas la taille à la compilation mais seulement au cours de l'exécution du programme, c'est à dire au fur et à mesure de la lecture du fichier `texte`.

```
with Ada.Text_io;use Ada.Text_io;
with Ada.Integer_Text_io;use Ada.Integer_Text_io;
procedure mots is
  type Donnee(lg:Positive) is
    record
      mot:String(1..lg);
      occ:Natural:=0;
    end record;

  type Cellule;
  type Liste is access Cellule;
  type Cellule(lg:Positive) is
    record
      valeur:Donnee(lg);
```

```
      suivant:Liste;
    end record;

  function vide(l:Liste) return Boolean is
  begin
    return l=NULL;
  end vide;

  function enTete(m:String;l:Liste) return Boolean is
  begin
    return l.valeur.mot=m;
  end enTete;

  procedure ajouter(l:in out Liste;d:in Donnee) is
    n:Liste:=new Cellule(d.lg);
  begin
    if not vide(l)
    then
      if enTete(d.mot,l)
      then
        l.valeur.occ:=l.valeur.occ+1;
      else
        ajouter(l.suivant,d);
      end if;
    else
      n.valeur.mot:=d.mot;
      n.valeur.occ:=1;
      n.suivant:=NULL;
      l:=n;
    end if;
  end ajouter;

  procedure put(l:in Liste) is
    x:Liste:=l;
  begin
    put_line(" ------ affichage de la liste -------");
    while not vide(x) loop
      put(x.valeur.mot);put(", ");
      put(x.valeur.occ,WIDTH=>2);
      new_line;
      x:=x.suivant;
    end loop;
  end put;
  f:FILE_TYPE;
  motMax:String(1..80);
  long:Positive;
  laListe:Liste:=NULL;

begin
  open(f,IN_FILE,"texte");
  while not END_OF_FILE(f) loop
     get_line(f,motMax,long);
     declare
       r:Donnee:=(long,motMax(1..long),0);
     begin
       ajouter(laListe,r);
     end;
   end loop;
```

```
    close(f);
    -- affichage de la liste
    put(laListe);
end mots;
```

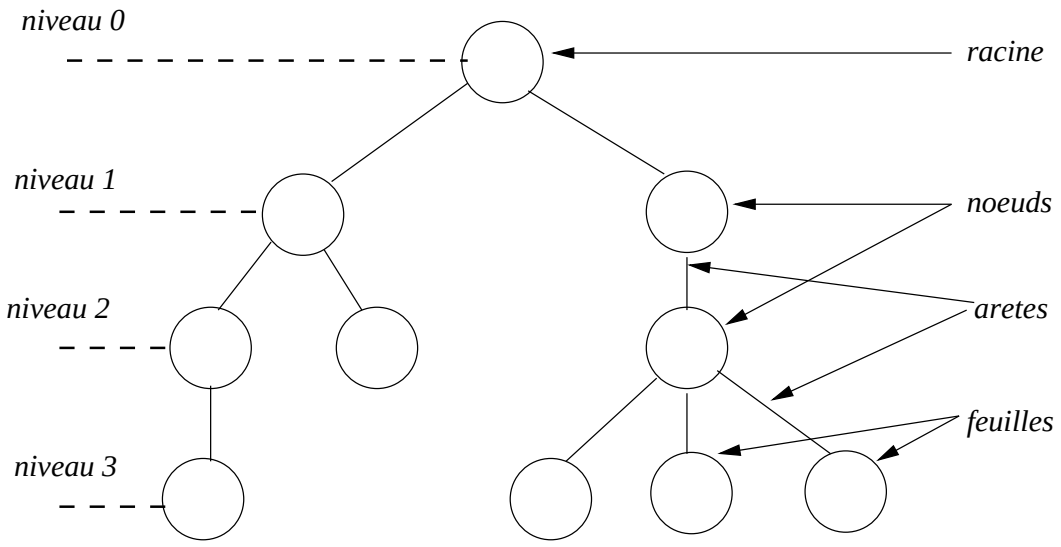
11.6 Notion d'arbre binaire

Les arbres binaires sont des structures de données naturellement adaptées à une grande variété d'algorithmes :

- recherche du meilleur coup à jouer aux échecs, au go, etc ...
- recherche d'une donnée associée à une clé

11.6.1 Représentation graphique

Un arbre binaire est représenté par un diagramme constitué de noeuds et d'arêtes



11.6.2 Terminologie

Les *noeuds* sont en relation père-fils.

La *racine* est un noeud sans père.

Une *feuille* est un noeud sans fils.

Les noeuds sont organisés en *niveaux*.

La racine est au niveau 0.

Un *chemin* vers un noeud N est le parcours suivi à partir de la racine pour atteindre le noeud N.

La longueur d'un chemin est le nombre d'*arêtes* traversées.

11.6.3 Définition

Un arbre binaire est :

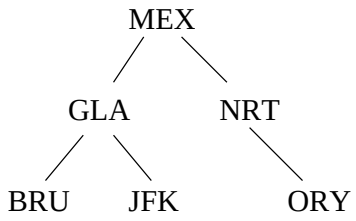
- soit une arbre vide
- soit un noeud qui possède un sous-arbre gauche et un sous-arbre-droite eux arbres binaires.

Là encore la structure de données arbre binaire est définie récursivement.

11.6.4 Exemple d'arbreBinaire

Imaginons un arbre binaire dont les données contenues associent les sigles des aéroports internationaux à des données descriptives de ceux-ci. Ces sigles sont constitués de 3 caractères (`ORY, GLA, JFK, MEX,NRT,...`).

Il est interessant de les classer dans un arbre binaire selon l'ordre lexicographique et non dans une liste. En effet, aussi bien pour la recherche d'un sigle que pour l'ajout d'un nouveau sigle dans l'arbre, la rapidité d'exécution de l'algorithme en est améliorée. Intuitivement, on peut admettre qu'en moyenne la recherche sera plus courte par rapport à un classement séquentiel dans une liste. En effet, la longueur de chacun des chemins est inférieure à la longueur de la séquence .



11.6.5 Typage de l'arbre binaire

```
type Noeud;
type Arbre is access Noeud;
type Element is
  record
    sigle:String(1..3);
    info:Donnee;-- on suppose défini le type Donnee
  end record;
type Noeud is
  record
    valeur:Element;
    filsGauche:Arbre;
    filsDroite:Arbre;
  end record;
```

11.6.6 Recherche d'une `Donnee` dans un arbreBinaire

Soient `a` l'arbre binaire, `ag` et `ad` respectivement ses fils gauche et droite, `<s,d>` un `Element` de `a=<<s,d>,ag,ad>`. Les éléments étant classés selon l'ordre lexicographique (opération de comparaison des sigles : `>`), on peut décrire l'algorithme de recherche :

```
rechercher(s':Sigle,<<s,d>,ag,ad>:Arbre)=
  si vide(a) alors exception
  sinon
    si s=s' alors d
    sinon si s>s'
          alors rechercher(s',ag)
          sinon rechercher(s',ad)
          finsi
   finsi
  finsi
```

11.6.7 Insérer un `Element` à un arbre binaire

En reprenant les notations du paragraphe précédent, l'algorithme s'écrit :

```
inserer(s':Sigle,d':Donnee,<<s,d>,ag,ad>:Arbre)=
  si vide(a) alors <<s',d'>,NULL,NULL>
  sinon
    si s/=s'
    alors si s>s'
          alors <<s,d>,inserer(s',d',ag),ad>
          sinon <<s,d>,ag,inserer(s',d',ad)>
          finsi
   finsi
  finsi
```

Pour implanter ces deux algorithmes, il faut disposer d'opérations primitives sur les arbres binaires. Cet aspect sera développé dans le chapitre suivant.

Chapter 12

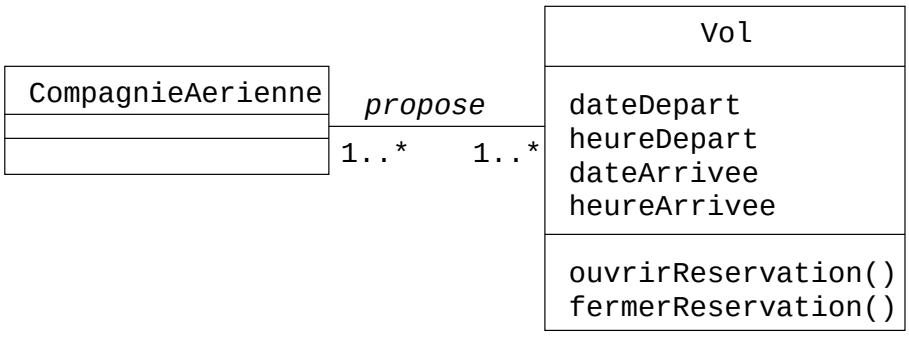
Conception modulaire : les paquetages

12.1 Notion de module

Lorsque le problème devient complexe, le travail en équipe s'impose.

La spécification consiste d'abord en un découpage en entités distinctes favorisant ainsi le développement séparé de chacune d'entre elles.

La spécification, la construction, la documentation de logiciel complexe se fait de nos jours à l'aide du langage UML (Unified Modeling Language) devenu un standard industriel



Ce langage simplifie le processus de conception logicielle mais son étude n'est pas couverte par ce cours.

Chaque programmeur peut ainsi développer de façon indépendante ses propres fonctions, procédures, construire ses propres objets.

Chaque programmeur crée ainsi son propre environnement fondée sur l'environnement initial, l'environnement importé et ses propres déclarations.

A chaque entité spécifiée (module) est associée un environnement.

12.2 Nommer des environnements

La notion de module (*paquetage*) permet de nommer un environnement et de manipuler comme un tout un environnement fermé.

12.3 Abstraire des données

Le concept de procédure permet au programmeur d'enrichir le langage avec de nouvelles instructions qu'il a lui-même définies.

Le concept de module (*paquetage*) permet au programmeur d'enrichir le langage avec de nouveaux types de données.

Un module est une unité indépendante qui rassemble les opérations de création et de manipulation des valeurs d'un type de données.

12.3.1 Abstraction de données

Un module est divisé en deux parties :

- La première définit *l'interface* ou *spécification* (vision utilisateur du type de données) et regroupe les déclarations des opérations possibles sur les valeurs du type.
- La seconde définit *l'implantation* du type de données (vision développeur du type); elle regroupe les déclarations des corps des fonctions et procédures qui manipulent les valeurs du type.

12.3.2 Encapsulation

C'est le mécanisme du langage qui réalise l'abstraction de données, c'est à dire l'association en une même entité des deux parties du module.

Elle permet à l'utilisateur d'avoir accès au *"quoi"* d'un type de données sans avoir à se préoccuper du *"comment"*.

12.4 Paquetage Ada

12.4.1 Déclaration d'une spécification (interface)

```
<déclaration_paquetage>::=
package <ident_module> is
  <liste_déclarations>
end <ident_module>;
```

Exemple

```
package Vecteur is
 type Vect_int is array (Integer range <>) of Integer ;
 function "+"(U,V:Vect_int) return Vect_int;
 function "*"(V:Vect_int;N:Integer) return Vect_int;
 procedure put(X:in Vect_int);
end Vecteur ;
```

12.4.2 Déclaration du corps (implantation)

```
<déclaration_corps_paquetage>::=
package body <nom_module> is
   <liste_déclarations>
begin
   <suite_instructions>
exception
   <traitement_exceptions>
end <nom_module>;
```

Exemple

```
package body vecteur is
 function "+"(U,V:Vect_int) return Vect_int is
   W:Vect_int (V'range);
 begin
   for I in U'range loop
     W(I):=U(I)+V(I);
   end loop;
   return W;
 end "+";
end vecteur ;
```

12.4.3 Sémantique

L'évaluation de la spécification d'un module a pour effet de lier le nom du module à un environnement.

Cet environnement est construit à partir des déclarations contenues dans la spécification du module.

Soit `Env`, l'environnement courant, il s'agit de créer la liaison `(ident, valeur d'environnement)` où `ident` est le nom du module et de l'ajouter à `Env`.

Evaluation de la spécification d'un module

1. Ajout de la liaison `(ident, ??)`,
2. Evaluation de la valeur d'environnement : évaluation des déclarations contenues dans la spécification et création des liaisons correspondantes,
3. Modification de la valeur d'environnement par ajout des liaisons créees à l'étape précédente.

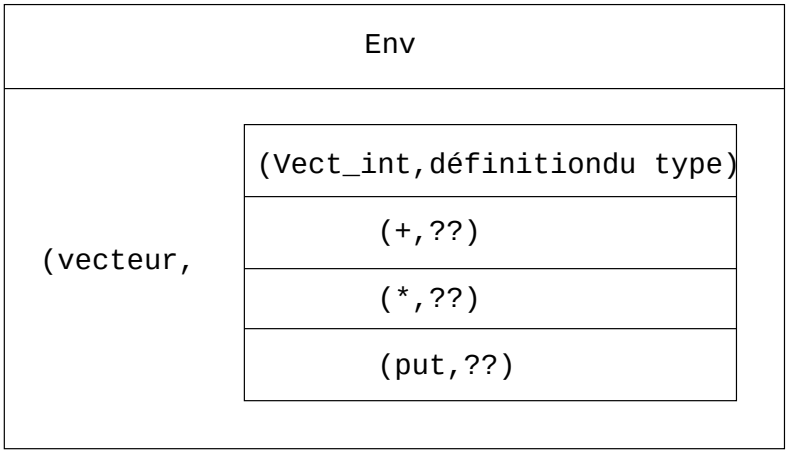
Evaluation du corps d'un module

1. Détermination des fermetures des fonctions et procédures déclarées dans la spécification
2. Modification des liaisons correspondantes de la valeur d'environnement.

12.4.4 Exemple

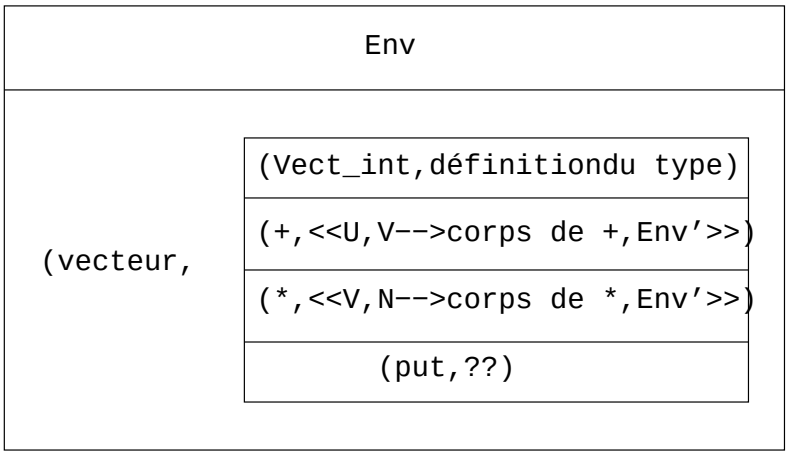
Evaluation de la spécification du paquetage `vecteur` dans `Env`.

```
package vecteur is
 type Vect_int is array (Integer range <>) of Integer;
 function "+"(U,V:Vect_int) return Vect_int;
 function "*"(V:Vect_int;N:Integer) return Vect_int;
 procedure put(X:Vect_int);
end vecteur ;
```



Evaluation du corps du paquetage `vecteur` dans `Env'`

```
with Ada.Text_io;
with Ada.Integer_Text_io;
package body vecteur is
 use Ada.Text_io;
 use Ada.Integer_Text_io;
 function "+"(U,V:Vect_int) return Vect_int is
   W:Vect_int(V'range);
 begin
   for I in U'range loop
     W(I):=U(I)+V(I);
   end loop;
   return W;
 end "+";
 function "*"(V:Vect_int;N:Integer) return Vect_int is
   W:Vect_int(V'range);
 begin
   for I in V'range loop
     W(I):=N*V(I);
   end loop;
   return W;
 end "*";
 procedure put(X:Vect_int)is
 begin
   for I in X'range loop
     put(X(I),width=>4);
 end put;
end vecteur ;
```



12.4.5 Opérations

Pour rendre visible un objet déclaré dans un module. On utilise la construction syntaxique :

<op_acces_objet>::=<ident_module>.<identificateur_objet>

Evaluation

1. Dans l'environnement courant, recherche de la liaison `(identificateur du module, valeur d'environnement)`
2. Soit `Env'` la valeur d'environnement trouvée, recherche dans `Env'` de la liaison `(identificateur d'objet, valeur de l'objet)`

Exemple d'accès à un objet d'un module Accès à un type

```
vect:Vecteur.Vect_int(1..4):=(3,8,6,0);
```

Accès à une opération

```
vect:Vecteur."+"(vect,vect);
```

12.5 Types abstraits de données et paquetages

Une manière répandue, dans l'esprit de la conception par objets, est de définir des *Types Abstraits de Données* (TAD).

Par exemple, si l'analyse de notre problème fait apparaitre que nous devons manipuler un compteur, alors il nous faut définir un ensemble de valeurs et un ensemble d'opérations sur ces valeurs, représentant les compteurs.

```
compteur
  ensemble de valeurs:
     N(ensemble des entiers naturels)
  ensemble des opérations :
     initialiser : vide -> N
     incrementer : N -> N
     decrementer : N -> N
     zero : N -> boolean
```

Pour être utilisable dans un programme, ce TAD doit être implanté sous la forme d'un paquetage Ada.

12.6 Interface du paquetage compteur

```
package compteur is
 type Total is limited private;
 procedure initialiser(X:out Total);
 procedure incrementer(X:in out Total);
 procedure decrementer(X:in out Total);
 function zero(X:Total) return Boolean;
private
 type Total is new Integer;
end compteur;
```

Ce paquetage constitue une nouvelle boite à outils

Son utilisateur pourra créer des objets du type `Total`. Pour cela, il n'a pas besoin de savoir comment le TAD a été implanté.

Seule l'interface lui est accessible.

Le type `Total` est déclaré *limited private*. Cela signifie que l'affectation *(*`:=`*)* est interdite sur les objets du type `Total`. Si l'on veut modifier la valeur d'un `compteur`, il est obligatoire d'utiliser les procédures `incrementer` et/ou `decrementer`.

12.7 Implantation du paquetage compteur

```
package body compteur is
 procedure initialiser(X:out Total)
 begin
   X:=0;
 end initialiser;
 procedure incrementer(X:in out Total)
 begin
   X:=X+1;
 end incrementer;
 procedure decrementer(X:in out Total)
 begin
   X:=X-1;
 end decrementer;
 function zero(X:Total) return Boolean
 begin
   return X=0;
 end zero;
end compteur;
```

12.7.1 Solution 1

([?] Fayard et Rousseau)

Ecrire un programme qui, étant donnée une suite de températures, cherche si il y a un nombre égal de températures positives et négatives.

1ère solution : on utilise le paquetage `compteur` et on définit un type `Temperature`.

```
with Ada.Text_io, Compteur;
procedure SOLUTION1 is
  use Ada.Text_io;
-- partie declarations
  TEMP_MAX:constant Float:=+60.0;
  TEMP_MIN:constant Float:=-60.0;
  type Temparature is
            Float range TEMP_MIN..TEMP_MAX;
  type Reponse is (oui,non);
  package temperature_io is
            new Float_io(Temperature);
  package reponse_io is
            new enumeration_io(Reponse);
  temp:Temperature;
  posNeg:compteur.Total;
  encore:Reponse:=oui;
-- partie instructions
begin
  Compteur.initialiser(posNeg);
  put_line("Entrez une température ");
  while encore=oui loop
    temperature_io.get(temp);new_line;
    if temp>0.0 then
       compteur.incrementer(posNeg);
    else
        compteur.decrementer(posNeg);
    endif;
    put_line("Voulez vous entrer une nouvelle temperature (oui/non)?");
    reponse_io.get(encore);
  end loop;
  if Compteur.zero(posNeg) then
    put_line("Il y a equilibre entre les temperatures positives et negatives");
  else
   put_line("Il y a desequilibre entre les temperatures positives et negatives");
  endif;
end SOLUTION1;
```

Le programme n'est pas indépendant de la manière dont sont implantées les opérations sur le type `Temperature`.

Les opérations sur les températures sont dissociées du type `Temperature`. C'est le signe `d'une` mauvaise architecture du programme, rendu plus lourd et moins lisible.

12.8 Solution 2

On pourrait rendre le programme plus lisible en définissant `Temperature` comme un type de données abstrait et en l'implantant dans un paquetage `temperatures`.

Définition du TAD

```
Temperature
   ensemble de valeurs : [-60.0..+60.0]
   ensemble des opérations :
      get : vide -> Temperature
      put : Temperature -> vide
      supZero : Temperature -> Boolean
```

Implantation du TAD, interface du paquetage `temperatures`

```
package temperatures is
 type Temperature is private;
 procedure get(X:out Temperature);
 procedure put(X:in Temperature);
 function supZero(X:Temperature) return Boolean;
private
 TEMP_MAX:constant Float:=+60.0;
 TEMP_MIN:constant Float:=-60.0;
 type TEMPERATURE is Float range TEMP_MIN..TEMP_MAX;
end temperatures;
```

12.9 Corps du paquetage temperatures

```
with Ada.Text_io;use Ada.Text_io;
package body temperatures is
  package temperature_io is new Float_io(Temperature);
  procedure get(X:out Temperature)is
  begin
    put_line("Entrez une temperature ");
    temperature_io.get(X);new_line;
  end get;

  procedure put(X:in Temperature)is
  begin
   temperature_io.put(X);
  end put;

  function supZero(X:Temperature) return Boolean is
  begin
   if X>0.0 then return true;
   else
     return false;
   endif;
  end supZero;
end temperatures;
```

12.10 Solution 2, le programme

```
with Ada.Text_io,compteur,temperatures;
procedure SOLUTION2 is
  use Ada.Text_io,compteur,temperatures;
  type Reponse is (oui,non);
  package reponse_io is new enumeration_io(Reponse);
  temp:Temperature;
  posNeg:Total;
  encore:Reponse:=oui;
```

```
begin
  initialiser(posNeg);
  get(temp);
  while encore loop
    if supZero(temp) then
        incrementer(posNeg);
    else
        decrementer(posNeg);
    endif;
    put_line("Voulez vous entrer une nouvelle temperature (oui/non)?");
    reponse_io.get(encore);
    get(temp);
  end loop;
  if zero(posNeg) then
    put_line("Il y a equilibre entre les temperatures positives et negatives");
  else
    put_line("Il y a desequilibre entre les temperatures positives et negatives");
  endif;
end SOLUTION2;
```

12.11 Maintenir un paquetage

12.11.1 Cacher l'implantation : partie privée, partie publique

Un module possède deux parties : une partie privée et une partie publique.

```
<déclaration_paquetage>::=
package <nom_module> is
  <liste_déclarations>
-- partie publique
private
  <liste_déclarations>
-- partie privée
end <nom_module>;
```

- La partie publique constitue la partie visible du module et fournit l'information disponible à l'extérieur.
- La partie privée contient les déclarations d'objets privés du module invisibles et inutilisables par les autres modules.

12.11.2 Types privés

Intérêt : cacher l'implantation d'un type aux utilisateurs extérieurs du module où il est déclaré.

On peut ainsi modifier l'implantation d'un type sans avoir à modifier les programmes qui utilisent ce type.

Un type dont le nom seul est introduit dans la partie publique d'un module (spécification) est un type privé. Il est privé bien que sa définition soit vue, mais étant défini dans le paragraphe `private`, il est inutilisable dans le programme utilisateur (client du module).

12.11.3 Opérations sur les types privés (déclarés en `private`)

<type_limite_prive>::=type <ident_type> is private;

Les seules opérations disponibles pour un utilisateur extérieur au module sont :

1. les opérations sur ce type déclarées dans la partie publique du module
2. l'affectation *(:=)*, l'égalité *(=)* et l'inégalité *(/=)*

12.11.4 Opérations sur les types privés (`limited private`)

<type_limite_prive>::=type <ident_type> is limited private;

Les opérations suivantes sont indisponibles pour un utilisateur extérieur au module

1. l'affectation *(:=)*, l'égalité *(=)* et l'inégalité *(/=)*

12.12 Le paquetage listeEntiers

12.12.1 Spécification

```
package ListeEntiers is
  ListeVide :exception;
  type Liste is private;
  function cons(elt:Integer;L:Liste) return Liste;
  function vide return Liste;
  function estVide(L:Liste) return Boolean;
  function tete(L:Liste) return Integer;
  function queue(L:Liste) return Liste;
private
  type Cellule;
  type ListeEntiers is access Cellule;
  type Cellule is
     record
       valeur:Integer;
       suivant:ListeEntiers;
     end record;
end ListeEntiers;
```

12.12.2 Implantation

```
package body ListeEntiers is
  -- construire une nouvelle liste dont elt
  -- appartient à la première Cellule suivie de la liste l
  function cons(elt:Integer;l:Liste) return Liste is
  begin
    return new Cellule'(elt,l);
  end cons;
  -- construire une liste vide
  function vide return Liste is
  begin
    retrun NULL;
  end vide;
```

```
  -- retourne TRUe si la liste est vide, FALSE sinon
  function estVide(l:Liste) return Boolean is
  begin
    return l=NULL;
  end estVide;
  -- retourne l'entier contenu dans la première Cellule
  function tete(l:Liste) return Integer is
  begin
    if l=NULL
    then raise listeVide;
    else return l.valeur;
    end if;
  end tete;
  -- retourne la liste privée de la première Cellule
  function queue(l:Liste) return Liste is
  begin
    if l=NULL
    then raise listeVide;
    else return l.suivant;
    end if;
  end queue;
end ListeEntiers;
```

12.13 Le paquetage arbreBinaire

On peut maintenant décrire sous la forme d'un paquetage l'ensemble des opérations primitives sur les arbres binaires du chapitre précédent (association entre un sigle d'aéroport et ses données descriptives).

12.13.1 Paquetage arbreBinaire : spécification

```
package arbreBinaire is
  type Arbre is private;
  arbreVide:exception;
  type Element is
    record
      sigle:String(1..3);
      info:Donnee; -- on suppose ce type défini
    end record;
  function feuille(e:Element) return Arbre;
  function arbreVide return Arbre;
  function racine(a:Arbre) return Element ;
  function sousArbreGauche(a:Arbre) return Arbre;
  function sousArbreDroite(a:Arbre) return Arbre;
  function estFeuille(a:Arbre) return Boolean;
  function estVide(a:Arbre) return Boolean;
  function consArbre(e:Element;a1,a2:Arbre) return Arbre;
  function ajouterAGauche(e:Element;a:Arbre) return Arbre;
  function ajouterADroite(e:Element;a:Arbre) return Arbre;
private
  type Noeud;
  type Arbre is access noeud;
  type Noeud is
    record
      valeur:Element;
      filsGauche:Arbre;
```

```
      filsDroite:Erbre;
    end record;
end arbreBinaire;
```

12.13.2 Paquetage arbreBinaire : implantation

```
package body arbreBinaire is
  -- construit un arbre réduit à une feuille
  function feuille(e:Element) return Arbre is
  begin
    return consArbre(e,NULL,NULL);
  end feuille;
  -- construit un arbre vide
  function arbreVide return Arbre is
  begin
    return NULL;
  end arbreVide;
  -- retourne l'Element situé à la racine
  function racine(a:Arbre) return Element is
  begin
    return a.valeur;
  end racine;
  -- retourne le fils gauche de l'arbre a
  function sousArbreGauche(a:Arbre) return Arbre is
  begin
    if estVide(a)
    then raise arbreVide;
     else return a.filsGauche;
    end if;
  end sousArbreGauche;
  -- retourne le fils droite de l'arbre a
  function sousArbreDroite(a:Arbre) return Arbre is
  begin
    if estVide(a)
    then raise arbreVide;
    else return a.filsDroite;
    end if;
  end sousArbreDroite;
  -- retourne TRUE si l'arbre est réduit à une feuille
  function estFeuille(a:Arbre) return Boolean is
  begin
    return a.filsGauche=NULL and filsDroite=NULL;
  end estFeuille;
  -- retourne TRUE si l'arbre a est vide
  function estVide(a:Arbre) return Boolean is
  begin
    return a=NULL;
  end estVide;
  -- retourne l'arbre dont la racine contient l'Element e et dont
   -- a1 et a2 sont les fils gauche et droite
  function consArbre(e:Element;a1,a2:Arbre) return Arbre is
  begin
    return new Noeud'(e,a1,a2);
  end consArbre;
  -- fonction récursive qui parcourt l'arbre en profondeur vers la
  -- droite jusqu'à trouver le fils le plus à droite ayant
  -- une valeur NULL
```

```
  function ajouterADroite(e:Element;a:Arbre) return Arbre
  is
  begin
    if estVide(a) then return feuille(e);
    else return
         consArbre(racine(a),sousArbreGauche(a),
                   ajouterADroite(e,sousArbreDroite(a)));
   end if;
  end ajouterADroite;
  -- fonction récursive qui parcourt l'arbre en profondeur vers la
  -- gauche jusqu'à trouver le fils le plus à droite ayant
  -- une valeur NULL
  function ajouterAGauche(e:Element;a:Arbre) return Arbre
  is
  begin
    if estVide(a) then return feuille(e);
    else return
         consArbre(racine(a),
              ajouterAGauche(e,sousArbreGauche(a)),
              sousArbreDroite(a));
   end if;
  end ajouterAGauche;
end arbreBinaire;
```

12.13.3 Utilisation du paquetage

Nous pouvons maintenant coder en Ada les algorithmes de recherche et d'insertion en arbre binaire spécifiés dans le chapitre précédent.

Algorithme de recherche

```
rechercher(s':Sigle,<<s,d>,ag,ad>:Arbre)=
  si vide(a) alors exception
  sinon
    si s=s' alors d
    sinon si s>s'
          alors rechercher(s',ag)
          sinon rechercher(s',ad)
          finsi
   finsi
  finsi
```

Codage Ada de la fonction `rechercher` On suppose définis et appartenant à l'environnement de déclaration de la fonction les type `Cle` (`String(1..3)`) et `Donnee`.

On suppose aussi que le paquetage `arbreBinaire` est importé :

```
with arbreBinaire; use arbreBinaire;
function rechercher(s:Cle;a:Arbre) return Donnee is
begin
  if estVide(a) then raise arbreVide;
  elsif s=racine(a).sigle then return racine(a).info;
     elsif racine(a).sigle>s
        then return rechercher(s,a.filsGauche);
        else return rechercher(s,a.filsDroite);
```

```
  end if;
end rechercher;
```

Algorithme d'insertion

```
inserer(s':Sigle,d':Donnee,<<s,d>,ag,ad>:Arbre)=
  si vide(a) alors <<s',d'>,NULL,NULL>
  sinon
    si s/=s'
    alors si s>s'
          alors <<s,d>,inserer(s',d',ag),ad>
          sinon <<s,d>,ag,inserer(s',d',ad)>
          finsi
   finsi
  finsi
```

Codage Ada de la fonction `inserer`

```
function inserer(s:Cle;d:Donnee;a:Arbre) return Arbre is
  e:Element:=(s,d);
begin
  if estVide(a) then return feuille(e);
  elsif s/=racine(a).sigle
     then
        if racine(a).sigle>s
        then return  ajouterAGauche(e,a);
        else return  ajouterADroite(e,a);
  end if;
end inserer;
```

Remarques Les fonctions `rechercher` et `inserer` n'appartenant pas au paquetage `arbreBinaire`, elles n'ont pas accès à la partie privée de son interface. La structure sous la forme d'un *record* du type `Element` leur est donc inconnue. Pour accéder à ses champs, elles ne peuvent qu'utiliser les fonctions publiques du paquetage. Ainsi pour accéder au champ `valeur`, l'utilisation de la fonction `racine` est-elle necéssaire. De même l'accès aux champs `filsGauche` et `filsDroite` est effectué à travers les fonctions `sousArbreGauche` et `sousArbreDroite`.

Chapter 13

Construction de logiciel

13.1 Modularité

13.1.1 Définition

Une unité de programmation, pour être modulaire, doit être *indépendante* et *autonome*.

1. *Autonome* : elle doit posséder une forte cohérence interne. Pour cela elle doit être le reflet fidèle d'un objet de l'espace du problème.
2. *Indépendante* : elle est faiblement couplée avec les autres unités. Elle entretient un ensemble de relations minimum et explicitement identifié avec les autres unités constituant le programme dont elle fait partie.

13.1.2 Identification des modules

Il s'agit de procéder à un découpage du problème de manière à faire apparaitre les modules constituant son architecture.

Deux méthodes sont disponibles :

- l'analyse descendante
- l'analyse ascendante

13.1.3 Nature des modules

Les modules (paquetages Ada) remplissent des fonctions de nature différente :

- types abstraits de données
- machine abstraite
- regroupement des ressources

13.1.4 Outils de contrôle sur les objets d'un module

- Visibilité
- Intégrité des données
- Gestion des noms : surcharge

13.1.5 Réutilisabilité

Construction de composant par abstraction

- Abstraction de données (encapsulation, séparation interface/implantation)
- Abstraction : généricité

13.2 Analyse descendante et compilation séparée

13.2.1 Décomposition descendante d'un problème

L'analyse descendante procède par la décomposition d'un problème en sous-problèmes et ainsi de suite jusqu'à un sous-problème atomique (trivial).

Elle n'est valide que si à chaque niveau de raffinement la solution des sous-problèmes identifiés est supposée connue.

13.2.2 Traduction Ada

Il s'appuie sur la notion d'unité (paquetage, sous-programme).

Chaque unité contient la spécification des sous-unités du niveau inférieur.

Afin que chaque unité puisse être compilée séparément, le corps des sous-unités internes est remplacé par le mot clé `separate`.

Représentation Ada de l'architecture

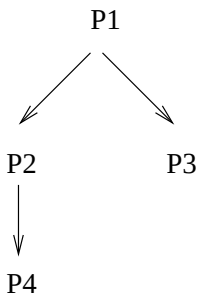
```
procedure P1 is
  ..
  procedure P2(...) is separate;
  procedure P3(...) is separate;
  ..
end P1;

separate (P1)
procedure P2 is
  ..
  procedure P4(...) is separate;
  ..
end P2;

separate (P1)
procedure P3 is
  ..
```

```
end P3;

separate (P1.P2)
procedure P4 is
  ..
end P4;
```



Remarque: le mot clé `separate` s'applique aux unités suivantes

- Sous-programme

```
procedure <nom_proc>(<liste _paramètres>) is separate;
```

- Paquetage

```
package body <nom_module> is separate
```

13.3 Paquetage et compilation séparée

```
with Ada.Text_io;use Ada.Text_io;
with Ada.Integer_Text_io;
use Ada.Integer_Text_io;
procedure client is
 -- interface du paquetage rectanglePack
 package rectanglePack is
  type Rectangle is private;
  procedure get(r:out Rectangle);
  procedure put(r:in Rectangle);
  function perimetre(r:Rectangle) return Positive;
  function aire(r:Rectangle) return Positive;
 private
  type Rectangle is
   record
    long:Positive;
    larg:Positive;
   end record;
 end rectanglePack;
 -- corps du paquetage rectanglePack
 package body rectanglePack is
  procedure getInt(n:out Positive)is
  begin
   loop
```

```
    begin
     get(n);exit;
    exception
     when DATA_ERROR=>skip_line;
                      put("Erreur!Recommencez");
    end;
   end loop;
  end getInt;
  procedure get(r:out Rectangle) is
  begin
   put_line("Saisie d'un rectangle");
   put("Entrez la largeur: ");
   getInt(r.larg);
   put("Entrez la longueur: ");
   getInt(r.long);
  end get;
  procedure put(r:in Rectangle)is
  begin
   put_line(Affichage d'un rectangle");
   put("Largeur=");put(r.larg);new_line;
   put("Longueur=");put(r.long);new_line;
  end put;
  function perimetre(r:Rectangle) return Positive is
  begin
   return 2*(r.long+r.larg);
  end perimetre;
  function aire(r:Rectangle) return Positive is
  begin
   return r.long*r.larg;
  end aire;
end rectanglePack;

-- Corps de la procédure client
begin
 declare
  rect:rectanglePack.Rectangle;
 begin
  rectanglePack.get(rect);
  rectanglePack.put(rect);
  put(rectanglePack.perimetre(rect));
  put(rectanglePack.aire(rect));
 end;
end client;
```

La procédure *client* n'a pas accès à l'implantation (au *corps*) du module `rectanglePack`, elle a seulement connaissance de son *interface* (ensemble des services mis à disposition).

Le module pourrait être développé séparément du client .

Le paquetage `rectanglePack` devient extérieur au programme client.

```
-- interface du paquetage rectanglePack
package rectanglePack is
  type Rectangle is private;
  procedure get(r:out Rectangle);
  procedure put(r:in Rectangle);
  function perimetre(r:Rectangle) return Positive;
  function aire(r:Rectangle) return Positive;
```

```
 private
  type Rectangle is
   record
    long:Positive;
    larg:Positive;
   end record;
 end rectanglePack;
```

L'interface du module est importée par le client sans que ce dernier ait necéssité de connaitre l'implantation de ce module. Le client n'est pas dépendant des modifications apportées à la représentation du type `Rectangle` exporté par le module `rectanglePack`.

```
with rectanglePack;
procedure client is
  rect:rectanglePack.Rectangle;
begin
  rectanglePack.get(rect);
  rectanglePack.put(rect);
  put(rectanglePack.perimetre(rect));
  put(rectanglePack.aire(rect));
end client;
```

Le développement du module peut être réalisé de manière incrémentale en utilisant les possibilités de compilation séparée.

```
-- corps du paquetage implanté séparément
package body rectanglePack is
  procedure getInt(n:Positive) is separate;
  procedure get(r:out Rectangle) is separate;
  procedure put(r:in Rectangle)is separate;
  function perimetre(r:Rectangle) return Positive
                                        is separate;
  function aire(r:Rectangle) return Positive
                                        is separate;
end rectanglePack;

-- implantation séparée des procédures et fonctions
-- du paquetage rectanglePack
separate rectanglePack;
procedure getInt(n:out Positive)is
begin
  loop
   begin
    get(n);exit;
   exception
    when DATA_ERROR=>skip_line;
                     put("Erreur!Recommencez");
   end;
  end loop;
end getInt;
separate rectanglePack;
procedure get(r:out Rectangle) is
begin
  put_line("Saisie d'un rectangle");
  put("Entrez la largeur: ");
  getInt(r.larg);
```

```
  put("Entrez la longueur: ");
  getInt(r.long);
end get;
separate rectanglePack;
procedure put(r:in Rectangle)is
begin
  put_line(Affichage d'un rectangle");
  put("Largeur=");put(r.larg);new_line;
  put("Longueur=");put(r.long);new_line;
end put;
separate rectanglePack;
function perimetre(r:Rectangle) return Positive is
begin
  return 2*(r.long+r.larg);
end perimetre;
separate rectanglePack;
function aire(r:Rectangle) return Positive is
begin
  return r.long*r.larg;
end aire;
```

13.4 Analyse ascendante et compilation séparée

13.4.1 Construction ascendante d'une solution

Elle consiste à construire un programme à partir d'unités précompilées en bibliothèque.

13.4.2 Mécanisme Ada

Importation de l'unité avec la clause `with`

Remarque: l'environnement ainsi importé est transmis aux sous-unités (et éventuellement au corps de l'unité).

13.5 Contrôle sur la manipulation de données

13.5.1 Types privés

A un type (ensemble de valeurs) est associé un ensemble de sous-programmes (procédure ou fonction) opérant sur lui-même.

Aucun autre sous-programme ne doit pouvoir agir sur ce type et avoir ainsi la possibilité de porter atteinte à l'intégrité des objets (valeurs) de ce type.

Pour satisfaire cette spécification, la structure interne du type doit être cachée.

Ada propose le mécanisme de *type privé* qui permet la séparation entre la déclaration d'un type et sa représentation.

```
-- spécification du paquetage dates sans type privé
package dates is
  type Date is
```

```
    record
       leJour:Jour;
       leMois:Mois;
       lAn:An;
    end record;
  function quelJour(d:Date) return Jour;
  ...
end dates;
-- spécification du paquetage dates avec type privé
package dates is
  type Date is private;
  function quelJour(d:Date) return Jour;
  ...
private
  type Date is
    record
      leJour:Jour;
      leMois:Mois;
      lAn:An;
    end record;
end dates;
```

13.5.2 Types limités privés

Lorsqu'un type est déclaré *limité privé*, les opérateurs définis par défaut sur les objets de ce type ne sont plus applicables

```
:= , < , <= , > , >=
```

Il est néammoins toujours possible de les redéfinir dans le paquetage.

Exemple: les complexes, on redéfinira l'affectation *(*`:=`*)*

13.6 Nature d'une solution

13.6.1 Types abstraits

Lors de l'analyse d'un problème, on identifie les données manipulées. Dans une première étape, il s'agit de spécifier les opérations qui sont réalisées sur ces données ainsi que leurs propriétés en laissant de coté à la fois la représentation physique des dites données ainsi que l'implantation des opérations.

En définissant le *"quoi"* sans le *"comment"*, on définit un type de données de manière abstraite.

Un *Type de Données Abstrait* (TAD) est implanté en *Ada* par un paquetage qui exporte le *type concret* correspondant. Il devient, après importation, un composant réutilisable à l'image des types de données prédéfinis. Cette vue abstraite se positionne comme une interface claire (*"un contrat"*) entre l'utilisateur et le développeur chargé de respecter les propriétés des données à l'implantation.

13.6.2 Machine abstraite

- Le paquetage encapsule la donnée

- Pas de type exporté, seulement des opérateurs
- La donnée est déclarée dans le corps
- Le paquetage contient un objet et ses sous-programmes en modifient l'état.

```
-- paquetage livres ([?]G.Booch)
package livres is
 subtype Livre is String(1..20);
end livres;
-- paquetage bibliotheque
package bibliotheque is
 procedure ajouter(l:in Livre);
 procedure retirer(l:in Livre);
 procedure put;
end bibliotheque;
-- corps du paquetage bibliotheque
with livres;
with Ada.Text_io;use Ada.Text_io;
package body bibliotheque is
 biblio:array(1..1000) of Livre;
 indice:Integer range 1..biblio'last:=1;
 procedure ajouter(l:in Livre) is
 begin
  biblio(indice):=l;
  indice:=indice+1;
 end ajouter;
 procedure retirer(l:in Livre) is
 begin
  for i in 1..biblio'last loop
   if biblio(i)=l
   then
    indice:indice-1;
    for j in i..biblio'last-1 loop
     biblio(j):=biblio(j+1);
    end loop;
   end if;
  end loop;
 end retirer;
 procedure put is
 begin
  for i in 1..biblio'last loop
   put_line(biblio(i));
  end loop;
 end put;
end bibliotheque;

-- programme utilisateur
with bibliotheque;
with livres;use livres;
procedure testbib is
 y:Livre:="oui-oui en ballon";
begin
 bibliotheque.ajouter(y);
 bibliotheque.put;
end testbib;
```

13.6.3 Regroupement de ressources

Le paquetage regroupe des constantes, exceptions, sous-prgrammes d'intérêt général.

```
with Ada.Text_io;use Ada.Text_io;
package e_s is
  package positif_io is new integer_io(Positive);
  package naturel_io is new integer_io(Natural);
  package bool_io is new  enumeration_io(Boolean);
end e_s;
```

13.7 Généraliser un problème

13.7.1 Généricité

Ada permet la généralisation des unités (fonction, procédure, paquetage) en offrant la possibilité de les paramétrer avec des types ou des fonctions

Ada permet ainsi une certaine forme de définition et de manipulation des valeurs fonctionnelles et l' introduction du polymorphisme en déclarant explicitement les paramètres de type.

Rappel : un objet est de *type polymorphe* s'il contient au moins un paramètre de type.

13.7.2 Fonctions génériques

Exemple

```
generic
  type Elt is private;
  with function "*"(u,v:Elt) return Elt is <>;
function auCarre(x:Elt) is
begin
  return x*x;
end auCarre;
function carreEntier is new auCarre(Integer);
function carreReel is new auCarre(Float);
```

13.7.3 Les paramètres génériques

Un paramètre générique peut être :

- Un type

```
type <identificateur> is private;
```

Dans ce cas, le type pourra être spécialisé par n'importe quel type.

Seules les opérations `=`, `\=` et `:=` sont prédéfinies sur le type.

```
type <identificateur> is (<>);
```

Dans ce cas, le type ne pourra être spécialisé que par un type discret.

Les attributs des type discrets sont alors utilisables sur les objets du type.

```
type <identificateur> is range <>;
```

Dans ce cas, le type ne pourra être spécialisé que par un type entier.

Toutes les opérations prédéfinies sur les entiers sont alors utilisables.

- Un paramètre usuel

```
<identificateur>:<type_param>:=<expression>;
```

- Une fonction

```
with function <id_f>
{(<id_p>:<id_t>;...;<id_p>:<id_t>)} return <id_t> is <>;
```

avec

```
<id_f>::=identificateur de fonction
<id_p>::=identificateur de paramètre
<id_t>::=identificateur de type
```

13.7.4 Déclaration de fonction générique

Syntaxe

```
<déclaration_fonction_générique>::=
generic
<paramètres_génériques>
function <id_f> {(<id_p>:<id_t>;...;<id_p>:<id_t>)}
                                                return <id_t>;
```

Sémantique La déclaration de la fonction générique dans l'environnement `Env` de l'état courant s'effectue en 2 étapes :

- Ajout de la liaison avec une valeur indéfinie dans `Env` :
  `(ident,??)`
- Détermination du type des paramètres formels génériques et du type de la fonction elle-même.
- Le type de la fonction générique est alors :
  `Tg1*Tg2*..*Tgn-->(Tp1*..*Tpj->T)`

La déclaration du corps de la fonction générique dans l'environnement `EnvDef` de l'état courant s'effectue en 2 étapes :

- Détermination de la fermeture, à partir du corps de la fonction générique, dans son environnement `EnvDef` :

```
F=ń(g1,g2,..,gn)->(p1,p2,..,pj)-->
                              corps de la fonction),EnvDefż
```

- Remplacement de la valeur indéfinie `??` par `F` dans `Env`.

Remarques

Le rôle de la déclaration de la fonction générique est :

1. d'ajouter la liaison dans l'environnement
2. de typer l'identificateur de la fonction générique

Le rôle de la déclaration du corps de la fonction générique est :

1. de déterminer la fermeture avec l'environnement du moment
2. de remplacer la valeur indéfinie par la fermeture

La déclaration de la fonction générique est obligatoire : la déclaration du corps ne peut en tenir lieu.

13.7.5 Exemple

Déclaration dans l'environnement `Env` :

```
generic
  type T is private;
function identite(X:T) return T;
```

- Ajout de la liaison `(identité,??)` dans `Env`

(identite, ??)	Env

- Détermination du type de la fonction

T ----->	(T-----> T)
type générique	type de `identite`

```
-- autres déclarations éventuelles
```

Soit `EnvDef` l'environnement courant au moment de la déclaration du corps de la fonction :

```
function identite(X:T) return T is
begin
    return X;
end identite;
```

- Calcul de la fermeture

```
F=<<T->(T->corps de identite),EnvDef>>
```

- Remplacement dans `EnvDef` de la valeur `??` par `F`

(identite,F)	Env

13.7.6 Instanciation

Pour utiliser une fonction générique, il faut la spécialiser; c'est à dire spécialiser chaque paramètre générique.

On précise ainsi sur quels types, fonctions ou paramètres usuels on veut appliquer la fonction générique.

Syntaxe

<instanciation>::=
function <id> is new <id_générique>[(<param génériques effectifs)];

Sémantique Soit la déclaration dans `Env` :

```
function id is new id_gen(Par_Gen_Eff);
```

- Ajout dune nouvelle liaison dans l'environnement courant `Env` avec une valeur indéfinie:

  `(id, ??)`

- Détermination de la fermeture de `id :(F_id)` et de son type à partir de la fermeture de `id_gen` :

  `F_id_gen= ńPar_Gen->(P->corps de id_gen),EnvDefż`

- Le code de `id` est celui de la fonction générique appliquée aux paramètres génériques effectifs

  `(P->corps de id_gen appliquée à Par_Gen_Eff)`

- L'environnement `E_id` de `id` est construit à partir de l'environnement de définition de la fonction générique :

  – par ajout des liaisons

  `(Par_Gen, Par_Gen_Eff)`

  on obtient

  `{(g1,e1),(g2,e2),..,(gn,en)<EnvDef}`

  – Modification physique de la liaison `(id_gen,F_id_gen)` par `(id_gen,F_id)`

  `E_id={(id_gen,F_id)(g1,e1)(g2,e2)..(gn,en)< Env_def}`

avec

`F_id=ń(P->corps de id_gen appliqué à Par_Gen_Eff),E_idż`

Le nouvel environnement `Env1={(id,F_id) < Env}`

Note : la présence de la liaison entre l'identificateur de la fonction générique et la fermeture de son instance permet la correction des appels récursifs.

13.7.7 Sémantique de l'instanciation : exemple

```
generic
        A:Integer:=1;
        B:Integer:=0;
function AF(X:Integer) return Integer;
function AF(X:Integer) return Integer is
begin
        return (A*X+B);
end AF;
```

Cette déclaration introduit dans `Env` la liaison :

```
(AF,F)
```

avec

```
F=<<(A->B)->X->Corps de AF, Env>>
```

Soit, dans `Env'`, la déclaration :

```
function INST_AF is new AF(A=>2);
```

La fermeture `F_INST_AF` de `INST_AF` est construite à partir de la fermeture `F` de la fonction générique `AF`

- Construction du corps de `INST_AF`:

```
X-->corps de AF appliqué aux params génériques effectifs
```

- Construction de son environnement de définition `EnvAF` à partir de l'environnement de définition de la fonction générique `AF`, par ajout

(INST_AF,F_INST_AF)	(A,2)	(B,0)	...	Env

d'où

```
F_INST_AF=<<X-->Corps de INST_AF,EnvAF>>
```

L'environnement courant `Env''` prend la forme :

```
Env''= {(INST_AF,F_INST_AF)...(AF,F)<Env}
```

13.7.8 Généralisation d'une fonction

A partir d'une fonction Ada, on peut obtenir une fonction plus générale par abstraction des constituants.

Exemple

```
function min(X,Y:Integer) return Integer is
begin
 if X<=Y
 then return X;
 else return Y;
 end if;
end min;
```

Remarques: La fonction `min` n'agit que pour le type `Integer`

La fonction `min` est basé sur l'opérateur `"<="` de type :

```
Integer*Integer-->Boolean
```

13.7.9 Généralisation de la fonction `min`

On peut définir `min` sur tout type entier au lieu de `Integer` seul :

```
generic
  type T is range <>;
function min (X,Y:T) return T;
```

On peut définir `min` sur tout type discret (entiers et énumérés) : l'affectation est correctement typée.

```
generic
 type T is (<>);
function min(X,Y:T) return T;
```

On peut aussi définir `min` sur tout type. Dans ce cas, il faut abstraire aussi `"<="`.

```
generic
 type T is private;
 with function "<="(X,Y:T) return Boolean is <>;
function min(X,Y:T) return T;
```

Dans tous les cas, le corps de la fonction `min` est :

```
function min(X,Y:T) return T is
begin
 if X<=Y
 then
  return X;
 else
  return Y;
 end if;
end min;
```

La fonction `min` sur le type `Integer` s'obtient maintenant par instanciation de la fonction générique `min`.

```
function minEntier is new min(Integer);
```

13.7.10 Correspondance de type

définition du type formel	signification	opérations possibles
private	tout type	= /= :=
(<>)	type discret	= /= := < > <= >=
(<>)	type discret	first last pos val succ pred
range <>	type entier	= /= := < > <= >=
range <>	type entier	first last pos val succ pred
range <>	type entier	+ - * / mod rem ** abs

13.7.11 Résumé

Les paramètres formels d'une fonction Ada ainsi que le résultat ne peuvent pas être d'un type fonctionnel.

Les types Ada sont monomorphes, c'est à dire qu'ils ne contiennent pas de variable de type.

Pourtant Ada, contrairement à la plupart des langages impératifs classiques, fournit le moyen de construire certaines fonctionnelles et certaines fonctions polymorphes grâce au concept de fonction générique.

Une fonction générique Ada est une fonction dont les paramètres formels sont séparés en deux groupes :

- les paramètres génériques
- les paramètres usuels.

Un paramètre générique peut être :

- un paramètre usuel

```
<id_param>: <type_param>;
```

- un paramètre fonctionnel

```
with function <id_f>  (p1:t1;..;pn:tn ) return type is <>;
```

- un paramètre de type

Le polymorphisme est rendu possible par la déclaration explicite des paramètres de type .

La spécialisation ultérieure d'un paramètre de type dépend de la manière dont il a été déclaré.

Un paramètre de type peut être spécialisé par n'importe quel type s'il est déclaré `private`:

```
<id_type> is private ;
```

Un paramètre de type peut être spécialisé par un type discret s'il est déclaré:

```
<id-type> is (<>);
```

Un paramètre de type peut être spécialisé par un type entier s'il est déclaré :

```
<id-type> is range (<>);
```

L'évaluation de la déclaration d'une fonction générique est semblable à celle d'une fonction.

L'utilisation d'une fonction générique passe nécessairement par son instanciation.

L'instanciation d'une fonction générique crée une nouvelle fonction dont l'identificateur est lié à l'application partielle de la fonction générique aux paramêtres génériques effectifs.

La généricité suggère une technique de programmation permettant d'augmenter la réutilisabilité du code. Il s'agit, à partir de la définition d'une fonction, de réaliser des abstractions sur les domaines d'application de la fonction (paramètre de type) ou sur les opérateurs qu'elle utilise (paramètres fonctionnels).

Cette généralisation permet une factorisation du code applicable dans des contextes variés après spécialisation des paramètres.

13.8 Procédures génériques

13.8.1 Syntaxe

```
<déclaration_procedure_générique>::=
generic
<paramètres_génériques>
procedure <id_p> {(<id_p>:<mode><id_t>;...;<id_p>:<mode><id_t>)};
```

avec

```
<mode>::=in|out|in out
```

13.8.2 Exemple : déclarations

```
generic
  type T is private;
procedure echange_generic(a,b:in out T);

procedure echange_generic(a,b:in out T)is
  z:T:=a;
begin
  a:=b;
  b:=z;
end echange_generic;
```

13.8.3 Exemple : Instanciations

```
procedure echanger is new echange_generic(Integer);
procedure echanger is new echange_generic(Character);
```

Remarques : Ces deux déclarations sont surchargées.

13.8.4 Exemple

```
generic
  type T is private;
  type R is private;
  with function "+"(x:T;y:R) return R is <>;
  with function F(x:T) return T is <>;
procedure accumule(x:T;accu:in out R;n:Natural);

procedure accumule(x:T;accu:in out R;n:Natural) is
  y:T:=x;
begin
  for I in 1..n loop
    y:=F(y);
    accu:=y+accu;
  end loop;
end accumule;
```

Rappels

Un type générique déclaré `private` peut être spécialisé par n'importe quel type.

Seules les opérations `=` et `/=` sont prédéfinies sur un type `private`.

13.8.5 Paquetages génériques

On peut abstraire les types et les opérateurs utilisés dans un module

Déclaration : Syntaxe

```
<déclaration_paquetage_générique>::=
generic
<paramètres_génériques>
package <id_module> is
<liste_déclarations>
private
<liste_ déclarations>
end <id_module>;
```

Les paramètres formels génériques ont la même forme syntaxique que les paramètres génériques des fonctions ou procédures.

Déclaration : Sémantique L'évaluation d'une déclaration de module générique dans un environnement `Env` étend ce dernier de la liaison `(nom du module, fermeture du module)` en 4 étapes:

1. Ajout de la liaison `(nom module, ??)` dans `Env`
2. Détermination du type du module à partir du type des paramètres formels et du type de sa valeur d'environnement
3. Détermination de la fermeture du module à partir de sa spécification et de son corps;l'environnement de définition étant celui du corps (`Env'` par exemple).
4. Modification de la liaison : `??` remplacé par `F`

```
F=(nom du module générique,
            <<g1,g2,..,gn->code du module générique,Env'>>)
```

avec,

$g_1, g_2, .., g_n$ les identificateurs des paramètres génériques

Remarques

- 1/ et 2/ réalisés à partir de la spécification du module
- 3/ et 4/ réalisés à partir de la spécification du module et de l'implantation (corps du module)

13.8.6 Instanciation

Pour utiliser un module générique, il faut l'instancier, c'est à dire préciser à quellles valeurs de paramètres génériques on veut l'appliquer.

Rappel: un paramètre générique peut être:

- un paramètre usuel
- un paramètre fonctionnel
- un paramètre de type

Syntaxe

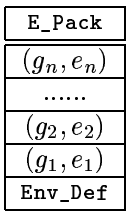
<instanciation_module_générique>::=
package <id_module_instancié> is new
<id_module_gen>(<liste_params_gen_effectifs>);
<liste_params_gen_effectifs>::=$e_1, e_2, ..., e_n$

avec e_i les paramètres génériques effectifs

Cette construction syntaxique est une déclaration

Sémantique L'évaluation d'une instanciation dans `Env` se fait en 3 étapes :

1. Ajout de la liaison `(identificateur du module instancié, ??)` dans `Env`
2. Calcul de la valeur de module, c'est à dire de l'environnement `E_Pack`, à partir de la fermeture du module générique. Soit `Env_def` l'environnement de définition du module générique contenu dans sa fermeture `E_Pack`
3. Modification de la liaison : `??` remplacé par `E_Pack` dans `Env`



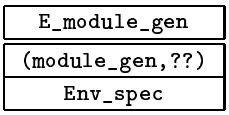
13.8.7 Exemple

Déclaration du module `module_gen`

Dans `Env`,

```
generic
  type T is private;
  with function F(x:T) return Character is <>;
  N:Integer:=20;
package module_gen is
  subtype Entier is Integer range 1..N;
  type T1 is private;
  function getC1(x:T1) return Character;
  function getC2(x:T1) return Natural;
  procedure setC1(x:in T;y:out T1);
  procedure setC2(x:in Entier;y:out T1);
private
  type T1 is
    record
      C1:T;
      C2:Entier;
    end record;
end module_gen;
```

L'évaluation de cette déclaration produit l'environnement `E_module_gen`



Le type de `module_gen` est :

```
T*(T->T)->[T1,T->T1,T1->T]
```

Soit `Env_Def`, l'environnement au moment de la déclaration du corps du module `module_gen`.

```
package body module_gen is
  function getC1(x:T1) return Character is
  begin
    return F(x.C1);
  end getC1;
  function getC2(x:T1) return Natural is
  begin
    return x.C2**2;
  end getC2;
  procedure setC1(x:in T;y:out T1) is
  begin
    y.C1:=x;
  end setC1;
  procedure setC2(x:in Entier;y:out T1) is
  begin
    y.C2:=x;
  end setC2;
end module_gen;
```

L'environnement `Env'` après évaluation du corps est :

Env'
(module_gen,F)
Env_spec

avec

```
F=<<T->F->module_gen,Env_Def>>
```

Instanciation du module `module_gen` dans `Env1`

```
with module_gen;
with Ada.Text_io;use Ada.Text_io;
with Ada.Integer_Text_io;
use Ada.Integer_Text_io;
procedure testmodule_gen is
  subtype Chaine is String(1..3);
  function car(x:Chaine) return Character is
  begin
    return x(1);
  end car;
  package monModule is new module_gen(Chaine,car);
  use monModule;
  data:T1;
begin
  setC1("abc",data);
  put_line("C1 = abc");
  setC2(15,data);
  put_line("C2 = 15");new_line;
  put("car(data.C1)=");put(getC1(data));
  new_line;
  put("data.C2**2=");put(getC2(data),width=>3);
end testmodule_gen;
```

Evaluation de la déclaration du module `monModule` dans `Env1`

- Ajout de la liaison `(monModule,??)`
- Détermination de la valeur d'environnement du module dans l'environnement `Env2`

Env2
(N,@N)
(F,fermeture de car)
(T,Chaine)
Env_Def

- Le résultat est l'environnement `Env3`, obtenu en évaluant le corps du module générique dans l'environnement `Env2`

Env3
(setC2,<<Entier*T1->corps de setC2,Env_inst>>)
(setC1,<<Entier*T1->corps de setC1,Env_inst>>)
(getC2,<<T1->corps de getC2,Env_inst>>)
(getC1,<<T1->corps de getC1,Env_inst>>)
(T1,type de T1)
(Entier,type Entier)

- Remplacement de la valeur indéfinie `??` par `Env3`, la liaison devient `(monModule,Env3)` dans `Env1`.

13.8.8 Le paquetage listes_generales : spécification

```
generic
  type T is private;
package listesGenerales is
  LISTE_VIDE :exception;
  type Liste is private;
  function cons(ELT:T;L:Liste) return Liste;
  function vide return Liste;
  function estVide(L:Liste) return boolean;
  function tete(L:Liste) return T;
  function queue(L:Liste) return Liste;
private
  type Cellule;
  type Liste is access Cellule;
  type Cellule is
    record
      valeur:T;
      suivant:Liste;
    end record;
end listesGenerales;
```

13.9 Comment se procurer le compilateur Ada

13.9.1 Gratuitement (et pour toute plate forme)

Télécharger le compilateur *gnat* à partir des sites :

```
ftp://ftp.lip6.fr/pub/gnat
ftp://ftp.cnam.fr/pub/Ada/PAL/compiler/gnat/distrib
-- lire d'abord le fichier README
```

Le compilateur est aussi fourni sur le CD-ROM du cycle probatoire (dominante ICD).

13.10 Quelques sites Web utiles

```
Site de référence
http://www.adahome.com
Manuel de référence
http://www.adahome.com/rm95/rm9x-toc.html
```

```
Tutoriels
http://www.einev.ch/cours/ada/HOME1.htm
http://www.scism.sbu.ac.uk/law/Section1/contents.html
http://www.adahome.com/Tutorials/Lovelace/lovelace.html
http://wwwarchive.wustl.edu/languages/ada/ajpo/index-text.html
```

13.11 Bibliographie

- *Concepts et outils de programmation, T.Hardin, V.Donzeau-Gouge, InterEditions*

description des aspects sémantiques du langage

- *Programmer en Ada (4ème édition), J.P.G. Barnes, Addison-Wesley*

le livre de référence

- *Ingénierie du logiciel avec Ada, G.Booch, InterEditions*

mis en avant de l'aspect conception du logiciel

- *Vers Ada95 par l'exemple, D.Fayard, M.Rousseau, Bibliothèque des universités*

beaucoup de petits exemples

- *Ada avec le sourire, J.M.Bergé&al, Presses polytechniques romandes*

introduction à base de petits exemples